Intel® Math Kernel Library

Application Notes for Intel® Math Kernel Library Summary Statistics

Contents

Legal Information

1. About this Document

1.1. About This Document

These Application Notes illustrate how to use the Intel® MKL Summary Statistics functions when creating your applications. This document covers algorithms, interfaces, the usage models, and the most important features and performance aspects of the Summary Statistics domain. See the following documents for more information:

1. Intel® Math Kernel Library Developer Reference for details on the algorithms, interfaces, and the supported languages
2. Intel® Math Kernel Library Developer Guide for information about building and linking the application that uses Summary Statistics algorithms

1.2. Conventions and Symbols

This document uses the following conventions:

Convention	**Explanation**
`This type style`	Indicates an element of syntax, parameter name, keyword, filename, computer output, or part of a program example. The text appears in lowercase unless uppercase is significant.
This type style	Indicates the exact characters you type as input. Also used to highlight the elements of a graphical user interface such as buttons or menu names.
This type style	Indicates a placeholder for an identifier, an expression, a string, a symbol, or a value. Substitute one of these items for the placeholder.
[*items*]	Indicates that the items enclosed in brackets are optional.

{ *item* | *item* }	Indicates to select only one of the items listed between braces. A vertical bar (|) separates the items.
... (ellipses)	Indicates that you can repeat the preceding item.

2. About Summary Statistics

Optimization Notice

The Summary Statistics is a subcomponent of the Vector Statistics (VS) included into the Intel® Math Kernel Library (Intel® MKL). The Summary Statistics component offers a solution for parallel statistical processing of multi-dimensional datasets. It contains functions for initial statistical analysis of raw data. You can use these functions to investigate the structure and understand the basic characteristics and internal dependencies of the analyzed datasets.

See Also

Algorithms and Interfaces in Summary Statistics

3. *Algorithms and Interfaces in Summary Statistics*

3.1. Algorithms and Interfaces in Summary Statistics

This section discusses different methods and usage specifics of the Summary Statistics algorithms. For some methods, interfaces are described. For details on the Summary Statistics API, see [MKLMan].

See Also

Estimating Raw and Central Moments and Sums, Skewness, Excess Kurtosis, Variation, and Variance-Covariance/Correlation/Cross-Product Matrix
Computing Median Absolute Deviation
Computing Mean Absolute Deviation
Computing Minimum/Maximum Values
Calculating Order Statistics
Estimating Quantiles
Estimating a Pooled/Group Variance-Covariance Matrix
Performing Robust Estimation of a Variance-Covariance Matrix
Detecting Multivariate Outliers
Handling Missing Values in Matrices of Observations
Parameterizing a Correlation Matrix
Sorting an Observation Matrix

3.2. Estimating Raw and Central Moments and Sums, Skewness, Excess Kurtosis, Variation, and Variance-Covariance/Correlation/Cross-Product Matrix

3.2.1. Estimating Raw and Central Moments and Sums, Skewness, Excess Kurtosis, Variation, and Variance-Covariance/Correlation/Cross-Product Matrix

Summary Statistics offers the following methods to support computation of raw and central moments and sums, skewness, excess kurtosis (further referred to as kurtosis), variation, and variance-covariance/correlation/cross-product matrix:

1. Method `VSL_SS_METHOD_FAST` is a performance-oriented implementation of an algorithm for estimate calculations.
2. Method `VSL_SS_METHOD_FAST_USER_MEAN` is an implementation of an algorithm for estimate calculations when a user-defined mean is provided.
3. Method `VSL_SS_METHOD_1PASS` is an implementation of a one-pass algorithm. In this case, all requested estimates are computed for a single pass. For example, see [West79].
4. Method `VSL_SS_METHOD_CP_TO_COVCOR` is an implementation of computation of a variance-covariance and/or correlation matrix from a corresponding cross-product matrix.
5. Method `VSL_SS_METHOD_SUM_TO_MOM` is an implementation of computation of raw/central statistical moments as well as kurtosis/skewness/variation from corresponding raw/central sums.

The `VSL_SS_METHOD_FAST` method for variance-covariance estimation can be numerically unstable for some datasets, such as a dataset from Gaussian distribution with a standard deviation several orders smaller than its mean. For such datasets, to estimate variance-covariance, cross-product or another estimate relying on mean, use the one-pass algorithm supported by the library, or the two-pass algorithm [West79], whose building blocks are available in the library. In the latter case, you need to do the following:

1. Compute the mean using Summary Statistics functions.
2. Compute the variance-covariance, cross-product or another estimate by providing the computed mean and applying the `VSL_SS_METHOD_FAST_USER_MEAN` method.

Each estimate is stored as a one-dimensional array. The size of the array may differ depending on the type of the estimate, as follows:

Estimate Type	**Size of the Array**
1. Raw and central moments 2. Raw and central sums 3. Kurtosis 4. Skewness 5. Variation	Must be sufficient to store at least *p* elements, where *p* is the dimension of the task.
1. Variance-covariance matrix 2. Correlation matrix 3. Cross-product matrix	Depends on the storage format. For details, see Table Storage formats of a variance-covariance/correlation/cross-product matrix in the Summary Statistics section of [MKLMan].

3.2.2. Computing Estimations for Large Datasets

Summary Statistics algorithms can compute estimates for large datasets, including datasets in blocks. Different Summary Statistics computation methods may require additional memory for correct processing of the blocks. The `VSL_SS_METHOD_FAST` method uses raw moments of different orders:

Computation Type	**Raw Moment Orders**
The central moment of order *i>1*	*1,...,i*
The central sum of order *i>1*	*1,...,i*
Kurtosis coefficient (a function of central moments of the second and fourth order)	1,2,3,4
Skewness coefficient	1,2,3
Variation coefficient	1,2
Variance-covariance/correlation matrix	The mean (the first raw moment)
Cross-product matrix	The mean (the first raw moment)

To support data arrays in blocks and compute their statistical estimates, the library requires buffers to store intermediate results. For this purpose, you need to allocate arrays and make them available to the library via task editors. These arrays store the value of the requested parameter and intermediate estimates of raw moments. The number of buffers necessary for intermediate results corresponds to the maximal order of raw moments required to calculate an estimate. The size of each buffer should store at least *p* elements, where *p* is the dimension of the task.

For example, to compute the skewness coefficient, you should allocate a buffer for the requested estimate and three one-dimensional arrays of size *p* to store the values of raw moments up to the third order. These intermediate results are required to process the next data portion and get the skewness estimate for the whole dataset.

Before the first call to the `Compute` routine, you should allocate and initialize necessary arrays and pass pointers into the library using one of the available editors. In most cases, elements of the arrays are initialized to zero. If you already have the estimates for the previous data portion, you can use these estimates to initialize the elements and continue the computation.
If there is no available memory for storing raw moments, the computation terminates with a corresponding error code.

The `VSL_SS_METHOD_1PASS` method relies on the mean estimate. Before using the method for processing data in blocks, you should allocate a buffer for mean and make it available to the library via task editors.
The `VSL_SS_METHOD_FAST_USER_MEAN` method relies on the mean estimate that you provide to the library before computation. You do not need additional buffers for this method.
You can obtain a correlation matrix from a variance-covariance matrix using a proper standardization. This scaling does not impact the main diagonal of the correlation matrix, that is, the matrix stores the variances of the random vector components. For details, see the Mathematical Notation and Definitions chapter in the Summary Statistics section of [MKLMan].When you compute matrices for the next block in the dataset, the library applies the following steps:

1. Converts the variance-covariance and/or correlation matrix into a cross-product matrix
2. Updates the cross-product matrix using the requested computation method

3. Updates a variance-covariance and/or correlation matrix from the cross-product matrix computed on step two.

If you compute a variance-covariance and/or correlation matrix, the library applies all the three steps. If you compute cross-product, variance-covariance and/or correlation matrices at the same time, the library applies only step two and step three. The library applies the same computation rule to raw/central sums and statistical moments up to the fourth order.

For the best results, apply the following steps, before you compute the matrices:

Computation	**Preparation**
Variance-covariance/correlation matrix	Provide the buffer for the cross-product matrix of the full storage format
Statistical moments	Allocate and register buffers for statistical sums of the corresponding order

With Intel MKL Summary Statistics algorithms, you can compute variance-covariance and/or correlation matrices for the dataset available as *n* blocks using the following technique:

1. For blocks 1,2,..., n of the dataset, call Summary Statistics algorithm for computation of a cross-product matrix using one of supported methods
2. Convert the final cross-product matrix into a variance-covariance and/or correlation matrix by applying `VSL_SS_METHOD_CP_TO_COVCOR`

The example below demonstrates these steps:

```
#include "mkl_vsl.h"
#define NBLOCKS 10 /* number of blocks in the dataset      */
#define DIM 3      /* dimension of the task                */
#define N   1000   /* number of observations in each block */
int main()
{

    int i;
    VSLSSTaskPtr task;
    double x[DIM][N];  /* matrix of data block */
    double cp[DIM*DIM], cor[DIM*DIM], mean[DIM];
    double w[2];
    MKL_INT p, n, xstorage, corstorage, cpstorage;
    int status;

    /* Parameters of the task and initialization */
    p = DIM;
    n = N;
    xstorage   = VSL_SS_MATRIX_STORAGE_ROWS;
    corstorage = VSL_SS_MATRIX_STORAGE_FULL;
    cp         = VSL_SS_MATRIX_STORAGE_FULL;

    w[0] = 0.0; /* sum of weights */
    w[1] = 0.0; /* sum of squares of weights */
    for ( i = 0; i < p  ; i++ ) mean[i] = 0.0;
    for ( i = 0; i < p*p; i++ )   cp[i] = 0.0;

    /* Create a task */
    status = vsldSSNewTask( &task, &p, &n, &xstorage, x, 0, 0 );

    /* Initialize the task parameters */
    status = vsldSSEditCP  ( task, mean, NULL, cp, &cpstorage );
    status = vsldSSEditTask( task, VSL_SS_ED_COR, cor );
    status = vsldSSEditTask( task, VSL_SS_ED_COR_STORAGE, corstorage );
    status = vsldSSEditTask( task, VSL_SS_ED_ACCUM_WEIGHT, w );

    /* Compute a cross-product matrix for NBLOCKS-1 */
    for( i = 0; i < NBLOCKS; i++ )
    {
        /* Get i-th data block to array x */
        status = GetBlock( i, x, p, n );

        /* Update cross-product matrix using latest block */
        status = vsldSSCompute( task, VSL_SS_CP, VSL_SS_METHOD_1PASS );
    }
    /* Convert cross-product matrix into correlation matrix */
    status = vsldSSCompute( task, VSL_SS_COR,
                              VSL_SS_METHOD_CP_TO_COVCOR);

    /* Deallocate the task resources */
    status = vslSSDeleteTask( &task );

    return 0;
}
```

When converting the cross-product matrix computed for a given data array into a variance-covariance/correlation matrix, the library applies the following rule:

1. If the array of accumulated weights is available, the library computes the standardization coefficient using values in this array
2. Otherwise, the library computes the standardization coefficient using the number of observations passed as the input parameter to the Summary Statistics task constructor.

