Making the Monte Carlo Approach Even Easier and Faster

By Sergey A. Maidanov and Andrey Naraikin
Libraries of random-number generators for general probability distributions can make implementing Monte Carlo simulations easier and faster. The most important role of high-performance libraries is to provide facilities that make writing efficient programs easier and as natural as possible, while substantially increasing the speed of development.

Introduction

The name Monte Carlo was originally introduced by Nicholas Metropolis during the Manhattan Project, because of the similarity of statistical simulation to games of chance[17]. Methods that utilize random numbers to perform a simulation of phenomena have many applications in various disciplines. Monte Carlo became a tool to perform the most complex simulations in natural and social sciences, financial analysis, physics of turbulence, rarefied gas and fluid simulations, high-energy physics, chemical kinetics and combustion, radiation transport problems, and photorealistic rendering.

Monte Carlo methods are intended for various numerical problems such as solving ordinary stochastic differential equations, ordinary differential equations with random entries, boundary value problems for partial differential equations, integral equations, and evaluation of high-dimensional integrals including path-dependent integrals. Monte Carlo methods also include random variables and order statistics simulation, stochastic processes, and random samplings and permutations (see, for instance, [12], [13], [14]).

On one hand, many Monte Carlo methods simulate a phenomenon directly: a system behavior can often be described in terms of probability functions rather than in terms of differential equations, for instance. The idea of Monte Carlo is to simulate random variables from the distributions describing a phenomenon and to calculate statistical estimation of the quantity of interest. On the other hand, a convergence of the standard Monte Carlo method is of order $n^{-1/2}$, where $n$ is a number of trials used to estimate that quantity. Thus, the Monte Carlo method is computationally expensive and good for solving very complex problems only when classical numerical methods are too slow or not applicable at all.

This article discusses the role of random-number generators in Monte Carlo simulations, guiding criteria for selection of appropriate random-number generators and requirements for a modern numerical library with random-number generation capabilities. It also provides a brief overview of three industrial library solutions with random-number generation capabilities and a discussion of how these requirements are implemented or not implemented there. Finally, this document discusses three examples of applications from different areas where Monte Carlo methods are widely used, focusing on various techniques for implementing efficient Monte Carlo simulation.

Core Considerations for Random-Number Generator Implementations

At the core of any Monte Carlo method is a source to simulate random outcomes, a random-number generator. For various reasons, generators based on completely deterministic algorithms have wide use in simulations [12]. Such deterministic algorithms are called pseudorandom-number generators, since they only imitate the randomness. The term 'pseudorandom-number generator' generally means 'uniform distribution generator,' that is, a generator that imitates i.i.d. (independent identically distributed) sequences uniformly distributed over some interval. In
practice, however, many other univariate and multivariate, continuous and discrete distribution
generators are used in various applications.

The uniform distribution plays a key role in generating random sequences of other distributions,
because other distributions are essentially obtained using different transformation techniques
applied to the i.i.d. random numbers of the uniform distribution. Thus, the properties of non-
uniform random-number generator depend not only on the transformation method but also, to a
large extent, on the properties of the underlying uniform random-number generator. We call
random-number generators of the uniform distribution basic random-number generators, or
BRNGs.

Although there are some classes of applications where many pseudorandom-number generators
are inapplicable (problems dealing with very high dimensions) or simply dangerous
(cryptography), they are still suitable for most practical situations (see [15] and references
therein). Due to use of different deterministic algorithms, properties of generators might vary a
great deal. Thus, there are a number of aspects that should be considered when dealing with
random-number generators in order to select truly suitable ones for simulation purposes. We
outline below some of these aspects.

First, determine whether independence of random numbers is really required. Some Monte Carlo
methods, such as Monte Carlo integration, do not really require independence between random
numbers. For such methods, super-uniformity, super-normality, and so on are of interest.
Although pseudorandom generators are applicable here, there are also specially designed
generators that provide more evenly distributed sequences to the detriment of independence and
which might lead to more efficient implementation of the method [25]. These are called quasi-
random-number generators, and sequences they produce are called low-discrepancy sequences.
Low-discrepancy sequences with super-uniformity can be transformed into sequences with
distributions other than uniform ones.

Another important aspect is the period of a pseudorandom-number generator. All sequences
produced by pseudorandom-number generators repeat themselves starting from some moment. If
the selected generator has insufficient period length, this might cause misleading simulation
results. A good theoretically and empirically proved limit is not to use more random numbers
than the square root of the generator period.

A closely related question is the dimension of the random vectors to be used in the simulation.
The higher the dimension used, the larger the generator period should be. At the same time,
randomness of pseudorandom vectors rapidly degrades with a growth of dimension. Hence, the
higher the dimension, the more carefully the generator should be selected. Many articles discuss
related questions such as lattice structure, equidistribution, and discrepancy of the generators in
high dimensions (see [15] and references therein).

Multiprocessor hardware systems with shared and distributed memory are being used more and
more extensively in many computational areas. As indicated by data published at Top 500
Supercomputer Sites, [5] clusters built on standard and relatively inexpensive basic blocks are
gaining market share. High performance computing (HPC) is moving from its academic origins
into mainstream businesses. Even the desktop market is being affected by this trend as shared
memory systems start to appear there (e.g., in systems with Intel® Hyper-Threading Technology, providing thread-level-parallelism for multi-threading and multi-tasking operations).

