Intel® Math Kernel Library

Notes for Intel® MKL Vector Statistics

Contents

Legal Information

1. Introduction

Computer simulation has become a new and commonly recognized approach to scientific research along with conventional experimentation. The latter harshly restricts a mathematical model that is supposed to be as sophisticated as the available conventional research methods permit. As for computer simulation, with ever-growing computing power, the degree of mathematical model complexity depends mainly on the researchers' understanding of phenomena they try to model. This is a key factor in the success that computer simulation has recently achieved in the scientific community.

1.1. What's New

These Notes document Intel® Math Kernel Library (Intel® MKL) 2019 Update 4 release.

The document has been updated to reflect the following changes to the product:

- Added description of the Chi-Square distribution algorithm. See Continuous Distribution Random Number Generators for details.
- Fixed inaccuracies.

1.2. About Vector Statistics

Optimization Notice

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

Notice revision #20110804

Vector Statistics (VS) performs pseudorandom and quasi-random vector generation as well as summary statistics calculations, convolution and correlation mathematical operations. VS is an integral part of Intel® Math Kernel Library (Intel® MKL).

VS provides a number of generator subroutines implementing commonly used continuous and discrete distributions to help improve their performance. All these distributions are based on the highly optimized Basic Random Number Generators (BRNGs) and Vector Mathematics (VM), a set of vector transcendental functions.

1.3. About This Document

Optimization Notice

This document includes a conceptual overview of random number generation problems and the product capabilities, with the focus on interpretation of results and the related figures of merit of VS Random Number Generators (RNGs). In contrast to the *Intel® Math Kernel Library Reference Manual*, VS Notes expand on the concept of random number generation and its application. The document provides extensive comparative analysis of the library generators and describes the basic tests applied. Apart from the VS distribution generators and service subroutines, the VS Notes describe testing of distribution generators.

If you are interested in general issues related to random number generators, their quality and applications in computer simulation, see Randomness and Scientific Experiment, Random Numbers and Figures of Merit for Random Number Generators sections that briefly cover the relevant matters and provide references for further studies.

To learn about the factors that help optimize the VS generators for Intel® processors, see the VS Structure section, which covers the concept underlying VS, its structure and potential for functionality enhancement. This section gives special attention to VS ease of use and other advantages in parallel programming.

For information on tests for the VS generators of various probability distributions, see the Testing of Basic Random Number Generators and Testing of Distribution Random Number Generators sections. You can see the latest test results in the Vector Statistics (VS) Performance Data document published at http://software.intel.com/en-us/articles/intel-math-kernel-library-documentation/.

NOTE

This document does not cover the fundamentals of the mathematical statistics, probability theory, or the theory of numbers and statistical simulation. You can find this information in books and articles listed in the Bibliography section.

1.4. Conventions

This document uses the following mathematical notations:

Notation	**Explanation**
$\oplus$	Bitwise exclusive OR.
&	Bitwise AND.
|	Bitwise OR.

2. Randomness and Scientific Experiment

Randomness is closely related to unpredictability of observation results and impossibility to predict them with sufficient accuracy. The nature of randomness is based on the lack of exhaustive information about the phenomenon under observation. As soon as you learn the origin of that phenomenon, you no longer consider it accidental or random. On the other hand, a random phenomenon whose origin has been revealed loses nothing of its random character. Randomness can be characterized as a type of relation stipulated by conditions that are inessential, superfluous, and extraneous to this particular phenomenon. Thus, the knowledge of the phenomenon is incomplete by definition.

Since our knowledge is incomplete, the observation results may prove impossible to predict with great accuracy. For instance, the initial state of the objects under observation may change imperceptibly for the used instruments, but these small changes may cause significant alterations in the final results. The sophisticated nature of the observed phenomenon may make accurate computation impossible in practice, if not in theory. Finally, even minor uncontrollable disturbing factors may cause serious deviations from a hypothetically "true value".

Although irregularities and deviations may occur, observational or experimental results still reveal a certain typical regularity called statistical stability. Various forms of statistical stability are formulated as specific rules that mathematical statistics calls laws of large numbers. This stability is the basis for the mathematical theory underlying the mathematical model of random phenomena. This theory is known as the theory of probability.

3. Random Numbers

Experimental observations are characterized by sets of distinctive features. These features fall into two groups:

1. quantitative features, such as results of measurements or calculations
2. qualitative features, such as the color of the object, occurrence or non-occurrence

Qualitative results may be presented as quantitative if some appropriate conventions have been developed and applied. Thus, a number can express the result even if the result is a particular quality feature. If the result is a random phenomenon, such number is called random.

Besides data from experimental observations, numerical methods produce imitations of a large amount of random numbers for various computational areas [Knuth81]. Methods that use random numbers to perform a simulation of phenomena are called Monte Carlo methods. Monte Carlo methods perform the most complex simulations in natural and social sciences, financial analysis, physics of turbulence, rarefied gas and fluid simulations, physics of high energies, chemical kinetics and combustion, radiation transport problems, and photorealistic rendering.

Monte Carlo methods can help you solve various numerical problems, such as:

1. ordinary stochastic differential equations
2. ordinary differential equations with random entries
3. boundary value problems for partial differential equations
4. integral equations
5. evaluation of high-dimensional integrals including path-dependent integrals.
6. random variables and order statistics simulation
7. stochastic processes
8. random samplings and permutations

For various reasons [Brat87], random number generation based on completely deterministic algorithms has become most common. It is obvious, however, that numbers obtained in a strictly deterministic way cannot be considered truly random as they only imitate randomness. In fact, such numbers are pseudorandom. Ideally, pseudorandom numbers imitate "truly" random numbers so well that without knowing the method of pseudorandom number generation and judging only by the output sequence, you cannot distinguish it within a reasonable time from a "truly" random sequence with more than 50% probability [L'Ecu94]. The output sequence of most pseudorandom number generators is easily predictable. This is acceptable because practical applications may not require strict unpredictability. However, there are certain applications for which most now existing pseudorandom generators are useless and at times simply dangerous. For example, the applications dealing with geometrical behavior of high-dimensional random vectors. Most of the existing pseudorandom generators should never be used for cryptographic purposes.