If you need to convert a computed cross-product matrix into a variance-covariance and/or correlation matrix, use the task editors to provide pointers to the matrices, their storage formats, and the array of accumulated weights to the library. Afterwards, call the SSCompute routine function to convert a cross-product matrix into a variance-covariance/correlation matrix as shown below:

```
    status = vsldSSEditTask( task, VSL_SS_ED_COR, cor );
    status = vsldSSEditTask( task, VSL_SS_ED_COR_STORAGE, corstorage );
    status = vsldSSEditTask( task, VSL_SS_ED_CP, cp );
    status = vsldSSEditTask( task, VSL_SS_ED_CP_STORAGE, cpstorage );
    status = vsldSSCompute( task, VSL_SS_COR,
                              VSL_SS_METHOD_CP_TO_COVCOR );
```

You can convert a cross-product matrix into both variance-covariance and correlation matrices by specifying two estimates, the conversion method and one call to the Compute function:

```
    status = vsldSSCompute( task, VSL_SS_COV|VSL_SS_COR,
                              VSL_SS_METHOD_CP_TO_COVCOR );
```

If you provide a combination of the conversion method and another supported method (for example, fast) to compute a variance-covariance/correlation matrix, the library discards the conversion method and applies another (for example, fast) method to produce the estimate.

You can apply similar approaches to compute raw/central statistical moments from raw/central sums of the corresponding order:

```
    status = vsldSSCompute( task, VSL_SS_3R_MOM| VSL_SS_4C_MOM,
                              VSL_SS_METHOD_SUM_TO_MOM );
```

By applying the conversion method `VSL_SS_METHOD_SUM_TO_MOM`, you can also get estimates of kurtosis/skewness/variation coefficient, given available buffers for the moments described in the table above.

3.2.3. Calculating Multiple Estimates

Using Summary Statistics, you can calculate several estimates at a time. In this case, the maximal order of the raw moment required for the computation is defined by the computation method and determines the number of arrays to hold the raw moments. For details, see Computing Estimations for Large Datasets.

To compute a new estimate for the next data portion, you need to allocate, initialize, and pass into the library additional buffers before calling the `Compute` routine. You can also compute a specific estimate for the next data portion in the environment of a new task.
Summary Statistics provides unbiased estimates for the central moment of the second order and a variance-covariance matrix with standardizing coefficient:

$$B = W - \sum_{j=1}^{n} w_j^2 / W$$

where

$$W = \sum_{i=1}^{n} w_i$$

For details, see the Mathematical Notation and Definitions chapter in the Summary Statistics section of [MKLMan].

Before the first call to the `Compute` routine, you should initialize the elements of the array with zeros or any other values that meet the requirements of the application.
To ensure correct computation of the estimates, you need to pass a pointer to the *WA* array of two elements:

1. The first element of the array holds the sum of weights assigned to the observations $\sum_{i=1}^{n} w_i$
2. The second element contains the sum of squares of the weights:
   $\sum_{i=1}^{n} w_i^2$

If the whole matrix of observations is available at once and no other data portions are expected, passing a pointer to the *WA* array is unnecessary.

Estimates of the third and fourth central moments provided by the library are biased and require the sum of weights only, which is the first element of the array described above.

The following example illustrates calculation of the central sum of second order (sums of squares) and a correlation matrix:

```
#include "mkl_vsl.h"
#define DIM 3    /* dimension of the task */
#define N   1000 /* number of observations */
int main()
{

    int i;
    VSLSSTaskPtr task;
    double x[DIM][N];  /* matrix of observations */
    double cor[DIM*(DIM+1)/2], mean[DIM];
    double w[2];
    MKL_INT p, n, xstorage, corstorage;
    int status;

    /* Parameters of the task and initialization */
    p = DIM;
    n = N;
    xstorage    = VSL_SS_MATRIX_STORAGE_ROWS;
    corstorage = VSL_SS_MATRIX_STORAGE_U_PACKED;

    w[0] = 0.0; /* sum of weights */
    w[1] = 0.0; /* sum of squares of weights */
    for ( i = 0; i < p; i++ ) mean[i] = 0.0;
    for ( i = 0; i < p*(p+1)/2; i++ ) cor[i] = 0.0;

    /* Create a task */
    status = vsldSSNewTask( &task, &p, &n, &xstorage, x, 0, 0 );

    /* Initialize the task parameters */
    status = vsldSSEditTask( task, VSL_SS_ED_ACCUM_WEIGHT, w );
    status = vsldSSEditCovCor( task, mean,
                               NULL, NULL, cor, &corstorage );

    /* Compute a correlation matrix */
    status = vsldSSCompute( task, VSL_SS_COR, VSL_SS_METHOD_1PASS );

    /* Deallocate the task resources */
    status = vslSSDeleteTask( &task );

    return 0;
}
```

3.3. Computing Median Absolute Deviation

Use the `VSL_SS_METHOD_FAST` method to compute a median absolute deviation estimate in the datasets. The calculation is straightforward and follows the pattern of the example below:

```
#include "mkl_vsl.h"

#define DIM 3      /* dimension of the task */
#define N   1000   /* number of observations */

int main()
{
    VSLSSTaskPtr task;
    float x[DIM][N];  /* matrix of observations */
    float mdad[DIM];
    MKL_INT p, n, xstorage;
    int status;

    /* Parameters of the task and initialization */
    p = DIM;
    n = N;
    xstorage = VSL_SS_MATRIX_STORAGE_ROWS;

    /* Create a task */
    status = vslsSSNewTask( &task, &p, &n, &xstorage, (float*)x, 0, 0 );
    /* Initialize the task parameters */
    status = vslsSSEditTask( task, VSL_SS_ED_MDAD, mdad );

    /* Compute median absolute deviation in observations */
    status = vslsSSCompute(task, VSL_SS_MDAD, VSL_SS_METHOD_FAST );

    /* Deallocate the task resources */
    status = vslSSDeleteTask( &task );

    return 0;
}
```

The size of the array to hold median absolute deviation should be sufficient for storing at least *p* values of the estimate, where *p* is the dimension of the task.

Computation of median absolute deviation is only possible for data arrays available at once, or in separate blocks of the dataset.

3.4. Computing Mean Absolute Deviation

Summary Statistics offers two methods for computation of mean absolute deviation:

1. Method VSL_SS_METHOD_FAST is a performance-oriented implementation of the algorithm.
2. Method VSL_SS_METHOD_FAST_USER_MEAN is an implementation of the algorithm when a user-defined mean is provided.

The calculation is straightforward and follows the pattern of the example below:

```
#include "mkl_vsl.h"

#define DIM 3      /* dimension of the task */
#define N   1000   /* number of observations */

int main()
{
    VSLSSTaskPtr task;
    float x[DIM][N];  /* matrix of observations */
    float mnad[DIM];
    MKL_INT p, n, xstorage;
    int status;

    /* Parameters of the task and initialization */
    p = DIM;
    n = N;
    xstorage = VSL_SS_MATRIX_STORAGE_ROWS;

    /* Create a task */
    status = vslsSSNewTask( &task, &p, &n, &xstorage, (float*)x, 0, 0 );

    /* Initialize the task parameters */
    status = vslsSSEditTask( task, VSL_SS_ED_MNAD, mnad );

    /* Compute median absolute deviation in observations */
    status = vslsSSCompute(task, VSL_SS_MNAD, VSL_SS_METHOD_FAST );

    /* Deallocate the task resources */
    status = vslSSDeleteTask( &task );
    return 0;
}
```

The size of the array to hold mean absolute deviations should be sufficient to hold at least p elements, where p is the dimension of the task.

Computation of mean absolute deviation is only possible for data arrays available at once, or in separate blocks of the dataset.

To achieve the best results, before you compute mean absolute deviation, provide the buffer for estimate of mean (or corresponding sum) even if you do not need this estimate.

3.5. Computing Minimum/Maximum Values

Use the `VSL_SS_METHOD_FAST` method to compute the minimum/maximum values in the datasets. The calculation is straightforward and follows the pattern of the example below:

```
#include "mkl_vsl.h"

#define DIM 3      /* dimension of the task */
#define N   1000   /* number of observations */

int main()
{
    VSLSSTaskPtr task;
    float x[DIM][N];  /* matrix of observations */
    float min_est[DIM], max_est[DIM];
    MKL_INT p, n, xstorage;
    int status;

    /* Parameters of the task and initialization */
    p = DIM;
    n = N;
    xstorage = VSL_SS_MATRIX_STORAGE_ROWS;
    for ( int i = 0; i < p; i++ ) min_est[i] = max_est[i] = x[i][0];

    /* Create a task */
    status = vslsSSNewTask( &task, &p, &n, &xstorage, (float*)x, 0, 0 );

    /* Initialize the task parameters */
    status = vslsSSEditTask( task, VSL_SS_ED_MIN, min_est );
    status = vslsSSEditTask( task, VSL_SS_ED_MAX, max_est );

    /* Compute the minimum and maximum values in observations */
    status = vslsSSCompute( task, VSL_SS_MIN|VSL_SS_MAX,
                                  VSL_SS_METHOD_FAST );

    /* Deallocate the task resources */
    status = vslSSDeleteTask( &task );

    return 0;
}
```

The size of the arrays to hold the minimum/maximum values should be sufficient for storing at least *p* values of each estimate, where *p* is the dimension of the task.

You can use the computation of these estimates to find the minimum/maximum values in the dataset available in blocks. In this case, the estimates computed for the previous data portion are used for processing the next block of the data array.

Before the first call to the `Compute` routine, initialize the initial values of the estimates with reasonable values, such as the values of the first observation.

3.6. Calculating Order Statistics

Order statistics is stored as a one-dimensional array. To hold results of the calculations, the size of this array should be at least *m*n*

where

1. *m* is the number of vector components to process.
2. *n* is the number of observations.

The calculation results are packed according to the value of the `ostatsstorage` variable. For the supported storage formats, please see table Storage format of matrix of observations and order statistics in the Summary Statistics section of [MKLMan].

3.7. Estimating Quantiles

3.7.1. Estimating Quantiles

You can use the Summary Statistics routines to compute quantiles for a matrix of observations. The computation routine can calculate more than one quantile at a time. Quantile orders belonging to the interval (0,1) are packed and passed into the library as an array. You should allocate enough memory to hold results of the calculations. The size of the array should provide storage for at least *d*p* elements, where

1. *p* is the dimension of the task.
2. *d* is the number of the requested quantiles.