Due to that trend and the computational cost of Monte Carlo methods, the question of their use in parallel environments is an important one. Many Monte Carlo methods allow quite natural parallelization. In this case, the choice of random number generators should be made according to their ability to work in parallel. One important aspect that should be considered is independence between sequences generated in parallel. There are a number of methods to generate independent sequences. We consider three of them below.

The first is simultaneous use of several random-number generators, parameters of which are chosen so that sequences produced by different generators are independent in some sense (for example, in terms of the spectral test [11]). The other two use splitting of the original pseudorandom sequence into \( k \) non-overlapping subsequences, where \( k \) is the number of threads/processes so that different threads/processes use random numbers from the corresponding subsequence only.

One of them is known as the block-splitting or skip-ahead method. In this case, the original sequence is being split into \( k \) non-overlapping blocks of subsequent elements, and each block maps to a corresponding subsequence. The other method is known as the leapfrog method. This differs from the skip-ahead method in that the leapfrog method splits the original sequence \( x_1, x_2, x_3, \ldots \) into \( k \) subsequences so that the first subsequence generates random numbers \( x_1, x_{k+1}, x_{2k+1}, x_{3k+1}, \ldots \), the second generates random numbers \( x_2, x_{k+2}, x_{2k+2}, x_{3k+2}, \ldots \), and, finally, the \( k^{th} \) generates random numbers \( x_k, x_{2k}, x_{3k}, \ldots \).

There are advantages and disadvantages to these approaches in various parallel Monte Carlo methods. Using the first method, the maximal number of independent streams is limited by the number of suitable parameters chosen. With the skip-ahead method, a high correlation between blocks is possible, although the randomness properties of subsequences themselves are the same as with the original sequence.

When the leapfrog method is used, randomness properties of subsequences degrade dramatically when the number of subsequences increases. Finally, for some generators, the implementation of the latter two methods could be as inefficient as generation of the whole original sequence to pick out a required subsequence. Hence, the most appropriate generators should be selected while taking into account all these considerations.

Performance is also an important aspect. Monte Carlo methods usually are computationally expensive and require a lot of random numbers to be generated to achieve suitable accuracy of the estimation. One way to improve simulation performance is to increase the speed of standard Monte Carlo convergence from \( n^{-1/2} \). There are many variance reduction techniques to do this ([12], [13], [14]) such as antithetic variates and low discrepancy sequences, as well as importance and stratified sampling.

For example, use of low-discrepancy sequences (if applicable) might substantially improve the speed of convergence compared with a standard Monte Carlo method using pseudorandom sequences [25]. Another way to improve performance is parallelization, as discussed above. In
any case, the speed of random-number generation remains very important. All finer randomness properties might be useless if the random-number generator used is too slow to provide results in a reasonable time.

The more complex the probability distribution, the more time is required to provide a sufficient quantity of random numbers. The fraction of time being spent in random-number generators varies in different applications and might be very significant in some of them. As illustrated by examples given later in this article, highly optimized generators allow substantial improvements of the overall speed of the application.

**Using Numerical Libraries in Random-Number Generators**

We outlined above a set of possible issues people might encounter when dealing with Monte Carlo methods in general and random-number generators in particular. They illustrate quite well the idea that there are a number of criteria, sometimes contradicting one another, in selecting random-number generators suitable for a particular application. Such selection might become a non-trivial task. Besides the work on the Monte Carlo algorithm itself, people often create software themselves or use public domain software to implement random-number generators as well.

When the problem specifics are known, selecting appropriate random-number generation algorithms is simplified. The wider the class of problems, however, the more aspects should be taken into account and wider functionality should be implemented. This leads to increasing the time spent on the implementation of such auxiliary functionality rather than time for work on the original problem of interest.

Another approach is to use general-purpose numerical software libraries, which provide a set of structural elements programmers may use to construct more complex structural elements. The computing environment and support services might provide substantial savings in the time and cost to develop numerical algorithms, as well as helping to improve the efficiency of those algorithms in terms of performance. While we give an overview of offerings in the area of statistical computing for three general purpose numerical libraries later in this article, we will now outline some requirements for random-number generation capabilities which are important for enabling of their use in a wide class of applications.

First, the library with random-number generator capabilities should provide a set of basic random-number generators with different properties (pseudo- and quasi-random, varying in period length, speed, statistical properties, applicability for sequence splitting techniques, etc.) permitting users to select the most appropriate ones for their purposes. On top of that set, there should be functionality for generating random numbers from commonly used univariate and multivariate, continuous, and discrete distributions. Since a distribution generator can be implemented using several transformation methods, and again as a rule there is no one superior method, the library should support different transformation methods for a given distribution.

Appropriate service functionality should be implemented to enable convenient and efficient use of structural elements. This includes mechanisms that allow for the use of different transformation methods with different basic generators, supporting simultaneous work with
several sequences based on the same or different basic generators. The latter includes efficient implementation of sequence-splitting techniques in single-processor and parallel environments. One useful feature is a mechanism for registering in the library one or several user-designed basic generators if none of the library basic generators serves well for the user’s purposes. In this case, service functionality should also enable convenient use of user-designed basic generators with the rest of the library.

On the user-interface side, the library should be programming-language-oriented, (i.e., it should support natural use in different programming languages. There should also be support for different integer and floating-point data types and their arbitrary mixture in the user’s program, since different levels of precision might be required in the same application. The library should have a consistent API to simplify integration in the user’s application. Unfortunately, there is no commonly recognized standard for random-number generators or their interface, as there is, for example, for the BLAS [6]. This is a serious barrier for the adoption for use of a numerical library due to portability issues, among other things. But at least within the single library, the API should be consistent and convenient.