Pseudorandom number generators imitate finite sequences of independent identically distributed (i.i.d.) random numbers. However, some numerical methods do not really require independence between random numbers in a sequence. Such methods (for example, a numerical integration and optimization) fill the space with numbers as close to a given distribution as possible at the expense of independence. Such sequences do not look random at all and are called quasi-random (or low-discrepancy) sequences. The respective generators are called quasi-random number generators, and Monte Carlo methods dealing with quasi-random numbers are called Quasi-Monte Carlo methods.

Unlike pseudo- and quasi-random number generators based on deterministic algorithms, non-deterministic random number generators rely on physical phenomena, such as metastability, thermal noise in resistors, quantum or atmospheric noise. The most important feature of non-deterministic random number generators is unpredictability specified in terms of forward and backward resistance [NIST800-22]. Such generators are primarily used in cryptographic applications, gambling and other games of chance. High quality non-deterministic random number generators may be also applied in scientific simulations, especially those that are sensitive to multidimensional structure of random vectors.

Hereinafter, the term "random number generator", or RNG, refers to pseudo-, quasi-, and non-deterministic random number generators, unless the difference between them is important in the context.

4. Figures of Merit for Random Number Generators

This section discusses figures of merit for basic pseudo- and quasi-random number generators as well as for general (non-uniform) distribution generators.

4.1. Uniform Probability Distribution and Basic Pseudo- and Quasi-Random Number Generators

When considering various probability distributions, you should pay special attention to a uniform distribution over a certain interval. Firstly, such a distribution is very convenient for analysis. Secondly, an RNG of uniform distribution can always serve as a basis for an RNG of any other distribution type. That is why this document uses the term basic generators in reference to pseudorandom number generators of uniform distribution.

The observational output sequence of a basic generator should ideally possess the same properties as a sequence of independent variates uniformly distributed over a certain interval. It means that the basic generator should pass various statistical tests for uniformity and independence. It is an a priori fact that the output sequence of such generator is non-random. In other words, a fairly powerful statistical test can always be created for any individual BRNG, which the said generator will definitely fail. However, you should also consider the time required to detect "non-randomness" in the generator. For cryptography, the time required to detect non-randomness may be measured in years of testing conducted on a powerful cluster, while it may be significantly shorter for most of other applications.

Cryptographic RNGs can be slow for other fields. Most of applications benefit from simpler (and faster) generators, such as linear congruential, multiple recursive, feedback-shift-register, or add-with-carry.

Thus, checking the quality of BRNGs requires a "reasonable" set, or battery, of statistical tests. Ideally, you should choose such tests depending on types of problems the generator is intended to solve. A suitable test battery for general-purpose RNG libraries is fairly hard to choose, as the tests it should include are supposed to be versatile and sufficient for many simulation tasks. DIEHARD Battery of Tests by G. Marsaglia [Mars95] is an example of a good set of empirical tests for basic generators. Cryptographic standards such as NIST SP800-22 [NIST800-22] define the requirements including batteries of the tests for non-deterministic random number generators. Still, a specific application type may require a more complete generator testing.

Both empirical testing and theoretical methods are very important for estimating the quality of basic generators:

1. Theoretical evaluation is the first stage in rejecting bad generators. Theoretical research provides the basis for better understanding of generator properties, such as its period length, lattice structure, discrepancy, or equidistribution. The results obtained through theoretical testing refer to a basic generator used over the entire period.
2. Empirical tests ensure that the remaining generators are of acceptable quality by testing a generator over a small fraction of the period actually used.

Good behavior of k-dimensional random number vectors over the entire period indicates (yet does not prove) that similarly good statistical behavior may be observed over a smaller portion of the period [L'Ecu94].

Period of a basic generator is the most important feature that characterizes its quality. For example, one of the VS BRNGs - multiplicative congruential generator MCG31m1 - has a period length of about 2^{31}, which can be exhausted within seconds on modern Intel® processors. Taking into account that good statistical behavior of the generator is observed only over a fraction of its period (B.D. Ripley [Ripley87] recommends to take no more than a square root of the period length) you may not consider such period length acceptable. However, in Monte Carlo applications with a relatively small quantity of random numbers to be used, such generators may be useful because of the speed, small memory requirements for keeping the generator state, and efficient methods available for generation of random subsequences. For example, while estimating a global solution to an integral equation through the Monte Carlo method, the same random numbers should be used for different parameters [Mikh2000]. Modern computational capacities require BRNGs of at least 2^{60} period length. All other VS BRNGs meet these requirements.

Pseudorandom number generators are commonly recursive integer sequences in modular arithmetic, for example:

$$x_n = a_1 x_{n-1} + a_2 x_{n-2} + \ldots + a_k x_{n-k} (\bmod m)$$

Theoretical research looks for such values for parameters *k, ai, m* that result in good quality properties of the output sequence in terms of period length, lattice structure, discrepancy, equidistribution, and so on. In particular, if *m* is a prime number selected with proper coefficients *ai*, a period length of order *mk* may be obtained. Nevertheless, *m* is often taken as 2*p*(*p* >1) because of efficient modulo *m* reduction. Some authors do not recommend using *m* in the form of a power of 2 as the lower bits of the generated random numbers prove to be non-random on the whole. For example, see D. Knuth [Knuth81], P. L'Ecuyer [L'Ecu94]. However, this is irrelevant for most of Monte Carlo applications. Moreover, even if *m* is a prime number, you should also be careful when selecting random bits in the output sequence.

For the same reasons, quasi-random number generators filling a hypercube as evenly as possible are called in VS *Basic Random Number Generators* as well. Quasi-random sequences filling a space according to a non-uniform distribution can be generated by transforming a sequence produced by a basic quasi-random number generator. In most cases, tests designed for pseudorandom number generators cannot be used for quasi-random number generators. Special batteries of tests should be designed for basic quasi-random number generators.