Quantiles in the array are packed component by component, starting from the first component of the random vector and following the quantile orders.

See the Mathematical Notation and Definitions chapter in the Summary Statistics section of [MKLMan] for additional information.

The example below illustrates quantile-related calculations:

```
#include "mkl_vsl.h"
#include <stdio.h>
#define DIM 3      /* dimension of the task */
#define N   1000   /* number of observations */
#define M   100    /* number of quantiles to compute */

int main()
{
   int i, status;
   VSLSSTaskPtr task;
   float x[DIM][N];       /* matrix of observations */
   float order_stats[N]; /* matrix to store order statistics */
   float q_order[M], quants[M];
   MKL_INT q_order_n;
   MKL_INT p, n, xstorage, ostatstorage;
   unsigned long long estimates;
   int indices[DIM]={1,0,0}; /* the first vector component is processed */
   /* Parameters of the task and initialization */
   p = DIM;
   n = N;
   q_order_n = M;
   xstorage  = VSL_SS_MATRIX_STORAGE_ROWS;
   ostatstorage = VSL_SS_MATRIX_STORAGE_ROWS;

   /* Calculate percentiles */
   for ( i = 0; i < M; i++ ) q_order[i] = (float)i / (float)M;

   /* Create a task */
   status = vslsSSNewTask( &task, &p, &n,
                           &xstorage, (float*)x, 0, indices );

   /* Initialize the task parameters */
   status = vslsSSEditQuantiles( task, &q_order_n, q_order,
                                 quants, order_stats, &ostatstorage );

   /* Compute the percentiles and order statistics */
   estimates = VSL_SS_QUANTS|VSL_SS_ORDER_STATS;
   status = vslsSSCompute( task, estimates, VSL_SS_METHOD_FAST );

   /* Deallocate the task resources */
   status = vslSSDeleteTask( &task );

   return 0;
}
```

3.7.2. Computing Quantiles for Streaming Data with VSL_SS_METHOD_SQUANTS_ZW

Use the `VSL_SS_METHOD_SQUANTS_ZW` method to compute quantiles for streaming data [Zhang2007]. The algorithm supports two usage models of data processing for a given task dimension:

1. Dataset is available as a single block. The number of observations is known by the time you pass the data into the library.
2. Dataset is available as a sequence of blocks. The number of observations in each chunk is available prior to passing the data into the library. The number of observations can vary from block to block. The number of blocks may be unknown in advance.

The accuracy of quantile estimation ε, which is a parameter of the algorithm, is packed into the `params` array and passed to the library using the `EditStreamQuantiles` editor. Before computation, the algorithm checks the value of ε. If $\varepsilon \le 0$ or $\varepsilon > 1$, the library terminates the computation and returns the `VSL_SS_ERROR_BAD_STREAM_QUANT_PARAMS` error status. For correct values of ε, the algorithm returns a quantile estimate located in the interval $[r-\varepsilon n, r+\varepsilon n]$
where

1. r is the rank of the desired quantile.
2. n is the total number of available observations.

For details, see [Zhang2007].

If a single data block is available, the quantiles are computed as shown in the example below:

```
#include "mkl_vsl.h"
#include <stdio.h>
#define DIM 3       /* dimension of the task */
```

```
#define N   1000   /* number of observations */
#define M   100    /* number of quantiles to compute */
#define EPS 0.01   /* accuracy of quantile computation */

int main()
{
   int i, status;
   VSLSSTaskPtr task;
   float x[DIM][N];  /* matrix of observations */
   float q_order[M], quants[M];
   float params;
   MKL_INT q_order_n;
   MKL_INT p, n, nparams, xstorage;
   int indices[DIM]={1,0,0}; /* the first vector component is processed */

   /* Parameters of the task and initialization */
   p = DIM;
   n = N;

   q_order_n = M;
   xstorage  = VSL_SS_MATRIX_STORAGE_ROWS;
   params    = EPS;
   nparams   = VSL_SS_SQUANTS_ZW_PARAMS_N;

   /* Calculate percentiles */
   for ( i = 0; i < M; i++ ) q_order[i] = (float)i / (float)M;

   /* Create a task */
   status = vslsSSNewTask( &task, &p, &n, &xstorage, x, 0, indices );

   /* Initialize the task parameters */
   status = vslsSSEditStreamQuantiles( task, &q_order_n, q_order,
                                       quants, &nparams, &params );

   /* Compute the percentiles with accuracy eps */
   status = vslsSSCompute( task, VSL_SS_STREAM_QUANTS,
                                 VSL_SS_METHOD_SQUANTS_ZW );

   /* Deallocate the task resources */
   status = vslSSDeleteTask( &task );

   return 0;
}
```

If the dataset is available as a sequence of blocks, the quantiles are computed as follows:

```
#include "mkl_vsl.h"
#include <stdio.h>
#define DIM 3     /* dimension of the task */
#define N   1000  /* number of observations */
#define M   100   /* number of quantiles to compute */
#define EPS 0.01  /* accuracy of quantile computation */
#define NBLOCKS 5 /* number of data blocks of size N */

int main()
{
   int i, status;
   VSLSSTaskPtr task;
   float x[DIM][N];  /* matrix of observations */
   float q_order[M], quants[M];
   float params;
   MKL_INT q_order_n;
   MKL_INT p, n, nparams, xstorage;
   int indices[DIM]={1,0,0}; /* the first vector component is processed */

   /* Parameters of the task and initialization */
   p = DIM;
   n = N;
   q_order_n = M;
   xstorage  = VSL_SS_MATRIX_STORAGE_ROWS;
   params    = EPS;
   nparams   = VSL_SS_SQUANTS_ZW_PARAMS_N;

   /* Calculate percentiles */
   for ( i = 0; i < M; i++ ) q_order[i] = (float)i / (float)M;

   /* Create a task */
   status = vslsSSNewTask( &task, &p, &n, &xstorage, x, 0, indices );

   /* Initialize the task parameters */
   status = vslsSSEditStreamQuantiles( task, &q_order_n, q_order,
                                    quants, &nparams, &params );
   for ( i = 0; ; i++ )
   {
      /* Update the internal data structures of the algorithm */
       status = vslsSSCompute( task, VSL_SS_STREAM_QUANTS,
                                  VSL_SS_METHOD_SQUANTS_ZW FAST );
       if ( ++i >= NBLOCKS ) break;
       GetNextDataBlock( x ) ;
   }

    /* Compute the percentiles with accuracy eps */
    n = 0;
    status = vslsSSCompute( task, VSL_SS_STREAM_QUANTS,
                               VSL_SS_METHOD_SQUANTS_ZW );

    /* Deallocate the task resources */
    status = vslSSDeleteTask( &task );

    return 0;
}
```

If intermediate quantile estimates are not required, you can analyze your data using the fast computation method `VSL_SS_METHOD_SQUANTS_ZW_FAST`. In this mode, the algorithm only updates internal data structures when processing the next available block. Actual estimates of the required quantiles are computed after the whole sequence of the blocks is processed. An additional call to the `Compute` routine with method `VSL_SS_METHOD_SQUANTS_ZW` returns the final estimate. Before making the final call to the routine, set to zero the variable that holds the number of observations and is registered in the library.
If intermediate estimates of quantiles are required, use the `VSL_SS_METHOD_SQUANTS` method. In this case, you do not need the additional call to the `Compute` routine:

```
#include "mkl_vsl.h"

#include <stdio.h>
#define DIM 3      /* dimension of the task */
#define N   1000   /* number of observations */
#define M   100    /* number of quantiles to compute */
#define EPS 0.01   /* accuracy of quantile computation */
```

```
#define NBLOCKS 5  /* number of data blocks of size N */

int main()
{
    int i, status;
    VSLSSTaskPtr task;
    float x[DIM][N];  /* matrix of observations */
    float q_order[M], quants[M];
    float params;
    MKL_INT quant_order_n;
    MKL_INT p, n, n_params, xstorage;
    int indices[DIM]={1,0,0}; /* the first vector component is processed */

    /* Parameters of the task and initialization */
    p = DIM;
    n = N;
    q_order_n = M;
    xstorage  = VSL_SS_MATRIX_STORAGE_ROWS;
    params    = EPS;
    params_n  = VSL_SS_SQUANTS_ZW_PARAMS_N;

    /* Calculate percentiles */
    for ( i = 0; i < M; i++ ) q_order[i] = (float)i / (float)M;

    /* Create a task */
    status = vslsSSNewTask( &task, &p, &n, &xstorage, x, 0, indices );

    /* Initialize the task parameters */
    status =  vslsSSEditStreamQuantiles( task, &q_order_n, q_order,
                                         quants, &n_params, &params );

    for ( i = 0; ; i++ )
    {
        /* Compute the percentiles with accuracy eps */
        status = vslsSSCompute( task, VSL_SS_STREAM_QUANTS,
                                      VSL_SS_METHOD_SQUANTS_ZW );
        if ( ++i >= NBLOCKS ) break;
        GetNextDataBlock( x ) ;
    }

     /* Deallocate the task resources */
    status = vslSSDeleteTask( &task );

    return 0;
}
```

3.8. Estimating a Pooled/Group Variance-Covariance Matrices/Means

Use the `VSL_SS_METHOD_1PASS` method to compute pooled/group variance-covariance matrices, or pooled/group means.
For the definition of pooled/group variance-covariance matrices, see the Mathematical Notation and Definitions chapter in the Summary Statistics section of [MKLMan].

To compute a pooled variance-covariance and/or a pooled mean, you need to split observations into *g* groups by allocating array `grp_indices` of size *n,* where *n* is the number of observations. Indices of the groups take values from the range [0,1, … *g*-1]. Thus, `grp_indices[j]` = *k* if observation *j* belongs to the group indexed *k*. The pooled variance-covariance matrix is packed as a one-dimensional array. For information on available storage formats and memory requirements, see Table Storage formats of a variance-covariance/correlation matrix of the Summary Statistics section of [MKLMan]. The pooled mean estimate is returned in the array that should store at least *p* elements, where *p* is the dimension of the task.

You can get estimates for group variance-covariance matrices and/or group means by passing into the library the array `grp_cov_indices` of size *g*. This array determines the group variance-covariance matrices and/or means to be returned:

1. If the group variance-covariance matrix and/or the vector of means are to be returned, `grp_cov_indices[idx]` = 1.
2. Otherwise, `grp_cov_indices[idx]` = 0.

The estimates for group variance-covariance matrices and group means are stored in one-dimensional arrays `grp_cov` and `grp_means`, respectively.
The group means are packed in the `grp_means` array in series. The size of the array should be sufficient for at least *p***k* elements,
where

1. *p* is the dimension of the task.
2. *k* is the number of group matrices to be returned.