To improve program efficiency in terms of performance, the library should effectively utilize processor and system architectural features. Toward that end, this is an issue similar to that of the user interface. Specifically, the majority of modern computer architectures allow substantial speedup for vector operations vs. scalar operations. (In contrast to the scalar-type routine producing just one resulting value, vector-type routines produce an array of resulting values.)

The following figure shows a typical performance tradeoff for vector vs. scalar routines, when routines of both types are highly optimized. Two scenarios are illustrated: one call to a vector routine to produce an array of results vs. sequential calls to a scalar routine to produce the same array. We show the dependency of time needed for producing each result as a function of the vector length (time divided by the number of elements produced). The units used are processor Clocks required per generated Element, or CPE.
This is a typical picture for vector operations in general and is not specific to random-number generators. In many cases, vector-type routines provide significant speedup for arrays longer than a few elements.

Use of vector-type random-number generators is quite natural for Monte Carlo applications, since they require many random numbers. On the other hand, a vector API can be rather inconvenient for some Monte Carlo algorithms, despite the fact that a large quantity of random numbers is needed. Hence, the library with vector random-number generators should provide mechanisms to make vector programming easier.

**Monte Carlo Implementations with Multiple Generators**

The situation when several distribution generators are used in combination is typical in Monte Carlo methods. Consider the following flowchart fragment repeated in the loop:

```
| Generate random number \( x \) from the distribution \( F_X \) |
|-----------------|-----------------|
| Generate random number \( y \) from the distribution \( F_Y \) |
| DO WORK \((x, y, \ldots)\) |
```

The pseudo-code and picture below illustrate how this fragment could be implemented with a software library, which has one scalar BRNG and scalar implementations of distribution generators:
If the same library has vector generators, the above example could be implemented as follows:

```plaintext
FOR i FROM 1 TO N DO
  x := RngFDistrScalar();
  y := RngGDistrScalar();
  DO_WORK(x, y, ...);
END FOR
```

We assume here that each vector-type generator has two parameters: the first is a vector length (quantity of random numbers to generate) and the second is a buffer in which to store the generated numbers. Such an implementation might lead to better overall performance, but let us share two observations. The first is that the order of use of uniform random numbers coming from the underlying BRNG to distribution generators has been changed. In many cases, such a change is acceptable, but sometimes it might be undesirable.

The other observation is connected with performance when $N$ is large. In this case, improvements in the speed of random-number generation from using vector routines might not lead to an overall speedup, due to an increasing number of cache misses when values $x$ and $y$ are being read from buffers. One possible solution is to process data by smaller portions as shown in the figure below, namely by using blocking, a common optimization method for numerical codes:

```plaintext
FOR i FROM 1 TO N DO
  x := BufferX[i];
  y := BufferY[i];
  DO_WORK(x, y, ...);
END FOR
```

For simplicity we assumed that $N$ is a multiple of $\text{BLOCK\_SIZE}$. $\text{BLOCK\_SIZE}$ can be set to assure that the buffers all fit in cache and still assure good performance, since the vector generators exceed scalar generators for quite short vector lengths. In the examples below, we assume that
such modification is done (if necessary) and will not show it explicitly in order to avoid overloading the illustrations.

Further, if the library supports several basic generators, then the following implementation is possible.

One additional parameter appeared in calls to distribution-generator subroutines: some identifier specifying the underlying BRNG to use, although this is not the only way to support the use of different BRNGs. For example, a service function can be provided to switch between basic generators. Explicit specification is a good way to minimize switching overheads, however.

Using different basic generators makes calls to distribution generators independent in terms of data and control flow. In addition, the appropriate selection of BRNGs allows statistical independence as well. Thus, this approach enables support of one of the parallelization methods discussed earlier in this section.

Notice that passing the same BRNG to both $F$ and $G$ distribution generators leads to the same situation as in the previous example, where in fact we work with a single sequence coming from the basic generator. More flexible mechanisms might be implemented to support simultaneous work with different sequences generated by the same BRNG. For example, this could be implemented as shown in the figure below:

Here, $StreamX$ and $StreamY$ are implementations of the mechanism uniquely identifying the combination of BRNG and its state. The key point is that the distribution generator operates with a particular sequence, rather than directly with a basic generator. Generally speaking, the structure of information describing the state of different BRNGs may differ. To simplify library adoption for users, it is important to provide unified mechanisms for operating with random
streams. Toward that end, it makes sense to shift dealing with state description differences to service functionality, which in turn should be unified as much as possible.

Notice that we still have not demonstrated how vector generators can preserve the same order of use of uniform numbers coming from BRNG as it was in the very first example based on the scalar implementation. The leapfrog method is what we really need to do that. The figure below shows how this could be done exploiting the mechanism of random streams:

We assume here that the library provides some service called Leapfrog, which allows splitting the original sequence associated with Stream into two subsequences StreamX and StreamY by the leapfrog method. Parameters Offs1 and Offs2 specify the offset from the first element of the original sequence in terms of number of elements, while Strd1 and Strd2 specify a stride with which elements of original sequence are picked.

The Leapfrog service adjusts state parameters so that a further target subsequence is generated without any more connection with the original sequence associated with Stream. As mentioned above, not all BRNGs allow effective implementation of such adjustment, due to features of underlying generation algorithms.