4.2. Figures of Merit for General (Non-Uniform) Distribution Generators

A general distribution generator depends on the quality of the underlying BRNG. You can use several basic approaches to test general distribution generators.

Random number distributions are characterized by various measures: probability moments, central and absolute moments, quantiles, mode, scattering, skewness, and excess (kurtosis) coefficients, and so on. All the ordinary sample characteristics converge in probability to the corresponding measures of distribution when the sample size tends to infinity [Cram46]. Commonly, the characteristics based on the distribution moments are asymptotically normal with large sample sizes. Some classes of sample characteristics that are not based on sampling moments are also asymptotically normal, while others have quite different asymptotic behavior. When the limit probability distribution is known, you can build a statistical test to check whether a particular sample characteristic agrees with the corresponding measure of the distribution.

Of greatest practical value for simulation purposes are sample mean and variance that are the main properties of the distribution bias and scattering. All the VS random number generators undergo testing for agreement between distribution sampling moments (mean and variance) and theoretical values calculated for various sample sizes and distribution parameters.

Another class of valuable tests aims to check how well the sample distribution function agrees with the theoretical one. The most important tests are:

1. chi-square Pearson goodness-of-fit test (for discrete and continuous distributions)
2. Kolmogorov-Smirnov goodness-of-fit test (for continuous distributions)

Every VS distribution is tested with chi-square Pearson test over various sample sizes and distribution parameters.

It may be useful to transform the sequence that is being tested into one of the distributions, for example, into a uniform, normal, or multidimensional normal distribution. Then the transformed sequence is tested using a set of statistical tests that are specific for the distribution to which the sequence was transformed.

Tests that are based on simulation are Monte Carlo applications. Their choice is optional and should be made in accordance with the application field of the generator, the only requirement being an opportunity to verify the results obtained against the theoretical value. A good example of such a test application, which is used in checking the VS generators for quality, is the self-avoiding random walk [Ziff98].

5. Vector Statistics Structure

The VS component of Intel® MKL contains a set of generators to create general probability distributions, most commonly used in simulations, such as uniform, normal (Gaussian), exponential, or Poisson. Non-uniform distributions are generated using various transformation techniques applied to the output of a basic (pseudo-random, quasi-random, or non-deterministic) number generators.

To generate random numbers of a given probability distribution, you can either choose one of the available VS BRNGs or register your own BRNG. To enhance their performance, all VS BRNGs are highly optimized for various architectures of Intel® processors. Besides, VS provides a number of different techniques for transforming uniformly distributed random numbers into a sequence of required distribution.

All VS RNGs are of vector type. Unlike scalar type generators that return a successive random number, vector generators produce a vector of *n* successive random numbers of a given distribution with given parameters.

VS is a thread-safe, convenient for parallel computing, with a variety of configurations of parallel systems. A random stream is a notion in the RNG subcomponent of VS. A mechanism of streams provides simultaneous generation of several random number sequences produced by one or more BRNGs, as well as splitting of the original sequence into several subsequences by the leapfrog and block-split methods. Several random streams are particularly useful not only in parallel applications but in sequential programs as well.

5.1. Why Vector Type Generators?

Vector type library subroutines often perform much more efficiently than scalar type routines. The overhead expenses of scalar type routines are often comparable with the total time required for vector type computations, especially for highly optimized RNGs. To reduce overhead expenses, all VS RNG subroutines are of vector type.

Although vector type RNGs may require more careful programming, their use results in a substantial speedup in the overall application performance. VS provides a number of services to make vector programming as easy as possible. For further discussion, see the *Independent Streams. Leapfrogging and Block-Splitting* section of this document.

Vector type interface is useful for Monte Carlo methods because Monte Carlo requires a lot of random numbers rather than just one. You can call a vector RNG subroutine to generate a single random number as well, but such use is much less efficient.

5.2. Basic Random Number Generators

Basic Random Number Generators (BRNG) are used to obtain random numbers of various statistical distributions. Non-uniform distribution generators depend on the quality of the underlying BRNGs. You should choose the BRNG depending on your application requirements, such as:

1. quality requirements
2. speed
3. memory use
4. efficient generation of random number subsequences
5. using random numbers as real ones
6. using random numbers as a bit stream

For example, a BRNG that cannot provide true randomness for lower bits is still applicable to applications using variates as real numbers.

VS provides a variety of BRNGs and permits you to register user-defined basic generators and use them in the same way as the BRNGs available with VS. You can also use random numbers generated externally, for example, from a physical source of random numbers [Jun99]. This makes VS a general-purpose library suitable for various tasks. See Abstract Basic Random Number Generators. Abstract Streams section for details.

Having several basic RNGs of different types available in VS also enables you to get more accurate verification results. Result verification is very important for computational experimentation. Typically, a researcher cannot verify the output since the solution is simply unknown. Any verification process involves testing of each structural element of the system. Being one of such structural elements, an RNG may produce inadequate results. To get more reliable results of the experiment, many authors recommend using several different BRNGs in a series of computational experiments.

VS provides the following basic pseudo-, quasi-, and non-deterministic random number generators:

BRNG	**Type**	**Description**
MCG31m1	pseudorandom	A 31-bit multiplicative congruential generator.
R250	pseudorandom	A generalized feedback shift register generator.
MRG32k3a	pseudorandom	A combined multiple recursive generator with two components of order 3.
MCG59	pseudorandom	A 59-bit multiplicative congruential generator.
WH	pseudorandom	A set of 273 Wichmann-Hill combined multiplicative congruential generators (j = 1, 2, ... , 273).
MT19937	pseudorandom	Mersenne Twister pseudorandom number generator.
MT2203	pseudorandom	A set of 6024 Mersenne-Twister pseudorandom number generators (j = 1,...,6024).