Group matrices are packed in the `grp_cov` array in series according to the contents of the array `grp_cov_indices`. The size of the `grp_cov` array should be sufficient for at least `cov_dim`**k*
where

1. `cov_dim` is the size of a single group matrix defined by the chosen storage format.
2. *k* is the number of group matrices to be returned.

The library checks that the initialization of the `grp_indices` pointer is correct and the values stored in the array are positive. If the initialization is wrong, computation of pooled/group variance-covariance matrix terminates with an error code. In this case, you need to make sure that the `grp_indices` array contains all values from 0 to *g*-1 inclusively, and the memory allocated for the `grp_cov_indices` array is sufficient to hold at least *g* values.
The example below shows pooled/group variance-covariance matrices that you can get:

```
#include "mkl_vsl.h"

#define DIM 3      /* dimension of the task */
#define N   1000   /* number of observations */
#define G   2      /* number of groups */
#define GN  2      /* number of group variance-covariance matrices */

int main()
{
   int i;
   VSLSSTaskPtr task;
   double g_indices[N];            /* indices of the groups */
   double x[N][DIM];               /* matrix of observations */
   double g_cov_indices[G]={1,1}; /* two group matrices to be returned */

   double pcov[DIM*DIM];           /* pooled variance-covariance matrix */
   double pmean[DIM];              /* array of pooled means */
```

```
    double gcov[DIM*DIM*GN];       /* array for group variance-covariance matrices
*/
    double gmean[DIM*GN];          /* array for group means */
    int status;

    MKL_INT p, n, xstorage, pcovstorage, gcovstorage;
    unsigned long long estimates;

    /* Parameters of the task and initialization */
    p = DIM;
    n = N;
    xstorage = VSL_SS_MATRIX_STORAGE_COLS;
    pcovstorage = VSL_SS_MATRIX_STORAGE_FULL;
    gcovstorage = VSL_SS_MATRIX_STORAGE_FULL;

    /* The first N/2 elements belong to the first group, the rest belong to the
second group */
    for ( i = 0; i < N/2; i++ )
    {
        g_indices[i+0] = 0; g_indices[i+N/2] = 1;
    }

    /* Create a task */
    status = vslsSSNewTask( &task, &p, &n, &xstorage, x, 0, 0 );

    /* Initialize the task parameters */
    status = vslsSSEditTask( task, VSL_SS_ED_POOLED_COV_STORAGE,
                             &pcovstorage );
    status = vslsSSEditTask( task, VSL_SS_ED_GROUP_COV_STORAGE,
                             &gcovstorage );
    status = vsldSSEditPooledCovariance( task, g_indices, pmean,
                                         pcov, g_cov_indices, gmean, gcov );

    /* Compute the pooled and group variance-covariance matrices */
    estimates = VSL_SS_POOLED_COV|VSL_SS_GROUP_COV;
    status = vsldSSCompute( task, estimates, VSL_SS_METHOD_1PASS );

    /* Deallocate the task resources */
    status = vslSSDeleteTask( &task );

    return 0;
 }
```

Computation of pooled/group variance-covariance matrices does not support datasets available in blocks.

3.9. Estimating a Partial Variance-Covariance Matrix

Use the `VSL_SS_FAST_METHOD` method to compute a partial variance-covariance matrix .
For the definition of a partial variance-covariance matrix, see the Mathematical Notation and Definitions chapter in the Summary Statistics section of [MKLMan].

To calculate the matrix, provide a variance-covariance matrix and split the random vector $\xi = (\xi_1,\ldots,\xi_7)$ of dimension p into two non-overlapping sub-components, Y and Z. Each component is encoded as follows:

$$partial\_index[i] = \begin{cases} -1, if\ \xi_i \in Z \\ 1, if\ \xi_i \in Y \end{cases}$$, for all $i = 1,\ldots,p$.

This partition defines the following structure of the variance-covariance matrix:

$$C = \begin{bmatrix} C_Z & C_{ZY} \\ C_{YZ} & C_Y \end{bmatrix}.$$

Partial variance-covariance is calculated as: $P = C_Y - C_{YZ}C_Z^{-1}C_{ZY}$.

The example below demonstrates computation of a partial variance-covariance matrix:

```
#include "mkl.h"

#define N          1000     /* number of observations */
#define DIM           4     /* dimension of the task */
#define PART_DIM (DIM/2)    /* dimension of partial variance-covariance */

int main()
{
    int i, j, status;
    VSLSSTaskPtr task;
    MKL_INT p, n, xstorage, covstorage, pcovstorage;
    double x[DIM][N];  /* matrix of observations */
    unsigned long long estimates;

    double mean[DIM], cov[DIM][DIM];
    MKL_INT p_index[DIM];
    double p_cov[PART_DIM][PART_DIM];

    p = DIM;
    n = N;
    xstorage    = VSL_SS_MATRIX_STORAGE_ROWS;
    covstorage  = VSL_SS_MATRIX_STORAGE_FULL;
    pcovstorage = VSL_SS_MATRIX_STORAGE_FULL;

    /* Splitting random vector into two components */
    for(i=0;i<DIM;i++)
    {
        p_index[i]=(i<PART_DIM)? 1 : -1;
        mean[i] = 0.0;
        for(j=0;j<DIM;j++) cov[i][j]=0;
    }

    for(i=0;i<PART_DIM;i++)
    {
        for(j=0;j<PART_DIM;j++) p_cov[i][j]=0;
    }

    /* Create a task */
    status = vsldSSNewTask( &task, &p, &n, &xstorage, x, 0, 0 );

    /* Initialize the task parameters */
    status = vsldSSEditCovCor( task, mean, cov, &covstorage, 0, 0 );
    status = vsldSSEditPartialCovCor( task, p_index, cov, &covstorage,
                                      0, 0, p_cov, &pcovstorage, 0, 0 );

    /* Compute the variance-covariance and partial variance-covariance matrices */
    estimates = VSL_SS_COV | VSL_SS_PARTIAL_COV;
    status = vsldSSCompute( task, estimates, VSL_SS_METHOD_FAST );

    /* Deallocate the task resources */
    status = vslSSDeleteTask( &task );

    return 0;
}
```

3.10. Performing Robust Estimation of a Variance-Covariance Matrix

Use the Translated Biweight S-estimator (TBS) method to perform robust estimation of a variance-covariance matrix and mean vector [Rocke96]. The start point of the algorithm is computed using a single iteration of the Maronna algorithm with the reweighting step [Marrona2002]. The parameters of the TBS algorithm are packed into the `params` array. A pointer to this array along with other required parameters is passed to the task descriptor using the `EditRobustCovariance` editor. The structure of the `params` array is available in Table Structure of the Array of TBS Parameters in the Summary Statistics section of [MKLMan].
The algorithm outputs a robust variance-covariance matrix and the mean vector. The following example illustrates computation of a robust estimation for the variance-covariance matrix with the help of the TBS estimator:

```
#include "mkl_vsl.h"

#define DIM   10   /* dimension of the task */
#define N   1000   /* number of observations */

int main()
{
    VSLSSTaskPtr task;
    double x[DIM][N];  /* matrix of observations */
    double params[VSL_SS_TBS_PARAMS_N];
    double rcov[DIM*(DIM+1)/2], rmean[DIM];
    MKL_INT nparams, xstorage, rcovstorage;
    MKL_INT p, n;
    int status;

    double breakdown, alpha, sigma, max_iter;

    /* Parameters of the task and initialization */
    p = DIM;
    n = N;
    xstorage    = VSL_SS_MATRIX_STORAGE_ROWS;
    rcovstorage = VSL_SS_MATRIX_STORAGE_U_PACKED;
    nparams     = VSL_SS_TBS_PARAMS_N; /* number of TBS parameters */

    /* Parameters of the TBS estimator */
    breakdown = 0.3;
    alpha = 0.01;
    sigma = 0.01;
    max_iter = 30;

    params[0] = breakdown;
    params[1] = alpha;
    params[2] = sigma;
    params[3] = max_iter;

    /* Create a task */
    status = vsldSSNewTask( &task, &p, &n, &xstorage, (double*)x, 0, 0 );
    /* Initialize the task parameters */
    status = vsldSSEditRobustCovariance( task, &rcovstorage,
                                         &nparams, params, rmean, rcov );

    /* Compute the robust variance-covariance matrix */
    status = vsldSSCompute( task, VSL_SS_ROBUST_COV, VSL_SS_METHOD_TBS );

    /* Deallocate the task resources */
    status = vslSSDeleteTask( &task );

    return 0;
}
```

To calculate a robust variance-covariance matrix, you need to get the invers variance-covariance matrix for computing the Mahalanobis distance. In some cases, the inverse matrix cannot be calculated, for example, if the random vector components are dependent. Summary Statistics TBS algorithm checks the reversibility of the matrix by calculating its eigenvalues. If the minimum eigenvalue is non-positive, the algorithm searches for the minimum positive matrix eigenvalue *E* that exceeds 1000**P*, where *P* is the minimal positive floating-point number. If the routine fails to find such eigenvalue, it terminates the computations and returns an error code. Otherwise, the variance-covariance matrix is corrected by adding 0.01**E* to the elements of the main diagonal, and the calculations continue. Upon successful completion, the function returns the `VSL_SS_NOT_FULL_RANK_MATRIX` warning, indicating that the algorithm has detected a variance-covariance matrix of an incomplete rank.

The `max_iter` parameter passed in the third position of the array of TBS parameters defines the maximal number of iterations the TBS algorithm can perform before terminating the calculations. If this parameter is set to zero, the function returns a robust estimate of the variance-covariance matrix computed by means of the Maronna method only.

Summary Statistics algorithms for computation of a robust variance-covariance matrix and an array of means do not support progressive processing of the datasets available in blocks.

3.11. Detecting Multivariate Outliers

Use the BACON algorithm to detect multivariate outliers [Billor2000].

The parameters of the algorithm are packed into the `BaconParams` array. Use the `EditOutliersDetection` editor to pass into the library the pointer to this array and other required parameters. The Structure of the Array of BACON Parameters table in the Summary Statistics section of [MKLMan] describes the structure of the `BaconParams` array.

The BACON algorithm outputs a vector of weights `BaconWeights` that can take the following values:

1. If the *i*-th observation is detected as an outlier, `BaconWeights`(*i*) = 0.
2. If the vector of input weights is not provided and the *i*-th observation is not detected as an outlier, `BaconWeights`(*i*) = 1.
3. In all other cases, `BaconWeights`(*i*) = *w*(i), where *w* is the vector of input weights.