Regardless of whether one or several BRNGs are used, the leapfrog method provides a means of supporting program parallelization in addition to the possibility of using several sequences, each of which is based on different BRNGs. Finally, using a mechanism of random streams, the skip-ahead service mentioned earlier could be implemented similar to the leapfrog method. This is one more way of supporting effective use of random-number generators in parallel environments. We will illustrate the use of both leapfrog and skip-ahead methods in the examples discussed later in this article.

The examples above are simple. Functionality and mechanisms supporting convenient and efficient use of random-number generators in wide classes of Monte Carlo applications can be implemented in a number of different ways, which might differ from the ideas we have discussed. On the other hand, we believe that the ideas behind these examples reflect well the historical trend in development of numerical libraries over the years in order to address growing user demands in functionality to support Monte Carlo simulations. The following sections provide an overview of random-number generation capabilities offered in three general-purpose numerical libraries.
The commercially available software libraries discussed below provide a wide range of functionality for mathematical and statistical applications beyond just random-number generators. They provide functionality in such numerical areas as linear algebra, Fourier transformations, transcendental math functions, and many others. In this overview, we focus on statistical functionality and specifically on random-number generation capabilities.

Intel® Math Kernel Library Random-Number Generators

Random-number generation capabilities recently appeared in the Intel® Math Kernel Library (Intel® MKL) 6.0 as the Vector Statistical Library component, or VSL. While version 6.0 contains only random-number generators today, VSL assumes functionality extensions, which help the user to achieve the best results (in terms of the library flexibility, reliability, accuracy and performance on Intel® architectures) in various areas statistical methods are used. All VSL random-number generators are highly optimized for the latest features and capabilities of the Intel® Pentium® 4 processor, the Intel® Xeon™ processor, and the Intel® Itanium® 2 processor. Intel MKL supports use of its functionality from various C and Fortran compilers working under Windows* and Linux* operating systems.

The following figure illustrates the structure of the VSL:

There are five basic generators implemented in Intel MKL 6.0. Four of them are multiplicative generator MCG31m1, generalized feedback shift register generator R250, multiple recursive generator MRG32k3a, and 59-bit multiplicative generator MCG59. The fifth is a Wichmann-Hill basic generator, which in turn is a set of 273 basic generators designed to produce statistically independent sequences. They all vary in speed and statistical properties. The smallest period length is in MCG31m1 generator, which is equal to \( \sim 2 \times 10^9 \), while R250 has the largest period, \( \sim 2 \times 10^{75} \).
As mentioned above, the period length is not the only criterion. For example, the R250 generator behaves inadequately for self-avoiding random-walk application, [20] while the rest of the VSL generators work well there. All VSL BRNGs except R250 support the skip-ahead method. In addition, MCG31m1, MCG59, and Wichmann-Hill basic generators support the leapfrog method. All Intel MKL 6.0 BRNGs are pseudorandom. BRNGs based on low-discrepancy sequences might be a possible area for future Intel MKL functionality extensions. VSL provides an option of registering one or more user-designed basic generators and use them with VSL subroutines in the same way as original VSL basic generators.

The random stream is the basic notion in VSL. The random-streams mechanism allows generating an arbitrary number of random sequences produced by one or more basic generators simultaneously. Wide service functionality with a unified user interface allows creating, copying and deleting random streams as well as their use for sequence-splitting techniques such as the leapfrog and skip-ahead methods. Since all these mechanisms are thread-safe, VSL random-number generators can be efficiently used in parallel programming on systems with shared memory. In addition, all these mechanisms can be efficiently harnessed to facilitate the use of VSL on systems with distributed memory.

There are 17 pseudorandom-number generators for widely used probability distributions, such as uniform, Gaussian, exponential, Poisson, etc. Non-uniform distribution random numbers are generated via different transformation techniques applied to the uniformly distributed sequences associated with random streams. VSL supports different transformation methods for the same distribution generator by passing a parameter with a method number in a call to a generator. For example, there are two methods for Gaussian (Box-Muller and Box-Muller2 methods) and Poisson (PTPE and POISNORM methods) distributions. All current VSL generators are for univariate distributions. Multivariate distribution generators are another possible area of functionality extensions.

VSL supports different integer and floating point data types and their arbitrary mixture in the user’s program. Specifically, discrete distribution generators return integer values; continuous distribution generators return floating-point numbers of both IEEE-754 single- and double-precision types. Since uniform distribution generators can be treated as both discrete and continuous, VSL provides interface for them to return integer and floating-point values.

All VSL generators are of the vector type. They are highly optimized to exploit the latest advantages of processors based on Intel architectures. The following table illustrates the performance of some VSL generators measured in processor Clocks required Per generated array Element, or CPE. The vector length is 1000 elements.

<table>
<thead>
<tr>
<th>Distribution Generator</th>
<th>Intel Xeon processor</th>
<th>Itanium 2 processor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MC G31</td>
<td>R250</td>
</tr>
<tr>
<td>vsRngUniform</td>
<td>11.52</td>
<td>8.98</td>
</tr>
<tr>
<td>(single-precision uniform distribution generator)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Measurements for the Intel Xeon processor were done on the following platform: a two-way Intel Xeon processor-based system, running at 2.2 GHz with 512 KB L2 cache and 512 MB memory with the Windows XP Professional operating system. Measurements for the Itanium 2 processor were done on the following platform: a two-way Intel Itanium 2 processor-based server platform, running at 1.3 GHz with 6 MB L3 cache and 2 GB of memory with Windows Server 2003, Enterprise Edition.