SFMT19937	pseudorandom	SIMD-oriented Fast Mersenne Twister pseudorandom number generator.
SOBOL (with Antonov-Saleev [Ant79] modification)	quasi-random	A 32-bit Gray code-based generator producing low-discrepancy sequences for dimensions $1 \le s \le 40$
NIEDERREITER (with Antonov-Saleev [Ant79] modification)	quasi-random	A 32-bit Gray code-based generator producing low-discrepancy sequences for dimensions $1 \le s \le 318$.
PHILOX4X32X10	pseudorandom	A Philox4x32-10 counter-based pseudorandom number generator.
ARS5	pseudorandom	A counter-based pseudorandom number generator, which uses instructions from the AES-NI set. Available in IA® architectures supporting this instruction set.
ABSTRACT	pseudorandom or quasi-random, depending on the user-provided settings	Abstract source of random numbers. See Abstract Basic Random Number Generators. Abstract Streams section for details.
NON-DETERMINISTIC	Non-deterministic	[BMT], available in the latest CPUs such as [AVX].

Each VS basic generator consists of the following subroutines:

1. Stream Initialization Subroutine. See section Random Streams and RNGs in Parallel Computation for details.
2. Integer Output Generation Subroutine. Every generated integral value (within certain bounds) may be considered a random bit vector. For details on randomness of individual bits or bit groups, see Basic Random Generator Properties and Testing Results.
3. Single Precision Floating-Point Random Number Vector Generation Subroutine. The subroutine generates a real arithmetic vector of uniform distribution over the interval $[a, b)$.
4. Double Precision Floating-Point Random Number Vector Generation Subroutine. The subroutine generates a real arithmetic vector of uniform distribution over the interval $[a, b)$.

The sections below discuss each basic generator in more detail and provide references for further reading.

MCG31m1

32-bit linear congruential generators including MCG31m1 [L'Ecuyer99] are still used as the default RNGs in various systems because of the simplicity of implementation, speed of operation, and compatibility with earlier versions of the systems. However, their period lengths do not meet the requirements for modern BRNGs. Nevertheless, MCG31m1 has good statistical properties and may provide optimal results in generating random numbers of various distribution types for relatively small samplings.

$$x_n = ax_{n-1} (\mathrm{mod}\, m)$$
$$u_n = x_n / m$$
$$a = 1132489760,\ m = 2^{31} - 1$$

R250

R250 is a generalized feedback shift register generator. Feedback shift register generators have extensive theoretical footing and were first considered as RNGs for cryptographic and communications applications. Generator R250 proposed in [Kirk81] is fast and simple in implementation and is commonly used in the field of physics. However, the generator fails a number of tests, for example, a 2D self-avoiding random walk [Ziff98].

$$x_n = x_{n-103} \oplus x_{n-250}$$
$$u_n = x_n / 2^{32},$$

MRG32k3a

A combined generator MRG32k3a [L'Ecu99] meets the requirements for modern RNGs, such as good multidimensional uniformity or a fairly large period. Being optimized for various Intel® architectures, this generator rivals other VS basic RNGs in speed.

$$x_n = a_{11} x_{n-1} + a_{12} x_{n-2} + a_{13} x_{n-3} (\mathrm{mod}\, m_1)$$
$$y_n = a_{21} y_{n-1} + a_{22} y_{n-2} + a_{23} y_{n-3} (\mathrm{mod}\, m_2)$$
$$z_n = x_n - y_n (\mathrm{mod}\, m_1)$$
$$u_n = z_n / m_1$$
$$a_{11} = 0,\ a_{12} = 1403580,\ a_{13} = -810728,\ m_1 = 2^{32} - 209$$
$$a_{21} = 527612,\ a_{22} = 0,\ a_{23} = -1370589,\ m_2 = 2^{32} - 22853$$

MCG59

A multiplicative congruential generator MCG59 is one of the two basic generators implemented in Numerical Algorithms Group (NAG) Numerical Libraries [NAG]. Since the module of this generator is not prime, its period length is 2^{57} instead of 2^{59}, if the seed is an odd number. The drawback of such generators is that the lower bits of the output sequence are not random, therefore breaking numbers down into their bit patterns and using individual bits may be error-prone. For example, see [Knuth81], [L'Ecu94]. Besides, block-splitting of the sequence over the entire period into 2D similar blocks results in full coincidence of such blocks in d lower bits. For example, see [Knuth81], [L'Ecu94].

$$x_n = ax_{n-1} (\mathrm{mod}\, m)$$
$$u_n = x_n / m$$
$$a = 13^{13},\ m = 2^{59}$$

WH

Wichmann-Hill (WH) is a set of 273 different BRNGs. It is the second BRNG in NAG libraries. The constants $a_{i,j}$ are in the range from 112 to 127 and the constants $m_{i,j}$ are prime numbers in the range from 16718909 to 16776971, which are close to 2^{24}. These constants have been chosen so that they give good results with the spectral test, see [Knuth81] and [MacLaren89]. The period of each WH generator should be at least 2^{92}, but it is affected by common factors between $(m_{1,j} - 1)$, $(m_{2,j} - 1)$, $(m_{3,j} - 1)$, and $(m_{4,j} - 1)$. Still, each generator should have a period of at least 2^{80}. Further discussion of the properties of these generators is given in [MacLaren89], which shows that the generated pseudorandom sequences are essentially independent of one another according to the spectral test.

$$x_n = a_{1,j} x_{n-1} (\mathrm{mod}\, m_{1,j})$$

$$y_n = a_{2,j} y_{n-1} (\mathrm{mod}\, m_{2,j})$$

$$z_n = a_{3,j} z_{n-1} (\mathrm{mod}\, m_{3,j})$$

$$w_n = a_{4,j} w_{n-1} (\mathrm{mod}\, m_{4,j})$$

$$u_n = \left(x_n / m_{1,j} + y_n / m_{2,j} + z_n / m_{3,j} + w_n / m_{4,j} \right) \mathrm{mod}\, 1$$

NOTE

The variables x_n, y_n, z_n, w_n in the above equations define a successive member of integer subsequence set by recursion. The variable u_n is the generator real output normalized to the interval (0, 1).