The example below illustrates the outlier detection using the BACON algorithm:

```
#include "mkl_vsl.h"

#define DIM 10      /* dimension of the task */
#define N   1000    /* number of observations */
#define M      3    /* number of BACON algorithm parameters */

int main()
{
    VSLSSTaskPtr task;
    double x[DIM][N];   /* matrix of observations */
    double BaconParams[VSL_SS_BACON_PARAMS_N];
    double BaconWeights[N];
    MKL_INT p, n, xstorage;
    MKL_INT NParams;
    int status;
    double init_method, alpha, beta;

    /* Task and Initialization Parameters */
    p = DIM;
    n = N;
    xstorage = VSL_SS_MATRIX_STORAGE_ROWS;

    /* Parameters of the BACON algorithm */
    init_method = VSL_SS_METHOD_BACON_MEDIAN_INIT;
    alpha  = 0.01;
    beta   = 0.01;
    NParams = VSL_SS_BACON_PARAMS_N;

    BaconParams[0] = init_method;
    BaconParams[1] = alpha;
    BaconParams[2] = beta;

    /* Create a task */
    status = vsldSSNewTask( &task, &p, &n, &xstorage, (double*)x, 0, 0 );

    /* Initialize the task parameters */
    status = vsldSSEditOutliersDetection( task, &NParams, BaconParams,
                                          BaconWeights );

    /* Detect the outliers in the observations */
    status = vsldSSCompute( task, VSL_SS_OUTLIERS, VSL_SS_METHOD_BACON );

    /* BaconWeights will hold zeros or/and ones */

    /* Deallocate the task resources */
    status = vslSSDeleteTask( &task );

    return 0;
}
```

NOTE

Outlier detection is only possible in data arrays available at once, or in separate blocks of the datasets.

Calculation of the Mahalanobis distance used in the BACON algorithm requires computation of an inverse variance-covariance matrix. In some cases, the inverse matrix cannot be calculated, for example, if components of the random vector are dependent. The Intel® MKL version of the BACON algorithm checks the reversibility of the matrix by calculating its eigenvalues. If the minimum eigenvalue is non-positive, the algorithm searches for the minimal matrix eigenvalue *E* exceeding 1000**P*, where *P* is the minimal positive floating-point number. If the routine fails to find such an eigenvalue, the computations terminate with a corresponding error code. Otherwise, the variance-covariance matrix is corrected by adding 0.01**E* to elements of the main diagonal, and the calculations continue. Upon successful completion, the function returns the `VSL_SS_NOT_FULL_RANK_MATRIX` warning, indicating that the algorithm has detected a variance-covariance matrix of an incomplete rank.

3.12. Handling Missing Values in Matrices of Observations

3.12.1. Handling Missing Values in Matrices of Observations

Summary Statistics provides the Multiple Imputation (MI) method `VSL_SS_METHOD_MI` to deal with missing values in a dataset. A typical usage flow is as follows:

1. In the MI paradigm, replace each missing value with a set of *m* values predicted from the underlying distribution.
2. After MI application, analyze each of the *m* complete datasets producing estimates and standard errors.
3. Combine the results of the first two steps according to the rules in [Rubin1987] to produce overall estimates and standard errors.

MI approach is integrated into the library as described in [Schafer1997].

See Also

Basic Assumptions under the MI Method
Basic Components of the MI Method

3.12.2. Basic Assumptions for the MI Method

The MI method is provided under the following assumptions:

1. The base model for the Summary Statistics version of MI is a multivariate normal distribution with parameters (μ, Σ)
   where
   a. μ is a vector of means.
   b. Σ is a variance-covariance matrix.
2. Prior distribution of μ is a conditionally-multivariate Gaussian given Σ with parameters $\mu_0 \in R^7$ and $\tau^{-1}\Sigma$, where τ is a positive constant. The variance-covariance matrix Σ follows the inverted-Wishart distribution for fixed parameters $m \geq p$ and a positive-definite matrix Λ.
3. Data points are Missed At Random (MAR).

The strict definition of this and other mechanisms supporting missing values are available in [Rubin1987].

3.12.3. Basic Components of the MI Method

Summary Statistics MI method comprises two components:

1. Expectation Maximization (EM) algorithm that computes the start point for the Data Augmentation (DA) algorithm
2. Simulation-based DA function that uses Intel® MKL random number generators

The parameters of the MI method are packed into the `params` array. See Table Structure of the Array of MI Parameters in the Summary Statistics section of [MKLMan] for the description of the parameters and their location in the array.

Beside the array of MI parameters, you can pass initial estimates of the mean and variance-covariance matrix (start point) into the EM algorithm. The array of means and variance-covariance matrix are packed as a one-dimensional array `init_estimates`. The size of the array should be at least $p+p(p+1)/2$. The package format is as follows:

1. For `i=0,..,p-1`, `init_estimates[i]` contains the start estimate of means.
2. The remaining positions of the array store the upper-triangular part of the variance-covariance matrix.

If you do not provide the start point for the EM algorithm, it uses the default start point, which is the vector of zero means, and the unitary matrix as a variance-covariance matrix.

You can pass into the library prior parameters for μ and Σ: μ_0, τ, m, and Λ^{-1}. As the DA function uses inverted matrix Λ, the MI algorithm expects the inverse of Λ. These parameters are packed as a one-dimensional array `prior`. The size of the array should be at least $(p^2 + 3p +4)/2$ to hold all the parameters. The storage format is as follows:

1. `prior[0],..., prior[p-1]` contain elements of vector μ_0.
2. `prior[p]` contains parameter τ.
3. `prior[p+1]` contains parameter m.
4. The remaining positions contain the upper-triangular part of the inverted matrix Λ^{-1}.

If the prior parameters are not provided, the algorithm uses their default values:

1. μ_0 is set to an array of p zeros.
2. τ is set to 0.
3. m is set to p.
4. for the initial approximate of Λ^{-1}, the zero matrix is used.

Proper processing of the default parameter values ensures correct computation.

The MI algorithm returns m sets of imputed values and/or a sequence of parameter estimates drawn during the DA procedure. The imputed values are returned as a single array `simul_missing_vals`. The size of the array should be sufficient to hold m sets, each of size `missing_values_num`, that is, m * `missing_values_num` in total. Imputed values are packed one by one in the order of their appearance in the matrix of observations.

Example:

Consider a task of dimension 4 with the total number of observations n=10, where the second vector misses values for the first and the second variables, and the seventh observation misses the first point. The number of sets to impute is m=2. Thus, `simul_missing_vals[0]` and `simul_missing_vals[1]` contain the first and the second points for the second observation vector, `simul_missing_vals[2]` holds the first point for the seventh observation. Similarly, positions 3, 4, and 5 are reserved for the second set of simulated values.
To estimate convergence of the DA algorithm and choose a proper value for DA iterations, you may generate a sequence of parameter estimates produced during the DA procedure. Elements of the sequence can then be analyzed to estimate convergence of the algorithm. For example, see [Schafer1998].

The sequence of the parameters is returned as a single array. The size of the array should be
$m*da\_iter\_num*(p + (p^2 + p)/2)$

where

1. m is the number of sets of values to impute
2. `da_iter_num` is the number of DA iterations
3. $p + (p^2 + p)/2$ is the size of the memory to hold one set of the parameter estimates.

Each set of parameters is packed as follows:

1. The vector of means occupies the first *p* positions.
2. The upper-triangular part of the variance-covariance matrix occupies the remaining *$(p^2 + p)/2$* positions.

Before the call to the MI algorithm, the dataset to be analyzed should be pre-processed and all missing observations should be marked with a predefined parameter, as follows:

1. `VSL_SS_DNAN`, if the dataset is stored in double precision floating-point arithmetic
2. `VSL_SS_SNAN`, if the dataset is stored in single precision floating-point arithmetic

Upon successful generation of *m* sets of imputed values, you can place them in cells of the data matrix with missing values and use other Summary Statistics algorithms to analyze and compute estimates for each of the *m* complete datasets.

NOTE

Intel® MKL implementation of the MI algorithm rewrites cells of the dataset that are marked with the `VSL_SS_DNAN/VSL_SS_SNAN` values.

If the combination of `VSL_SS_MISSING_VALS` and any other estimate parameter are passed into the `Compute` routine, only the algorithm for processing missing values is called.
The example below shows a typical MI usage scenario:

```
#include "mkl_vsl.h"

#define DIM    3                     /* dimension of the task */
#define N   10000                    /* number of observations */
#define M    VSL_SS_MI_PARAMS_SIZE  /* number of MI parameters */
#define M_VALUE    9                 /* total number of missing values */
#define M_COPIES   5                 /* number of sets of imputed values */
int main()
{
int status;
VSLSSTaskPtr task;
MKL_INT i;
double x[DIM * N]; /* matrix of observations */
double W[2];
double mean[DIM], r2m[DIM], c2m[DIM];
MKL_INT p, n, xstorage;
double em_iter_num, da_iter_num, em_accuracy, copy_num, missing_value_num;
double params[M], simul_missing_vals[M_VALUE * M_COPIES];
MKL_INT nparams = M, simul_missing_val_n = M_VALUE * M_COPIES;

 /* Pre-process the dataset and mark entries of missing values with VSL_SS_DNAN */
...

/* Parameters of the task and initialization */
p = DIM;
n = N;
xstorage = VSL_SS_MATRIX_STORAGE_ROWS;

/* Parameters of the MI algorithm */
em_iter_num       = 100;
da_iter_num       = 10;
em_accuracy       = 0.001;
copy_num          = M_COPIES;
missing_value_num = M_VALUE;
params[0] = em_iter_num;
params[1] = da_iter_num;
params[2] = em_accuracy;
params[3] = copy_num;
params[4] = missing_value_num;

/* Create a task */
status = vsldSSNewTask( &task, &p, &n, &xstorage, x, 0, 0 );

/* Initialize the task parameters */
status = vsldSSEditMissingValues( task, &nparams, params, 0, 0,
                                  0, 0, &simul_missing_val_n,
                                  simul_missing_vals, 0, 0 );

/* Generate m_values copies of missing value sets */
status = vsldSSCompute(task, VSL_SS_MISSING_VALS, VSL_SS_METHOD_MI );

/* Use the task to analyze the complete datasets */
W[0] = 0.0; W[1] = 0.0;
for ( i = 0; i < p; i++ )
{
mean[i] = 0.0; r2m[i] = 0; c2m[i] = 0.0;
}
status = vsldSSEditTask( task, VSL_SS_ED_ACCUM_WEIGHT, W );
status = vsldSSEditMoments( task, mean, r2m, 0,0, c2m, 0, 0 );
for ( i = 0; i < M_COPIES; i++ )

{
/* Perform imputation of  the next set of simulated values into x */
...

/* Compute the mean and the variance using the fast method */
errcode = vsldSSCompute(task, VSL_SS_MEAN|VSL_SS_2C_MOM,
                              VSL_SS_METHOD_FAST );

/* Analyze the computed estimates */
...
}

/* Deallocate the task resources */
status = vslSSDeleteTask( &task );

return 0;
}
```

3.13. Parameterizing a Correlation Matrix

Use a Spectral Decomposition method to parameterize a correlation matrix [Rebonato1999]. The input of the algorithm is a matrix that resembles a correlation matrix but lacks the property of positive semi-definiteness. The output of the algorithm is a parameterized correlation matrix with non-negative eigenvalues.