The throughput of random-number generation is quite comparable with latencies of basic floating-point arithmetical operations such as addition and multiplication. (The latency of the FMAC operation on the Itanium 2 processor is four cycles, while latencies of addition and multiplication operations on the Pentium 4 processor vary from 4 to 7 cycles depending on the type of operation and execution unit (i.e., X87 vs. SSE2).) Thus, quite efficient implementation of random-number generators along with service functionality to make vector programming easier can help to significantly improve the performance of Monte Carlo simulations on systems based on Intel architecture processors. We show some examples of such improvements below.

See the Intel MKL 6.0 documentation for more information [1].

**IMSL Library Random-Number Generators**

Visual Numerics* provides a collection of mathematical and statistical analysis subroutines for Fortran and C/C++ users, known as IMSL* F90 Library and IMSL® C Numerical Library,
respectively. There is also a Java* library JMSL*, which combines numerical-analysis functions with visualization capabilities. The functionality of these libraries may differ from each other. In our overview, we will mainly focus on IMSL F90 MP Library 5.0 functionality.

IMSL Libraries are targeted on a wide range of machine platforms including Intel architecture based platforms, Cray*, HP*, IBM* RS/6000, and Sun*.

There are seven basic pseudorandom-number generators in IMSL: one GFSR generator, three multiplicative congruential generators, and the same multiplicative generators with shuffling. The properties of multiplicative congruential generators are similar to the MCG31m1 generator in Intel MKL, while properties of GFSR generator are similar to the Intel MKL R250 generator. In addition, IMSL provides Faure low-discrepancy generator.

There is a service subroutine to select the basic generator, as well as to switch between basic generators if required. Hence, to use several BRNGs simultaneously explicit switching is required. Two other subroutines are used to initialize and retrieve the seed of the currently selected BRNG. Since the state of GFSR and shuffled generators is an array of values rather than just a seed value, there are additional service subroutines to save and restore such arrays. These subroutines are different for GFSR and shuffled generators.

In addition, the tables representing basic generator state for the shuffled generators are different for single and double precision. Thus, if precisions are mixed in a program, it is necessary to manage each precision separately for the shuffled generators. In addition, there is a distinct set of service subroutines to initialize and finalize the quasi-random-number generator. There are no service subroutines to manipulate the state of the quasi-random-number generator.

IMSL provides the skip-ahead method to control more than one pseudorandom sequence simultaneously: for three multiplicative generators without shuffling, there is a subroutine to skip 100000 random numbers in a sequence. Notice, however, that saving and restoring seed subroutines should be called to switch between split sequences.

IMSL has random-number generation capabilities for 21 univariate distributions, 3 multivariate distributions, and for general discrete, continuous and data based multivariate distributions (distribution descriptive information is passed by user). In addition, random orthogonal and correlation matrices, two-way tables, order statistics from normal and uniform distributions, random samples and permutations, as well as stochastic processes, are in the library. Several transformation methods are available for some distribution generators. In contrast to Intel MKL, however, there are different subroutines in terms of API for each method. Finally, IMSL quasi-random-number generators provide only uniform distributions.

Some generation routines have both scalar and a vector implementations, while others are vector or scalar only implementations. Both single- and double-precision floating-point implementations are provided for continuous distributions.

For further details, please refer to the Visual Numerics web site [3].
NAG Fortran Library Random-Number Generators

Numerical Algorithms Group provides a numerical library for C/C++ and Fortran users known as the NAG Fortran 77 Library, NAG Fortran 90 Library, and NAG C Library, as well as NAG SMP Library and NAG Parallel Library. NAG libraries are targeted for different platforms including Intel architecture-based platforms, Apple, Cray, HP, IBM and Sun. The functionality of these libraries differs from each other. Moreover, the functionality of particular library may vary depending on platform and operating system. We will focus mainly on NAG Fortran 77 Library (Mark 20) random-number generation functionality.

Two pseudorandom basic generators are available in the NAG library: multiplicative congruential (NAG calls ‘original’) and Wichmann-Hill BRNG. These BRNGs correspond to MCG59 and Wichmann-Hill basic generators in Intel MKL, respectively. In addition, three quasi-random basic generators are in NAG implementing Sobol, Niederreiter, and Faure low-discrepancy sequences.

There are two distinct sets of generation routines, representing different methods for communicating the data representing the current state of basic generator. In the first set, the data is stored and passed internally, while in the second set the data is held in parameters that are passed through the routine interfaces. Routines using the internal communication have the advantage of a simpler interface. However, they use some thread-unsafe constructs and cannot be safely used in multithreaded applications. At the same time, routines taking BRNG state as a parameter can be safely used in multithreaded applications.

By virtue of two different mechanisms of passing the BRNG state, NAG has two sets of service subroutines. For routines with internal communication there are services for BRNG selection (for switching between BRNGs), initialization of the selected BRNG as well as saving and restoring the state of the selected BRNG. By saving and restoring the BRNG state, several subsequences from one BRNG can be obtained. However, notice again that routines with internal communication are not thread-safe. Routines communicating through the interface have a single subroutine for BRNG selection and initialization. The state of the generator can be saved and restored manually by copying from a variable representing one generator state to another one. This is a thread-safe mechanism.

NAG does not support sequence splitting techniques such as the leapfrog and skip-ahead methods.