MT19937

The Mersenne Twister pseudorandom number generator [Matsum98] is a modification of a twisted generalized feedback shift register generator proposed in [Matsum92], [Matsum94]. Properties of the algorithm (the period length equal to 2^{19937}).

$$x_n = x_{n-(k-m)} \oplus ((x_{n-k} \,\&\, p) \,|\, (x_{n-k+1} \,\&\, q))A,$$

$$y_n = x_n,$$

$$y_n = y_n \oplus (y_n >> w),$$

$$y_n = y_n \oplus ((y_n << s) \,\&\, b),$$

$$y_n = y_n \oplus ((y_n << t) \,\&\, c),$$

$$y_n = y_n \oplus (y_n >> l),$$

$$u_n = y_n / 2^{32},$$

$k = 624$, $m = 397$, $w = 11$, $s = 7$, $t = 15$, $l = 18$, $b = \mathrm{0x9D2C5680}$, $c = \mathrm{0xEFC60000}$,
$p = \mathrm{0x80000000}$,

$q = \mathrm{0x7FFFFFFF}$,

with matrix *A*(32x32) having following format:

$$A = \left| \begin{array}{ccccc} 0 & 1 & 0 & & \\ 0 & 0 & \ldots & 0 & \\ & & & \ldots & \\ & & 0 & 0 & 1 \\ a_{31} & a_{30} & \ldots & \ldots & a_0 \end{array} \right|$$

where 32-bit vector $a = a_{31} \ldots a_0$ has the value a = 0x9908B0DF.

MT2203

The set of 6024 MT2203 pseudorandom number generators is an addition to MT19937 generator used in large-scale Monte Carlo simulations performed on distributed multi-processor systems. Parameters of the MT2203 generators are calculated using the methodology described in [Matsum2000] that provides mutual independence of the corresponding random number sequences. Every MT2203 generator has a period length equal to 2^{2203}.

$x_{n,j} = x_{n-(k-m),j} \oplus ((x_{n-k,j} \,\&\, p) \,|\, (x_{n-k+1,j} \,\&\, q))A_j$,

$y_{n,j} = x_{n,j}$,

$y_{n,j} = y_{n,j} \oplus (y_{n,j} >> w)$,

$y_{n,j} = y_{n,j} \oplus ((y_{n,j} << s) \,\&\, b_j)$,

$y_{n,j} = y_{n,j} \oplus ((y_{n,j} << t) \,\&\, c_j)$,

$y_{n,j} = y_{n,j} \oplus (y_{n,j} >> l)$,

$u_{n,j} = y_{n,j} / 2^{32}$,

$k = 69$, $m = 34$, $w = 12$, $s = 7$, $t = 15$, $l = 18$, $p = \mathtt{0xFFFFFFE0}$,

$q = \mathtt{0x1F}$,

where matrix A_j(32x32) has the following format:

$$A_j = \left| \begin{array}{ccccc} 0 & 1 & 0 & & \\ 0 & 0 & \dots & 0 & \\ & & & \dots & \\ & & 0 & 0 & 1 \\ a_{31,j} & a_{30,j} & \dots & \dots & a_{0,j} \end{array} \right|,$$

with 32-bit vector $a_j = a_{31,j} \dots a_{0,j}$.

SFMT19937

The SIMD-oriented Fast Mersenne Twister pseudorandom number generator [Saito08] is analogous to the MT19937 generator and uses Single Instruction Multiple Data (SIMD) and multi-stage pipelining CPU features. SFMT19937 generator has a period of a multiple of $2^{19937}-1$ and better equidistribution property than MT19937.

$w_n = w_0 A \oplus w_M B \oplus w_{n-2} C \oplus w_{n-1} D$

where w_0, w_m, w_{n-2}... are 128-bit integers, and are A, B, C, D sparse 128 x 128 binary matrices for which wA, wB, wC, wD operations are defined as follows:

1. $wA := (w \overset{128}{<<} a) \oplus w$, left shift of 128-bit integer w by a followed by exclusive OR operation
2. $wB := (w \overset{32}{>>} b) \,\&\, mask$, right shift of each 32-bit integer in quadruple w by b followed by and-operation with quadruple of 32-bit masks $mask$, $mask = (mask_1\ mask_2\ mask_3\ mask_4)$

3. $wC := (w \overset{128}{>>} c)$, right shift of 128-bit integer *w* by *c*
4. $wD := (w \overset{32}{<<} d)$, left shift of each 32-bit integer in quadruple *w* by *d*
5. $u_{4n+k} = w_n(k) / 2^{32}$, $k = 0, 1, 2, 3$, $w_n(k)$ k-th 32-bit integer in quadruple w_n
6. Parameters of the generator take the following values: *a* = 8, *b* = 8, *c* = 8, *d* = 18, $mask_1$=0xBFFFFFFG, $mask_2$=0xBFFAFFFF, $mask_3$=0xDDFECB7F, $mask_4$=0xDFFFFFEF

SOBOL

Bratley and Fox [Brat88] provide an implementation of the Sobol quasi-random number generator. VS implementation permits generating Sobol's low-discrepancy sequences of length up to 2^{32}. This implementation also permits registering user-defined parameters (direction numbers or initial direction numbers and primitive polynomials) during the initialization, which allows obtaining quasi-random vectors of any dimension. If you do not supply user-defined parameters, the default parameter values [Brat88] are used for generation of quasi-random vectors. The default dimension of quasi-random vectors can vary from 1 to 40 inclusive.

$$\mathbf{x}_n = \mathbf{x}_{n-1} \oplus \mathbf{v}_c$$

$$\mathbf{u}_n = \mathbf{x}_n / 2^{32}$$

NOTE

The value *c* is the right-most zero bit in *n*-1; x_n is an s-dimensional vector of 32-bit values. The s-dimensional vectors (calculated during random stream initialization) v_i, *i* = 1,32 are called direction numbers. Vector u_n is the generator output normalized to the unit hypercube $(0,1)^5$.