Summary Statistics supports two usage models for parametrizing a correlation matrix:

1. The dataset and its parameters are provided into the library through a constructor or editors of the Summary Statistics. The correlation matrix for this dataset is computed using the approaches described above. Parameterization of the correlation is performed with the method of Spectral Decomposition.

2. The correlation matrix computed earlier using Summary Statistics algorithms or other tools is simply registered in the Summary Statistics task. The dimension of the task defines the order of the matrix. In this case, you do not need to provide the dataset and its attributes such as number of observations and its storage format to the library.

The example below illustrates the second parameterization model:

```
#include "mkl_vsl.h"
#define DIM 3     /* dimension of the task */

int main()
{
    VSLSSTaskPtr task;
    MKL_INT p;
    MKL_INT corstorage, pcorstorage;
    int status;
    float cor[DIM*DIM], pcor[DIM*DIM];

    p = DIM;
    corstorage  = VSL_SS_MATRIX_STORAGE_FULL;
    pcorstorage = VSL_SS_MATRIX_STORAGE_FULL;

    /* Create a task */
    status = vslsSSNewTask( &task, &p, 0, 0, 0, 0, 0 );

    /* Register arrays for parameterization of the correlation matrix */
    status = vslsSSEditCorParameterization( task, cor, &corstorage,
                                            pcor, &pcorstorage );
    /* Compute a task */
    status = vslsSSCompute( task, VSL_SS_PARAMTR_COR, VSL_SS_METHOD_SD );

    /* Delete the task */
    status = vslSSDeleteTask( &task );

    return 0;
}
```

3.14. Sorting an Observation Matrix

You can use Summary Statistics routines to sort observation matrices by components of a random vector in the ascending order. The routines sort either rows or columns of the observation matrix depending on its storage, row- or column-major format. Sorting results are packed by the routines in accordance with your settings of the storage format for the output matrix. For supported storage formats see the table *Storage format of matrix of observations, order statistics and matrix of sorted observations* in the Summary Statistics section of [MKLMan].

You should allocate enough memory to hold the results of the calculations. The size of the array should provide storage for *p* x *n* elements, where:

- *p* is the dimension of the task
- *n* is the number of observations

Register the memory and storage format of the sorting result using the `vs[d|s]EditTask` editor before sorting the matrix of the observations. If you want the library to store sorting results into the input matrix, use

the storage type for matrix of the sorted observations identical to the one of the input dataset. In this case you cannot choose individual components of the random vector for sorting.
The library offers the radix sort method to sort the data. The sorting follows the pattern of the example below:

```
#define DIM 3      /* dimension of the task */
#define N  10     /* number of observations */
int main()
{
    VSLSSTaskPtr task;
    MKL_INT p, n;
    MKL_INT x_storage, sorted_x_storage;
    MKL_INT indices[DIM]={1,1,0}; /* the first two vector components are processed
*/
    double x[DIM][N];            /* matrix of observations */
    double y[DIM][N];            /* sorted matrix of observations */
    int status;
/* Parameters of the task and initialization */
    n = N;
    p = DIM;
    x_storage        = VSL_SS_MATRIX_STORAGE_ROWS;
    sorted_x_storage = VSL_SS_MATRIX_STORAGE_ROWS;
    /* Create a task */
    status = vsldSSNewTask(&task, &p, &n, &x_storage, (double *)x, 0, indices);
    /* Initialize the task parameters*/
    status = vsldSSEditTask(task, VSL_SS_ED_SORTED_OBSERV, y);
    status = vsliSSEditTask(task, VSL_SS_ED_SORTED_OBSERV_STORAGE,
&sorted_x_storage);
    /* Sort the observation matrix using the radix method **/
    status = vsldSSCompute(task, VSL_SS_SORTED_OBSERV, VSL_SS_METHOD_RADIX);
/* Deallocate the task resources */
    status = vslSSDeleteTask(&task);
    return 0;
}
```

4. Common Usage Model of Summary Statistics Algorithms

Any typical application that uses Summary Statistics passes four stages:

1. Creating a task
2. Modifying the task parameters
3. Computing statistical estimates
4. Destroying the task

Example:

To compute the mean, variance-covariance, and variation coefficient, you need to do the following:

1. Create a new task and pass into the library the parameters of the problem, dimension *p*, the number of observations *n*, and a pointer to the memory location where the dataset *X* is stored:

   ```
   xstorage = VSL_SS_MATRIX_STORAGE_COLS;
   errcode = vsldSSNewTask( &task, &p, &n, &xstorage, X, weights, indices );
   ```

   where

   a. The `weights` array contains the weights assigned to each observation.
   b. The `indices` array determines components of the random vector to be analyzed. Set the weights of the component to zero to exclude its observation from the analysis. For example, `indices` can be initialized as follows:
      ```
      indices[p] = {0, 1, 1, 0, 1,..};
      ```

   You can store the dataset in column-major or in row-major order. Use the `xstorage` variable to pass the storage format into the library. If you need to set all weights to 1 and process all components of the random vector, pass the `NULL` pointers instead of `weights` and `indices`.

2. Register arrays to hold computation results and other parameters. Use the editors available in the Summary Statistics domain. The example below illustrates how to use some of them:

   ```
   errcode = vsldSSEditTask( task, VSL_SS_ED_ACCUM_WEIGHT, W );
   errcode = vsldSSEditTask( task, VSL_SS_ED_VARIATION, Variation );
   errcode = vsldSSEditMoments( task, Xmean, Raw2Mom, 0, 0, Central2Mom, 0, 0 );
   ```

```
covstorage = VSL_SS_MATRIX_STORAGE_FULL;
errcode = vsldSSEditCovCor( task, Xmean, Cov, &covstorage, 0, 0 );
```

The arrays `Xmean`, `Raw2Mom`, `Central2Mom`, `Cov`, and `Variation` store estimates for the mean, the second algebraic moment, variance-covariance, and the variation coefficient, respectively. You need to specify the storage format for the variance-covariance matrix. You can choose between full and packed formats. Registration of an array of means is required in most cases even if you do not need this estimate. This is necessary as many other statistical estimates use the mean value. For more details, please see the Estimation of Raw and Central Moments and Sums, Skewness, Kurtosis, Variation, and Variance-Covariance/Correlation/Cross-Product Matrix chapter of this document and the Summary Statistics section of [MKLMan].

3. Compute the estimates of your interest by calling the computing routine that calculates them all at once:

```
estimates = VSL_SS_MEAN | VSL_SS_2C_MOM | VSL_SS_COV | VSL_SS_VARIATION;
errcode = vsldSSCompute( task, estimates, VSL_SS_METHOD_FAST );
```

The library only expects a pointer to the memory with the dataset. This permits placing another data to the same memory location and calling the `Compute` routine without re-editing the task descriptor.

4. Deallocate task resources:

```
errcode = vslSSDeleteTask( &task );
```

5. Processing Data in Blocks

Summary Statistics enables block-based data analysis that can help you:

1. compute statistical estimates for out-of-memory datasets, splitting them into blocks
2. analyze in-memory data arrays that become available block by block
3. tune your applications for out-of-memory data support

To compute statistical estimates for out-of-memory datasets, do the following:

1. Set the estimates of your interest to zero, or to any other meaningful value:

   ```
   for( i = 0; i < p; i++ )
   {
        Xmean[i] = 0.0;
        Raw2Mom[i] = 0.0;
        Central2Mom[i] = 0.0;
        for(j = 0; j < p; j++)
        {
            Cov[i][j] = 0.0;
        }
   }
   ```

2. Initialize array *W* of size 2 with zero values.

   This array holds accumulated weights that are important for correct computation of the estimates:

   ```
   W[0] = 0.0; W[1] = 0.0;
   ```

3. Get the first portion of the dataset into array *X*, and the corresponding weights into array `weights`:

   ```
   GetNextDataChunk( X, weights );
   ```

4. Follow the common usage model of the Summary Statistics algorithms:

   ```
   /* Create a task */
   xstorage = VSL_SS_MATRIX_STORAGE_COLS;
   errcode = vsldSSNewTask( &task, &p, &nblock,
                            &xstorage, X, weights, indices );

   /* Edit the task parameters */
   errcode = vsldSSEditTask( task, VSL_SS_ED_ACCUM_WEIGHT, W );
   errcode = vsldSSEditTask( task, VSL_SS_ED_VARIATION, Variation );
   ```

```
errcode = vsldSSEditMoments( task, Xmean, Raw2Mom, 0, 0, Central2Mom, 0, 0 );

covstorage = VSL_SS_MATRIX_STORAGE_FULL;
errcode = vsldSSEditCovCor( task, Xmean, cov, &covstorage, 0, 0 );

/* Compute the estimates for the dataset split into chunks */
estimates = VSL_SS_MEAN | VSL_SS_2C_MOM | VSL_SS_COV | VSL_SS_VARIATION;
for( nchunk = 0;  nchunk++; )
     errcode = vsldSSCompute( task, estimates, VSL_SS_1PASS_METHOD );
     If ( nchunk >= N ) break;
     GetNextDataChunk( X, weights );
}

/* Deallocate task resources */
errcode = vslSSDeleteTask( &task );
```

Summary statistics domain also enables reading the next data block into a different array. The whole computation scheme remains the same. You just need to provide the address of this data block to the library:

```
double* NextXChunk[N];
estimates = VSL_SS_MEAN | VSL_SS_2C_MOM | VSL_SS_COV | VSL_SS_VARIATION;
for( nchunk = 0; nchunk++; )
{
     errcode = vsldSSCompute( task, estimates, VSL_SS_1PASS_METHOD );
     If ( nchunk >= N ) break;
     GetNextDataChunk( NextXChunk, [nchunk], weights );
     errcode = vsldSSEditTask( task, VSL_SS_ED_OBSERV, NextXChunk,[nchunk] );
}
```

For the list of estimators that support processing datasets in blocks, see Table VS Summary Statistics Estimates Obtained with Compute Routine in the Summary Statistics section of [MKLMan].