NAG provides pseudorandom-number generator capabilities for 24 univariate distributions and two multivariate distributions as well as for general distribution (distribution-descriptive information is passed by user). In addition, NAG provides subroutines to generate samples and permutations, time-series, orthogonal and correlation matrices, as well as random tables. Quasi-random-number generators can be used to generate low-discrepancy sequences of uniform, Gaussian, and lognormal distributions only. Available functionality differs for routines with internal communication and for routines with explicit BRNG state passing. Some distribution generators have both vector and scalar forms, while others have a vector or a scalar form only.
Depending on the library and platform, NAG libraries have either double-precision or single-precision floating-point implementations only.

For further details please refer to the NAG web site [4].

### Summary of Random-Number Generators in Numeric Libraries

The table below presents summary information on random-number generation capabilities that exist in the Intel MKL 6.0, IMSL F90 MP 4.0 and NAG Fortran 77 (Mark 20) libraries.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Intel MKL</th>
<th>IMSL</th>
<th>NAG</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BRNGs</strong></td>
<td>• Five pseudorandom.</td>
<td>• Seven pseudorandom.</td>
<td>• Two pseudorandom.</td>
</tr>
<tr>
<td></td>
<td>• User-designed BRNGs are supported.</td>
<td>• One quasi-random.</td>
<td>• Three quasi-random.</td>
</tr>
<tr>
<td></td>
<td>• User-designed BRNG replaces library BRNGs.</td>
<td>• User-designed BRNG replaces library BRNGs.</td>
<td>• User-designed BRNGs are not supported.</td>
</tr>
<tr>
<td></td>
<td>Many library services do not work with user-</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>designed BRNG.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>**Sequence</td>
<td>• Arbitrary number of random streams based</td>
<td>• Service subroutine for switching between</td>
<td>• Service subroutines for switching between</td>
</tr>
<tr>
<td>manipulation</td>
<td>on one or more BRNGs.</td>
<td>BRNGs.</td>
<td>BRNGs (different for RNGs with implicit and</td>
</tr>
<tr>
<td></td>
<td>• Mechanisms of creating, copying, and deleting</td>
<td>Subroutines for saving and restoring BRNG seed.</td>
<td>explicit BRNG state use).</td>
</tr>
<tr>
<td></td>
<td>streams.</td>
<td>Subroutines for saving and restoring state</td>
<td>• Subroutines for saving and restoring BRNG</td>
</tr>
<tr>
<td></td>
<td>• Leapfrog method is supported by three</td>
<td>tables for GFSR and shuffled BRNGs.</td>
<td>seed. (for RNGs with implicit BRNG state use).</td>
</tr>
<tr>
<td></td>
<td>BRNGs.</td>
<td>• Leapfrog method is not supported.</td>
<td>• Leapfrog method is not supported.</td>
</tr>
<tr>
<td></td>
<td>• Skip-ahead method is supported by four</td>
<td>• Skip-ahead method is supported for three</td>
<td>• Skip-ahead method is not supported.</td>
</tr>
<tr>
<td></td>
<td>BRNGs.</td>
<td>BRNGs (limited functionality: skipping 100,000</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>elements only).</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distribution generators and other RNG related functionality</td>
<td>Vector interface only.</td>
<td>Some distribution generators have vector and scalar form, others are of either vector or scalar form only.</td>
<td>Some distribution generators have vector and scalar form, others are of either vector or scalar form only.</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>Vector/scalar interface</td>
<td>Vector interface only.</td>
<td>Some distribution generators have vector and scalar form, others are of either vector or scalar form only.</td>
<td>Some distribution generators have vector and scalar form, others are of either vector or scalar form only.</td>
</tr>
<tr>
<td>Supported hardware</td>
<td>• Intel architectures only. • Highly optimized functionality for target architecture.</td>
<td>Multiple platforms (e.g., Intel, Cray, HP, IBM, Sun).</td>
<td>Multiple platforms (e.g., Apple, Cray, HP, IBM, Sun).</td>
</tr>
</tbody>
</table>

**Note:** Intel MKL 7.0 Beta has been released recently. Among other improvements, Intel MKL 7.0 Beta provides new random-number generation capabilities: two quasi-random basic generators (Sobol and Niederreiter), multivariate (correlated) normal random-number generator as well as Poisson distribution generator with the mean varying in time.

**Sample Monte Carlo Applications**

The following discussions illustrate three example applications from various areas where Monte Carlo methods are widely used. We intentionally provide these examples in the simplest form to focus on different techniques that can be used for implementing efficient Monte Carlo simulation, rather than presenting new models or Monte Carlo methods.

The first example is a well-known ‘hit-or-miss’ Monte Carlo method that can be used to estimate areas or volumes. Special attention will be paid to demonstrate the ease of use of sequence-
splitting techniques such as the leapfrog and skip-ahead methods in sequential and parallel programming.

The Monte Carlo method plays an important role in everyday analysis of complex worldwide markets. The second example deals with the Black-Scholes option pricing model, demonstrating performance advantages of using highly optimized random-number generators in financial models. To do this we compare the application performance using random-number generators from the three numerical libraries discussed previously.

The third example deals with a light-propagation model with a Russian roulette technique. We discuss how the Russian roulette technique can be efficiently used in various Monte Carlo methods and provide some performance figures for this particular example.

All these examples were compiled using the Intel® C/C++ Compiler 7.1 [2] with maximum optimizations enabled for the target architecture.