NIEDERREITER

According to the results of Bratley, Fox, and Niederreiter [Brat92] Niederreiter's sequences have the best known theoretical asymptotic properties. VS implementation permits generating Niederreiter's low-discrepancy sequences of length up to 2^{32}. This implementation also permits registering user-defined parameters (irreducible polynomials or direction numbers), which allows obtaining quasi-random vectors of any dimension. If you do not supply user-defined parameters, the default parameter values [Brat92] are used for generation of quasi-random vectors. The default dimension of quasi-random vectors can vary from 1 to 318 inclusive.

$$\mathbf{x}_n = \mathbf{x}_{n-1} \oplus \mathbf{v}_c$$

$$\mathbf{u}_n = \mathbf{x}_n / 2^{32}$$

Philox4x32-10

This is a keyed family of counter-based BRNGs. The state consists of 128-bit integer counter *c* and two 32-bit keys k_0 and k_1. The generator output for each state consists of four 32-bit integer numbers obtained in the following way [Salmon2011]:

1. $c_n = c_{n-1} + 1$
2. $w_n = f(c_n)$, where *f* is a function that takes a 128-bit argument and returns a 128-bit number. The returned number is obtained as follows:

    1. The argument *c* is interpreted as four 32-bit numbers $c = \overline{L_1 R_1 L_0 R_0}$, where $\overline{ABCD} = A \cdot 2^{96} + B \cdot 2^{64} + C \cdot 2^{32} + D$, put $k_0^0 = k_0$ and $k_1^0 = k_1$.
    2. The following recurrence is calculated:

       $L_1^{i+1} = \mathrm{mullo}(R_1^i, 0xD2511F53)$

       $R_1^{i+1} = \mathrm{mulhi}(R_0^i, 0xCD9E8D57) xor\ k_0\ xor\ L_0$

       $L_0^{i+1} = \mathrm{mullo}(R_0^i, 0xCD9E8D57)$

       $R_0^{i+1} = \mathrm{mulhi}(R_1^i, 0xD2511F53)\ xor\ k_1\ xor\ L_1$

       $k_0^{i+1} = k_0^i + 0xBB67AE85$

       $k_1^{i+1} = k_1^i + 0x9E3779B9$

       Where `mulhi(a,b)` and `mullo(a,b)` are high and low 32-bit parts of the a*b product, respectively.
    3. Put $f(c) = \overline{L_1^N R_1^N L_0^N R_0^N}$, where $N = 10$
3. Integer output: $r_{4n+k} = w_n(k)$, where $w_n(k)$ is the *k*-th 32-bit integer in quadruple w_n, $k = 0, 1, 2, 3$
4. Real output: $u_n = r_n/2^{32}$

ARS5

ARS5 is a keyed family of counter-based BRNGs. The state consists of 128-bit integer counter *c* and a 128-bit key *k*. The BRNG is based on the AES encryption algorithm [FIPS-197]. The 32-bit output is obtained in the following way [Salmon2011]:

1. $c_n = c_{n-1} + 1$
2. $w_n = f(c_n)$, where *f* is a function that takes a 128-bit argument and returns a 128-bit number. The returned number is obtained as follows:
    1. Put $c_0 = c\ xor\ k$ and $k_0 = k$.
    2. The following recurrence is calculated N times:
        1. c_{i+1} = `SubBytes(c)`
        2. c_{i+1} = `ShiftRows(`c_{i+1}`)`
        3. c_{i+1} = `MixColumns(`c_{i+1}`)`, this step is omitted if $i + 1 = N$
        4. c_{i+1} = `AddRoundKey(`c_{i+1}, k_j`)`
        5. `Lo(`k_{i+1}`)` = `Lo(`k`)` + 0x9E3779B97F4A7C15

           `Hi(`k_{i+1}`)` = `Hi(`k`)` + 0xBB67AE8584CAA73B
    3. Put $f(c) = c_N$, where $N = 5$
3. Integer output: $r_{4n+k} = w_n(k)$, where $w_n(k)$ is the *k*-th 32-bit integer in quadruple w_n, $k = 0, 1, 2, 3$
4. Real output: $u_n = r_n/2^{32}$

Specification for the `SubBytes`, `ShiftRows`, `MixColumns` and `AddRoundKey` functions can be found in [FIPS-197].

ABSTRACT

Abstract basic generators enable VS distribution generators to be used with the underlying uniform random numbers that are already generated. This feature might be useful in the following cases:

1. Random numbers of the uniform distribution are generated externally [Mars95]. For example, in a physical device [Jun99].
2. You want to study the system using the same uniform random sequence but under different distribution parameters [Mikh2000]. It is unnecessary to generate uniform random numbers as many times as many different parameters you want to investigate.

There might be other cases when abstract basic generators are useful. See Abstract Basic Random Number Generators. Abstract Streams section for further reading. Because of the specificity of abstract basic generators, you cannot use `vslNewStream` and `vslNewStreamEx` functions to create abstract streams. VS provides special `vsliNewAbstractStream`, `vslsNewAbstractStream`, and `vsldNewAbstractStream` functions to initialize integer, single precision, and double precision abstract streams, respectively.

NON-DETERMINISTIC

This basic generator is an abstraction of the source of non-deterministic random numbers supported in hardware. A given hardware may support multiple non-deterministic sources. Each source supported by the library is associated with a unique identifier. By specifying the identifier during initialization of the basic generator, you can determine the non-deterministic random number generator to be used in the computations.

The current version of Intel MKL provides the interface to `RDRAND`-based non-deterministic source only, defined by identifier `VSL_BRNG_RDRAND` [AVX], [IntelSWMan].

Some non-deterministic sources may require non-constant time to generate a random number. In this case, multiple requests to a non-deterministic source may be required before the next random number becomes available. In extreme cases, for example, when the non-deterministic source fails, the generator may require infinite time to get the next random number. To avoid such situations, the number of retries for requests to the non-deterministic source is limited to `VSL_BRNG_NONDETERM_NRETRIES` equal to 10. You can redefine the maximum number of retries during the initialization of the non-deterministic random number generator with the `vslNewStreamEx` function. For details on the non-deterministic source implementation for Intel® processors, see Chapter 7.3.18 in [IntelSWMan] and Chapter 4.2.2 in [BMT].