6. Detecting Outliers in Datasets

6.1. Detecting Outliers in Datasets

Datasets may contain outliers or bad observations that do not belong to the distribution to be analyzed. The cause may be an unreliable process of data collection, as in the case of using micro-array technologies for measurement of gene expression levels, or intentional actions, such as network intrusion. Outliers can lead to biased estimates and wrong conclusions about the object.

To process datasets with outliers, you can choose between the BACON outlier detection algorithm and robust methods considered in the following sections.

6.2. Using the BACON Algorithm for Outlier Detection

The BACON algorithm is a tool for outlier detection that finds "suspicious" observations and provides unbiased statistical estimates for contaminated datasets.

Consider a dataset generated from a multivariate Gaussian distribution with the help of a corresponding generator available in Intel® Math Kernel Library (Intel® MKL). Some of the observations are then replaced with the outliers from the multivariate Gaussian distribution that has a significantly bigger mathematical expectation. The number of outliers is approximately 20%.

To detect the outliers using the BACON algorithm, do the following:

1. Initialize the algorithm parameters:

   a. Define the initialization scheme of the algorithm. You can choose between Median- and Mahalanobis distance-based schemes.

   b. Define rejection level as `alpha` and stopping criteria level as `beta`.

   The parameters are initialized as follows:

   ```
   init_method = VSL_SS_METHOD_BACON_MEDIAN_INIT;
   alpha = 0.05;
   beta  = 0.005;
   BaconN = VSL_SS_BACON_PARAMS_N;
   BaconParams[0] = init_method;
   ```

```
BaconParams[1] = alpha;
BaconParams[2] = beta;
```

For details on the parameters, see Table Structure of the Array of BACON Parameters of the Summary Statistics section in [MKLMan].

2. Pass the parameters into the library using a suitable editor:

```
errcode=vsldSSEditOutliersDetection( task, &BaconN, BaconParams, BaconWeights
);
```

The `BaconWeights` parameter is an array of weights that holds the output of the algorithm and points at suspicious observations. The size of the array equals the number of observations. The 0 value in the *i*-th position of the array indicates that the *i*-th observation requires special attention. The 1 value indicates that the observation is unbiased.

3. Call the `Compute` routine:

```
errcode = vsldSSCompute( task, VSL_SS_OUTLIERS, VSL_SS_METHOD_BACON );
```

When the computation completes, the `BaconWeights` array contains weights of the observations that have to be analyzed. You can use this array in further data processing. Register this array as an array of observation `weights` and use it in the usual manner. Expectedly, after all outliers are removed, the statistical estimates for the contaminated dataset are not biased.

See Also

Detection of Multivariate Outliers

6.3. Using Robust Methods

Robust methods of Summary Statistics provide two algorithms for outlier detection, Maronna, [Marrona2002] and TBS [Rocke96].

The Maronna algorithm computes the mean and variance-covariance matrix that serve as the start point for the TBS algorithm. The TBS algorithm permits iterating until the required accuracy is achieved or the maximal number of iterations completes. In addition to these parameters, you can specify and pass into the library the maximal breakdown point (the number of outliers the algorithm can hold) and an asymptotic rejection probability (ARP) [Rocke96].

To avoid iterations of the TBS algorithm and compute robust estimate of the mean and variance-covariance using the Maronna algorithm only, set the number of iterations to zero.

Consider a typical usage scenario for the robust methods editor and `Compute` routine provided below. Parameters of the algorithms, breakdown point, ARP, accuracy and the maximal number of TBS iterations are passed as an array:

```
breakdown_point = 0.2;
arp             = 0.001;
method_accuracy = 0.001;
iter_num        = 5;

params[0] = breakdown_point;
params[1] = arp;
params[2] = method_accuracy;
params[3] = iter_num;
```

Robust estimates are stored in memory as `rmean` and `rcov`. In the example below, the variance-covariance matrix is stored in the full format specified in the `rcov_storage` variable.

```
errcode =  vsldSSEditRobustCovariance( task, &rcov_storage,
                                       &nparams, params, rmean, rcov );
```

The `Compute` routine computes the estimates:

```
errcode=vsldSSCompute( task, VSL_SS_ROBUST_COV, VSL_SS_METHOD_TBS );
```

Example:

Consider a task with the dimension *p* = 10 and the number of observations *n* = 10,000. The dataset is generated from a multivariate Gaussian distribution with zero mean and a variance-covariance matrix holding 1 on the main diagonal and 0.05 in other entries. The dataset is contaminated with shift outliers that have a multivariate Gaussian distribution with the same variance-covariance matrix and a vector of means with all entries equal to 5.

Use of a non-robust algorithm for variance-covariance and mean estimation for this dataset results in biased estimates. Zero *p*-values for these estimates are returned.

```

Means:
0.2566,0.2583,0.2576,0.2633,0.2439,0.2556,0.2530,0.2716,0.2535,0.2519

Variance-Covariance:
2.2540
1.2715 2.1819
1.2852 1.2462 2.2046
1.2885 1.2684 1.2553 2.2310
1.2850 1.2581 1.2571 1.2526 2.2112
1.2650 1.2284 1.2419 1.2820 1.2430 2.1929
1.2789 1.2435 1.2550 1.2555 1.2574 1.2478 2.2113
1.2773 1.2692 1.2676 1.2751 1.2725 1.2733 1.2739 2.2448
1.2813 1.2579 1.2688 1.2723 1.2670 1.2713 1.2839 1.3061 2.2246
1.2696 1.2631 1.2515 1.2701 1.2597 1.2686 1.2554 1.2638 1.2780 2.1893
```

Use of the Maronna algorithm (that is, `iter_num = 0`) results in the following estimates:

```
Means:
-0.0022,0.0081,-0.0075,0.0049,-0.0054,0.0012,-0.0087,0.0194,-0.0073,0.0022

p-values for means:
0.1792 0.6077 0.5640 0.3869 0.4281 0.1014 0.6375 0.9570 0.5602 0.1846

Variance-Covariance:
0.9164
0.0605 0.8945
0.0617 0.0374 0.9269
0.0602 0.0570 0.0472 0.9294
0.0584 0.0469 0.0599 0.0443 0.9183
0.0552 0.0394 0.0395 0.0655 0.0484 0.9049
0.0487 0.0449 0.0471 0.0451 0.0564 0.0461 0.9186
0.0293 0.0555 0.0539 0.0456 0.0450 0.0574 0.0501 0.9149
0.0507 0.0339 0.0433 0.0504 0.0429 0.0603 0.0597 0.0696 0.8962
0.0375 0.0573 0.0470 0.0472 0.0502 0.0607 0.0420 0.0381 0.0484 0.8848

p-values for variance-covariance:
0.0000
0.2989 0.0000
0.2966 0.5842 0.0000
0.3471 0.4395 0.9592 0.0000
0.3994 0.9148 0.3590 0.8993 0.0000
0.5128 0.7023 0.6708 0.1869 0.8510 0.0000
0.8508 0.9752 0.9515 0.9411 0.4812 0.9714 0.0000
0.2669 0.4841 0.6001 0.9729 0.9530 0.4207 0.7751 0.0000
0.7151 0.4529 0.8765 0.7468 0.8689 0.2968 0.3317 0.0984 0.0000
0.6082 0.3734 0.9088 0.8997 0.7250 0.2720 0.8321 0.6358 0.7895 0.0000
```

These estimates are much better. However, the main diagonal of the matrix still gets a zero *p*-value. To improve the estimate, do five iterations of the TBS algorithm. Quick experiments show that further increase in the number of iterations does not change the estimates significantly:

```
Means:
-0.0018,0.0034,0.0026,0.0067,-0.0108,0.0012,-0.0024,0.0122,-0.0057,-0.0044

p-values for means:
0.1412 0.2612 0.2025 0.4860 0.7098 0.0943 0.1882 0.7693 0.4263 0.3381

Variance-Covariance:
1.0524
0.0583 1.0172
0.0757 0.0426 1.0403
0.0653 0.0630 0.0490 1.0538
0.0672 0.0604 0.0559 0.0462 1.0367
0.0493 0.0295 0.0434 0.0784 0.0442 1.0261
0.0620 0.0429 0.0509 0.0453 0.0491 0.0488 1.0397
0.0410 0.0503 0.0476 0.0507 0.0497 0.0514 0.0497 1.0367
0.0450 0.0370 0.0486 0.0464 0.0430 0.0526 0.0622 0.0719 1.0179
0.0477 0.0587 0.0461 0.0562 0.0514 0.0645 0.0443 0.0346 0.0485 1.0070

p-values for variance-covariance:
0.0002
0.6951 0.2249
0.1676 0.5972 0.0044
0.4613 0.5057 0.8450 0.0001
0.3761 0.5862 0.8152 0.7231 0.0095
0.8726 0.1942 0.6233 0.1170 0.6604 0.0646
0.5690 0.6118 0.9464 0.6795 0.8671 0.8653 0.0050
0.5092 0.9507 0.7992 0.9266 0.9002 0.9932 0.8944 0.0094
0.6867 0.4013 0.8656 0.7504 0.6147 0.9305 0.5185 0.2177 0.2065
0.8205 0.6243 0.7594 0.7800 0.9869 0.4071 0.6776 0.3207 0.8961 0.6185
```

For more details on robust methods, see Robust Estimation of Variance-Covariance Matrix chapter of this document and the Summary Statistics section of [MKLMan].

7. Dealing with Missing Observations

Real-life datasets can have missing values. For example, sociological surveys and measurement of complex biological systems have to deal with missing observations. Outliers in datasets can also be treated as lost samples. Intel® Math Kernel Library (Intel® MKL) provides the Expectation-Maximization and Data Augmentation (EMDA) method for accurate processing of datasets with missing points.

The EMDA method is based on the approach described in [Schafer1997], comprising the Expectation-Maximization (EM) and Data Augmentation (DA) algorithms. The EMDA method outputs *m* sets of simulated missing points that can be imputed into the dataset producing *m* complete data copies. For each dataset, you can compute a specific statistical estimate. The final estimate is a combination of such *m* estimates. For details on computational aspects and usage model of the algorithm, see Support of Missing Values in Matrices of Observations.

The parameters of the EMDA method are passed into the library as follows:

1. The EM algorithm iterates `em_iter_num` times to compute the initial estimate for the mean and variance-covariance used as the start point of the DA algorithm. The EM algorithm can terminate earlier if it achieves the given accuracy `em_accuracy`.
2. The DA algorithm iterates `da_iter_num` times. This algorithm uses Gaussian random numbers underneath. For this reason, EMDA algorithm uses `VSL_BRNG_MCG59` basic random number generator with the pre-defined *seed* = $2^{50}$and Gaussian distribution generator (ICDF method) available in Intel® MKL.