**Evaluating Areas and Volumes**

A good starting point to demonstrate different techniques for implementing efficient Monte Carlo simulation is a ‘hit-or-miss’ Monte Carlo experiment to estimate the $\pi$ value. This example is the first example in many books on Monte Carlo studies, just as ‘Hello, world’ is the first example in many books on programming. For this reason, we do not provide a background of the problem (for more details see [9], for example) but focus on performance improvement techniques, which can be used in a variety of sequential and parallel Monte Carlo methods.

Three versions of the example application have been implemented. The first uses the standard C `rand()` function. In the second we have changed the program in order to use the Intel MKL vector uniform random-number generator. The third is a parallelized version of the second one using directives of the OpenMP* standard [7], which is supported in the Intel® Compilers.

The following table demonstrates the performance of these versions measured on the Intel Xeon processor-based system specified above.

<table>
<thead>
<tr>
<th>Version of the Example</th>
<th>Running Time (seconds)</th>
<th>Speedup vs. <code>rand()</code> version (times)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation using standard C <code>rand()</code> function</td>
<td>31.50</td>
<td>1.00</td>
</tr>
<tr>
<td>Include Intel MKL\VSL random-number generator</td>
<td>12.83</td>
<td>2.46</td>
</tr>
<tr>
<td>OpenMP version (2 threads)</td>
<td>6.83</td>
<td>4.61</td>
</tr>
</tbody>
</table>

The following table demonstrates the performance of these versions measured on Intel Itanium 2 processor-based system specified above.
A huge amount $N$ of 2-dimensional random points $(x, y)$ are generated within a unit square to calculate the $\pi$ estimation. In the first version, $x$ and $y$ components are generated by calling to the `rand()` function at each step:

In the second and third versions, we call the Intel MKL vector type random-number generator to generate random points by portions of large enough size $n$. Instead of generating $x$ and $y$ components into a single buffer at each step, we split the original random sequence into two subsequences of $x$ and $y$ components, respectively, by the leapfrog method discussed previously. Among other things, this helps the compiler to apply performance optimizations more effectively.

In the third version, we also generate $x$ and $y$ components by portions of size $n$ in two different buffers. The OpenMP version generates $k$ portions simultaneously, however, where $k$ is the
maximum number of threads available (the `omp_get_max_threads()` function is used to determine \( k \)). We create two random streams `streamx` and `streamy` for \( x \) and \( y \) components respectively, which are visible (shared) in each parallel thread. In addition, the copies of these streams are created for each thread `streamx_thread[i]` and `streamy_thread[i]`, where \( i \) takes values from 0 to \( k-1 \).

To track portions that have already been generated, we have a portion counter \( L \) holding the index of the portion to be used on the next iteration. In the parallelized loop, we first restore the initial state of streams `streamx_thread[i]` and `streamy_thread[i]` (\( i=omp_get_thread_num() \)) by copying the stream state from `streamx` and `streamy`, respectively (vslCopyStreamState function is used). After that, we just skip \( L \times n \) elements in the `streamx_thread[i]` and `streamy_thread[i]` using the vslSkipAheadStream function.

Our performance results show quite good efficiency of parallelizing on an SMP system. The same approach, however, works well for an MPI implementation as well. The only portion index \( L \) needs to be sent between distributed machines, and local estimations need to be collected from distributed machines at the final step to compute the final \( \pi \) estimation.

**Option Pricing**

Monte Carlo is a valuable tool for performing real-time financial analysis of complex worldwide markets. In our example, we consider the well-known Black-Scholes option-pricing model (see [23], [24] for further reading), which is, in fact, a framework for thinking about option pricing and is a de facto standard in the financial world.

The following table presents numerical and performance results on the Intel Xeon processor-based system for the three libraries with random-number generation capabilities, namely for Intel MKL 6.0, IMSL F90 MP 4.0, and NAG Fortran 77 Library (Mark 20). Speedups are measured against the slowest version.

<table>
<thead>
<tr>
<th>Library</th>
<th>Basic Generator</th>
<th>Option Value (Exact Value)</th>
<th>Absolute Error (Standard Error)</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Option Value Call Put</td>
<td>Absolute Error Call Put</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel MKL</td>
<td>MCG31m1</td>
<td>16.7291 (16.7341)</td>
<td>0.0050 (0.0029)</td>
<td>7.246 sec</td>
<td>8.46</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.2184 (7.2179)</td>
<td>0.0005 (0.0014)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
To generate random numbers from lognormal distribution, we used the following library routines: \texttt{vdRngLognormal} (Box-Muller2 method) in Intel MKL 6.0, \texttt{DRNLNL} in the IMSL, and the \texttt{G05LKF} subroutine in the NAG Fortran 77 library.

We used the ‘minimal standard’ basic generator as the fastest one in the IMSL library. For the same reason, we used the ‘original’ basic generator in NAG library. On the Intel MKL 6.0 side, we used two basic generators, namely MCG31m1 and MCG59. The latter is identical with the ‘original’ basic generator in NAG libraries, while properties of MCG31m1 are similar to the ‘minimal standard’ basic generator from IMSL.

**Light Propagation**

Historically, Monte Carlo is the method widely used in radiation transport problems. In recent decades, similar models are extensively used in other particle-transport problems as well. One of the vivid application areas is a photorealistic rendering.

The idea is to simulate and record particle histories as they are scattered and adsorbed. The goal of our simulation is to estimate spherical, cylindrical, and planar concentration of photons in the 3D medium over the time they are alive. The Russian roulette technique has been involved in the method to break particle trajectories without biasing the statistical estimation. In this example, we demonstrate how vector random-number generators can be efficiently used with Russian roulette technique.