NOTE

You can use non-deterministic random number generators only if the underlying hardware provides the respective support. For example, see Chapter 8 in [AVX] or Chapter 4 in [BMT] for instructions on how to determine whether an Intel® processor supports a non-deterministic random number generator.

5.3. Random Streams and RNGs in Parallel Computation

This section describes the usage model for random streams and RNGs, including their creation, initialization, copying, saving, and restoring.

5.3.1. Initializing a BRNG

To obtain a random number sequence from a given BRNG, you should assign initial, or seed values. The assigning procedure is called generator initialization. The C language function analogous with the initialization function is srand(seed) in stdlib.h. Different types of BRNGs require a different number of initial values. For example, the seed for MCG31m1 is an integral number within the range from 1 to 2^{31}-2, the initial values for MRG32k3a are a set of two triples of 32-bit digits, and the seed for MCG59 is an integer within the range from

1 to 2^{59}-1. Quasi-random generators require the dimension parameter on input. Thus, all BRNGs, including user-registered ones, require an individual initialization function.

To avoid limiting the versatility of the routines by providing individual initialization functions for each generator, VS offers an interface with a universal mechanism for generator initialization that encapsulates user-provided details of the initialization process. In line with this concept, VS offers two subroutines to initialize any BRNG (see the functions of random stream creation and initialization in the *Random Streams* section). These initialization functions can also be used to initialize user-provided functions.

One of the subroutines initializes a given BRNG using a single 32-bit initial value, which is called the seed. If the generator requires more than one 32-bit seed, VS initializes the remaining initial values on the basis of the original seed. Thus, generator R250, which requires 250 initial 32-bit values, is initialized using one 32-bit seed by the method described in [Kirk81].

The second subroutine is a generalization of the first one. It initializes a BRNG by passing an array of *n* 32-bit initial values. If the number of the initial values *n* is insufficient to initialize a given BRNG, the missing initial values are initialized by default values. If the number of the initial values *n* is excessive, the redundant values are ignored.

VS Notes document the initialization process for each library BRNG. For details, see Basic Random Generator Properties and Testing Results.

When calling initialization functions, you may ignore acceptability of the passed initial values for a given BRNG. If the passed seeds are unacceptable, the initialization procedure replaces them with acceptable values for a given type of BRNG. See Basic Random Generator Properties and Testing Results for details on acceptable initial values.

If you add a new BRNG to VS, you should implement an appropriate initialization function that supports the above mechanism of initial values passing, and, if required, apply the leapfrog and block-splitting techniques.

5.3.2. Creating and Initializing Random Streams

VS assumes that at any moment during the program execution you may simultaneously use several random number subsequences generated by one or more BRNGs. Consider the following scenarios:

1. The simulation system has several independent structural blocks of random number generation. For example, one block generates random numbers of normal distribution, another block generates uniformly distributed numbers, and so on. Each of the blocks should generate an independent random number sequence, that is, each block is assigned an individual stream that generates random numbers of a given distribution.
2. You need to study correlation properties of the simulation output with different distribution parameters. In this case, it looks natural to assign an individual random number stream (subsequence) to each set of the parameters. For example, see [Mikh2000].
3. Each parallel process (computational node) requires an independent random number subsequence of a given distribution, that is, a random number stream.

A random stream means a certain abstract source of random numbers. By linking such a stream to a specific BRNG and assigning specific initial values, you can predetermine the random number sequence produced by this particular stream. In VS, a universal `stream state descriptor` identifies every random number stream (in C language this is just a pointer to the structure). The descriptor specifies the dynamically allocated memory space that contains information on the respective BRNG and its current state, as well as some additional data necessary for the leapfrog and/or skip-ahead method.

VS has two stream creation and initialization functions:

```
vslNewStream( stream, brng, seed )
vslNewStreamEx( stream, brng, n, params )
```

Each of these subroutines allocates memory space to store information on the basic generator `brng`, its current state, etc., and then calls the initialization function of the basic generator `brng` that fills the fields of the generator current state with relevant initial values. The initial values are defined either by a single 32-bit value `seed` (for `vslNewStream`) or an array of n 32-bit initial values `params` (for `vslNewStreamEx`). The output of `vslNewStream` and `vslNewStreamEx` is a pointer to `stream`, that is, the stream state descriptor.

To initialize a non-deterministic random number generator, use the same NewStream-based mechanism as shown below:

```
errstatus = vslNewStream( stream, VSL_BRNG_NONDETERM, VSL_BRNG_RDRAND );
nretries = 5;
params[0] = VSL_BRNG_RDRAND;
params[1] = nretries;
errstatus = vslNewStreamEx( stream, VSL_BRNG_NONDETERM, 2, params );
```

If the underlying hardware does not support this non-deterministic generator, creation and initialization function returns the corresponding error code.

You can create any number of streams through multiple calls of `vslNewStream` or `vslNewStreamEx` functions. For example, you can generate several thread-safe streams that are linked to the same BRNG.

The generated streams are further identified by their stream state descriptors. Although a random number stream is a source of random numbers produced by a BRNG, that is, a generator of uniform distribution, you can generate random numbers of non-uniform distribution using streams. To do this, the stream state descriptor is passed to the transformation function that generates random numbers of a given distribution. Each function uses the stream state descriptor to produce random numbers of a uniform distribution, which are further transformed into sequences of the required distribution. See section Generating Methods for Random Numbers of Non-Uniform Distribution for details.

When a given random number stream is no longer needed, delete it by calling `vslDeleteStream` function:

```
vslDeleteStream( stream )
```

This function frees the memory space related to the stream state descriptor `stream`. After that, you can no longer use the descriptor.

NOTE

You can use non-deterministic random number generators only if the underlying hardware provides the respective support. For example, see Chapter 8 in [AVX] or Chapter 4 in [BMT] for instructions on how to determine whether an Intel® processor supports a non-deterministic random number generator.