As the EMDA algorithm requires a number of missing values `missing_value_num`, you need to pre-process the dataset and mark all missing values using the `VSL_SS_DNAN` macro defined in the library. For a single-precision dataset, use the `VSL_SS_SNAN` macro. The algorithm parameters are passed into the library as the `params` array:

```
em_iter_num     = 10;
da_iter_num     = 5;
em_accuracy     = 0.001;
copy_num        = m;
miss_value_num  = miss_num;

params[0] = em_iter_num;
params[1] = da_iter_num;
params[2] = em_accuracy;
params[3] = copy_num;
params[4] = missing_value_num;
```

The editor for the EMDA method accepts the following set of parameters:

```
errcode = vsldSSEditMissingValues( task, &nparams, params, &init_estimates_n,
                                   init_estimates, &prior_n, prior,
                                   &simul_missing_vals_n, simul_missing_vals,
                                   &estimates_n, estimates );
```

The EM algorithm starts using the array of initial estimates `init_estimates`. The vector of means occupies the first *p* positions of the array. The upper-triangular part of the variance-covariance matrix occupies the rest *p*(p+1)/2* entries, where *p* is the dimension of the task. The `prior` array holds prior parameters for the EMDA algorithm.
The algorithm returns the sets of simulated missing points in the `simul_missing_vals` array. In total, *m**`missing_value_num` values are returned. Missing values are packed one by one, starting from the missing points for the first variable of the random vector.
To estimate convergence of the DA algorithm, pass the `estimates` array holding the mean/variance-covariance for all iterations and all sets of simulated missing points, `da_iter_num` $* (p + 0.5 * (p^2 + p))$ in total. In each set of the estimates, first *p* entries hold the mean, and the rest $0.5 * (p^2 + p)$ entries hold the upper-triangular part of the variance-covariance matrix.
For the description of parameters passed into the EMDA algorithm using an editor and the requirements for the size of the arrays, see Support of Missing Values in Matrices of Observations.

To start the EMDA algorithm, call the `Compute` routine:

```
errcode = vsldSSCompute( task, VSL_SS_MISSING_VALS, SL_SS_METHOD_MI );
```

Example:

Consider a task with the dimension *p = 10* and the number of observations *n = 10,000*. The dataset is generated from a multivariate Gaussian distribution with the zero mean and a variance-covariance matrix that holds 1 on the main diagonal and 0.05 in other entries. The ratio of missing values in the dataset is 10%. Each observation may have one missing point in any position. The goal is to generate *m=100* sets of lost observations. The start point for the EM algorithm is the vector of zero means and the identity variance-covariance matrix. The pointer to the `prior` array is set to 0. The size of this array `prior_n` is also 0.
The workflow is as follows:

1. A trial run of the algorithm with `da_iter_num` = 10 is performed.
   The analysis of the estimates in the `estimates` array shows that five iterations are sufficient for the DA algorithm.
2. 100 sets of missing values are simulated and imputed into the dataset, producing 100 complete data arrays.
3. For each complete dataset, means and variance are computed using Summary Statistics algorithms:

```
Set:            Mean:
    1        0.013687  0.005529  0.004011 ...  0.008066
    2        0.012054  0.003741  0.006907 ...  0.003721
    3        0.013236  0.008314  0.008033 ...  0.011987
  ...
   99        0.013350  0.012816  0.012942 ...  0.004076
  100        0.014677  0.011909  0.005399 ...  0.006457
_____________________________________________________

Average      0.012353  0.005676  0.007586 ...  0.006004

Set:         Variance:
    1        0.989609  0.993073  1.007031 ... 1.000655
    2        0.994033  0.986132  0.997705 ... 1.003134
    3        1.003835  0.991947  0.997933 ... 0.997069
   ...
    99       0.991922  0.988661  1.012045 ... 1.005406
   100       0.987327  0.989517  1.009951 ... 0.998941
```

```
________________________________________________

Average      0.99241   0.992136  1.007225 ... 1.000804

Between-imputation variance:
0.000007 0.000008 0.000008 ... 0.000007

Within-imputation variance:
0.000099 0.000099 0.000101 ... 0.000100

Total variance:
0.000106 0.000107 0.000108 ... 0.000108
```

For the vector of means, 95% confidence intervals are computed:

```
95% confidence interval:
Left boundary of interval:  -0.008234 -0.015020 -0.013233 ...  -0.014736
Right boundary of interval: +0.032939 +0.026372 +0.028406 ...  +0.026744
```

To test the output of the algorithm, the whole experiment is repeated 20 times. In all iterations, 95% confidence intervals contain the true value of mean.

8. Computing Quantiles for Streaming Data

Summary Statistics provides a method for computing quantiles that supports out-of-memory data.

In a nutshell, this method permits getting an <*e*-approximate quantile in one pass over the dataset, without knowing the total size of the dataset in advance. The <*e*-approximate quantile is an element in the dataset with the rank within the interval *[r- <e<n, r+<e<n]* for a user-provided error <e, size of dataset <*n* and any rank <*r=1,...,n.*

[Zhang2007] describes the theory and properties of this algorithm. For details on computational aspects, requirements, and usage model of the algorithm, see Using `VSL_SS_METHOD_SQUANTS_ZW` for Quantiles Computation chapter of this document.
Example:

Consider a simple application that uses this algorithm. The dimension of the task is set to <*p*=<*n*=10,000,000. The dataset is returned in blocks of 10,000 elements each. The goal is to compute deciles with a pre-defined error <e <= 0.00001, that is, the array elements with ranks deviating from the rank of accurate deciles by not more than 100 positions.

The library contains a special editor for this algorithm:

```
status = vsldSSEditStreamQuantiles ( task, &quant_order_n, quant_order,
                                     quant, &n_params, &params );
```

where

1. `quant_order_n` is the total number of quantiles, set to 9 in this example.
2. `quant_order` is an array initialized with quantile orders 0.1, 0.2,..., 0.9.
3. `&params` is an array of parameters. The only element in the array is the user-defined error <*e*, set to 0.00001 in this example.

The computation results are placed into the `quants` array.
To initialize the size of the array that contains parameters of the algorithm, you can use the macro defined in the library:

```
n_params = VSL_SS_SQUANTS_ZW_PARAMS_N
```

The loop for computing the deciles is as follows:

```
for ( block_index = 0; block_index < max_block_num; block_index++ )
{
    // Get the next data block of size block_n
    ...
    status = vsldSSCompute( task, VSL_SS_STREAM_QUANTS,
                                  VSL_SS_METHOD_SQUANTS_ZW );
    // Process computation results
    ...
}
```

Intermediate estimates of deciles are obtained immediately after processing the next block. As the dataset contains Gaussian numbers with the mean equal to 0 and the variance equal to 1, the sequence of the estimates is as follows:

```
Block   Streaming deciles:
index        D1      D2      D3      D4      D5     D6     D7     D8    D9
1        -1.2671 -0.8442 -0.5257 -0.2667 -0.0115 0.2524 0.5391 0.8496 1.2695
2        -1.2880 -0.8478 -0.5374 -0.2766 -0.0192 0.2400 0.5131 0.8327 1.2690
3        -1.2848 -0.8386 -0.5261 -0.2656 -0.0110 0.2428 0.5163 0.8366 1.2704
...
...
998      -1.2815 -0.8414 -0.5241 -0.2531  0.0009 0.2536 0.5248 0.8412 1.2814
999      -1.2815 -0.8414 -0.5241 -0.2531  0.0009 0.2536 0.5248 0.8413 1.2814
1000     -1.2815 -0.8414 -0.5241 -0.2531  0.0008 0.2536 0.5248 0.8412 1.2813
```

If you need to compute the estimate for the whole dataset only, you can use the fast version of the same method. It permits you to update the internal data structure in the library without actual computation of the intermediate estimates.

```

for ( block_index = 0; block_index < max_block_num; block_index++ )
{
    // Get the next data block of size block_n
    ...
    status = vsldSSCompute( task, VSL_SS_STREAM_QUANTS,
                              VSL_SS_METHOD_SQUANTS_ZW_FAST );
}
```

To get the estimate, set the `block_n` variable to zero, make sure it is registered in the library, and call the `Compute` routine:

```
block_n = 0;
status = vsldSSCompute( task, VSL_SS_STREAM_QUANTS,
                          VSL_SS_METHOD_SQUANTS_ZW );
```

The output of this application is identical to the last line of the previous table:

```
Streaming deciles:
   D1        D2       D3       D4      D5      D6      D7      D8     D9
-1.28154 -0.84141 -0.52418 -0.25312 0.00088 0.25367 0.52483 0.84129 1.28139
```

To check the estimates, the in-memory version of the quantile algorithm calculates accurate deciles for the same dataset. This algorithm returns the following output:

```
"Accurate" deciles:
   D1        D2       D3       D4      D5       D6      D7      D8     D9
 -1.28155 -0.84142 -0.52417 -0.25311 0.00089 0.25368 0.52484 0.84130 1.28140
```

The maximal difference between ranks of in-memory and out-of-memory deciles does not exceed 100, which fully aligns with the theory:

```
Rank difference
   D1        D2       D3       D4      D5      D6      D7      D8     D9
   4         5        3        1       3       7       8       7      4
```

9. Bibliography

[Billor2000] Nedret Billor, Ali S. Hadib, and Paul F. Velleman. BACON: blocked adaptive computationally efficient outlier nominators. Computational Statistics & Data Analysis, 34, 279-298, 2000.

[Maronna2002] R.A. Maronna and R.H. Zamar, Robust Multivariate Estimates for High-Dimensional Datasets. Technometrics, 44, 307-317, 2002.

[MKLMan] Intel® MKL Reference Manual.

[Rebonato1999] R. Rebonato and P Jackel, The most general methodology to create a valid correlation matrix for risk management and option pricing purposes, Quantitative Research Centre of the NatWest Group, October, 1999.

[Rocke96] David M. Rocke. Robustness properties of S-estimators of multivariate location and shape in high dimension. The Annals of Statistics, 24(3), 1327-1345, 1996

[Rubin1987] D.B. Rubin. Multiple Imputation for Nonresponse in surveys. J. Wiley & Sons, New York, 1987.

[Schafer1997] J.L. Schafer. Analysis of Incomplete Multivariate Data. Chapman & Hall, 1997.

[Schafer1998] J.L. Schafer and M.K. Olsen. Multiple Imputation for Missing-Data Problems: A Data Analyst's Perspective. Multivariate Behavioral Research, 33(4), 545-571, 1998.

[West79] D.H.D. West. Updating Mean and Variance Estimates: An improved method. Communications of ACM, 22(9), 532-535, 1979

[Zhang2007] Q. Zhang and W. Wang. A Fast Algorithm for Approximate Quantiles in High Speed Data Streams. SSDBM, p.29, 19th International Conference on Scientific and Statistical Database Management (SSDBM 2007), 2007.