The original Monte Carlo code is based on the published report [21]. The program source code is available [8]. At first sight, it is hard to use vector random-number generators there. Nevertheless, we believe that many Monte Carlo algorithms can be elegantly implemented using sequence-splitting techniques. A reward for that is a significant speedup in the performance of your application. The following table demonstrates the performance measured on an Intel Xeon processor-based system:

<table>
<thead>
<tr>
<th>Version of the Example</th>
<th>Running Time (seconds)</th>
<th>Speedup vs. Initial Version (times)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original code</td>
<td>99.52</td>
<td>1.00</td>
</tr>
</tbody>
</table>
Russian roulette is usually applied to estimators that are a sum of many terms:

\[ E = w_1 + w_2 + w_3 + \ldots \]

Often a contribution of many individual terms \( w_i \) is very small. In addition, \( w_i \) is often computationally expensive. The basic idea of Russian roulette is to randomly skip most of the evaluations associated with small contributions without biasing the final estimation.

In our example, \( E = E(S_j) \) means a photon concentration at given region \( S_j \) in the medium. Simulation is performed for a large number \( N \) of photon packets. The concentration of photons in the packet is characterized by the packet weight \( w_i \), where \( i = 1, N \). During a fixed time interval, each packet moves in a medium in a given direction by a random distance. Entering given region \( S_j \) we add the packet weight \( w_i \) to the estimation \( E(S_j) \). After each movement a fixed fraction of photons is absorbed (reducing packet weight).

The rest of the photons are randomly scattered (changing moving direction). After a fixed number of time steps the weight of the packet becomes too small and the Russian roulette technique is used to decide whether to break photons trajectory (with probability \( 1-p \)) or to allow them to survive (with probability \( p \)). To exclude the biasing of the estimation, the weight of surviving photons is increased by \( 1/p \) times:

\[
\tilde{w}_i = \begin{cases} 
  w_i/p & \text{with probability } p \\
  0 & \text{with probability } (1-p)
\end{cases}
\]

We start the propagation of a new photon packet as soon as the packet trajectory has been broken.

The vector version of the example tracks the vector of packets rather than individual packets. When the weight of the packets tracked becomes too small, we apply Russian roulette and remove ‘dead’ packets from the vector. In our implementation vector, size is reduced 16 times on average after each Russian roulette. When vector size becomes zero, we start tracking a new vector of packets.

Several random streams are created to perform the Monte Carlo simulation: one to generate random distances, two others to generate the new direction angles after scattering, and one to implement Russian roulette. Two Intel MKL\VSL basic generators have been used: the first to implement Russian roulette, and the second for generating distances and angles. The leapfrog method has been used to split original sequence into subsequences corresponding to the random distances and random angles.
Conclusion

Monte Carlo methods are extensively used in various areas of the science and technology dealing with the most complex numerical problems when traditional numerical methods are inefficient or simply inapplicable. In spite of the simple idea behind the Monte Carlo method, people performing Monte Carlo simulations encounter a number of issues that can interfere with the ease and efficiency of the implementation.

One of the acceptable decisions for developers is to shift the generation of the random numbers to a general-purpose numerical library. This article discussed requirements for a modern numerical library with random-number generator capabilities, which would make such a shift possible, natural, and reasonable. We reviewed the random-number generation features of the Intel MKL, IMSL and NAG numerical libraries.

Strengths of each library are outlined. Specifically IMSL F90 MP 4.0 and NAG Fortran 77 libraries have wider random-number generator functionality than Intel MKL 6.0. On the other hand, Intel Math Kernel Library random-number generation functionality provides advanced and flexible mechanisms to reduce the cost of user applications adoption to Intel MKL.

Intel Math Kernel Library random-number generators are highly optimized for a broad range of Intel processors including the Intel Pentium 4 processor, the Intel Xeon processor, and the Intel Itanium 2 processor. Three simple examples demonstrated how Intel MKL random-number generators can improve overall application performance significantly. Special attention has been paid in these examples to illustrate different techniques that can speed up Monte Carlo simulations.

About the Authors

Sergey A. Maidanov is a Senior Software Engineer for Intel® Corporation. He is a project lead of a team working on vector math and statistical components of Intel® Math Kernel Library and vector math component of Intel® Integrated Performance Primitives and takes part in implementation of runtime math libraries for Intel® C/C++ and Fortran Compilers. Sergey has M.S. from computer science department at Nizhny Novgorod State University. His research interests are in random number generation, numerical analysis and operations research.

Andrey Naraikin is a Senior Software Engineer for Intel® Corporation, where he has worked for over 6 years. Andrey has been a project lead working on Vector Math and Statistical components of Intel® Math Kernel Library and has taken part in implementation of run-time math libraries for Intel C/C++ and Fortran compilers for IA-32 and IA-64. Currently he is working with the Intel Compiler Lab on optimizing compilers development and performance analysis. Andrey has an M.S. from computer science department at Nizhny Novgorod State University. He can be contacted at Andrey.Naraikin@intel.com.
References

[1] Intel® Math Kernel Library
[2] Intel® C/C++ Compiler
[3] Visual Numerics*
[5] Top 500 Supercomputer Sites*
[6] NetLib Repository, Basic Linear Algebra Subprograms*
[7] OpenMP*
[8] Oregon Medical Laser Center*


[26] Monte Carlo Simulations Using Various Industry Library Solutions <docid=61383>

Intel Developer Centers

- Intel® Itanium® Processor
- Intel Xeon™ Processor
- High Performance Computing