5.3.3. Creating Random Stream Copy and Copying Stream State

VS provides an option of producing an exact copy of a generated stream by calling the vslCopyStream function:

```
vslCopyStream( newstream, srcstream )
```

A new stream `newstream` is created with parameters (stream descriptive information) that are exactly the same as those of the source stream `srcstream` at the moment of calling `vslCopyStream`. The stream state of `newstream` is exactly the same as that of `srcstream`, and both the streams generate random numbers using the same BRNG.

Another service function `vslCopyStreamState` copies the current state of the stream:

```
vslCopyStreamState( deststream, srcstream )
```

The streams `srcstream` and `deststream` are assumed to have been created by one of the above methods, both of the streams being related to the same BRNG. The function `vslCopyStreamState` copies the information about the current stream state from `srcstream` into `deststream`. Other stream-related information remains unchanged.

5.3.4. Saving and Restoring Random Streams

Typically, to get one more correct decimal digit in Monte Carlo, you need to increase the sample by the factor of 100. That makes Monte Carlo applications computationally expensive. Some of them take days or weeks while others may take several months of computations. For such applications, saving intermediate results to a file is essential to be able to continue computation using that result in case the application is terminated intentionally or abnormally.

In the case of BRNGs, saving intermediate results means that BRNG state and other descriptive data, if any, should be saved to a binary file. Since BRNG state is not directly accessible for the user, who operates with the random stream descriptor only, VS provides routines to save/restore random stream descriptive data to and from binary files:

```
errstatus = vslSaveStreamF( stream, fname )
errstatus = vslLoadStreamF( &stream, fname )
```

The binary file name is specified by the `fname` parameter. In the `vslSaveStreamF` function, a valid random stream to be written is specified by a stream input parameter. In `vslLoadStreamF`, the stream is the output parameter that specifies a random stream that has been created on the basis of the binary file data. Each of these functions returns the error status of the operation. A non-negative value indicates an error.

In addition to saving and restoring random stream descriptive data to and from binary files, VS provides the routines to save and restore the stream descriptive data to and from memory:

```
errstatus = vslSaveStreamM( stream, memptr );
errstatus = vslLoadStreamM( &stream, memptr );
```

Memory buffer is specified by the `memptr` pointer. To compute size of memory sufficient to hold random stream descriptive data, use VS service routine:

```
memsize = vslGetStreamSize( stream );
```

This routine returns the amount of memory in bytes necessary for random stream descriptive data.

Non-deterministic random number generators support service functions for saving and restoring random streams in a regular way. If you restore the state of the generator using `LoadStreamF` or `LoadStreamM` routines on the hardware that does not support the corresponding non-deterministic source, the functions return an error message.

5.3.5. Independent Streams. Block-Splitting and Leapfrogging

One of the basic requirements for random number streams is their mutual independence and lack of intercorrelation. Even if you want random number samplings to be correlated, such correlation should be controllable.

You can get independent streams using various methods. This document discusses the following methods supported by VS:

1. Using different parameter sets. For each stream, you may use the same type of generators (for example, linear congruential generators), but choose their parameters in such a way as to produce independent random number sequences. For example, the Mersenne Twister generator has 6024 parameter sets, which ensure that the resulting subsequences are independent (see [Matsum2000] for details). Another example is WH generator that can create up to 273 random number streams. The produced sequences are independent according to the spectral test (see [Knuth81] for the spectral test details).
2. Block-splitting. Split the original sequence into *k* non-overlapping blocks, where *k* is the number of independent streams. Each of the streams generates random numbers only from the corresponding block. This method is known as block-splitting or skipping-ahead.

3. Leapfrogging. Split the original sequence into *k* disjoint subsequences, where *k* is the number of independent streams, in such a way that the first stream would generate the random numbers x_1, x_{k+1}, x_{2k+1}, x_{3k+1}, ..., the second stream would generate the random numbers x_2, x_{k+2}, x_{2k+2}, x_{3k+2}, ..., and, finally, the k-th stream would generate the random numbers x_k, x_{2k}, x_{3k}, ... However, multidimensional uniformity properties of each subsequence deteriorate seriously as *k* grows. The method is useful if *k* is fairly small.

Karl Entacher presents data on inadequate subsequences produced by some commonly used linear congruential generators [Ent98].

VS permits you to use any of the above methods. Leapfrog and skip-ahead (block-split) methods are considered below in more detail.

Block-Splitting Method

VS implements block-splitting through function `vslSkipAheadStream`:

```
vslSkipAheadStream( stream, nskip )
```

The function changes the current state of the stream `stream` so that with the further call of the generator the output subsequence begins with the element x_{nskip} rather than with the current element x_0. Thus, if you wish to split the initial sequence into nstreams blocks of `nskip` size each, use the following sequence of operations:

Option 1

```
VSLStreamStatePtr stream[nstreams];
int k;
for ( k=0; k<nstreams; k++ )
{
  vslNewStream( &stream[k], brng, seed );
  vslSkipAheadStream( stream[k], nskip*k );
}
```

Option 2

```
VSLStreamStatePtr stream[nstreams];
int k;
vslNewStream( &stream[0], brng, seed );
for ( k=0; k<nstreams-1; k++ )
{
  vslCopyStream( &stream[k+1], stream[k] );
  vslSkipAheadStream( stream[k+1], nskip );
}
```

Leapfrog Method

VS implements the leapfrog method through function `vslLeapfrogStream`:

```
vslLeapfrogStream( stream, k, nstreams )
```

The function changes the stream `stream` so that the further call of the generator generates the output subsequence x_k, $x_{k+nstreams}$, $x_{k+2nstreams}$, ... rather than the output sequence x_0, x_1, x_2, ... Thus, if you wish to split the initial sequence into nstreams subsequences, the following sequence of operations should be implemented:

```
VSLStreamStatePtr stream[nstreams];
int k;
for ( k=0; k<nstreams; k++ )
{
  vslNewStream( &stream[k], brng, seed );
  vslLeapfrogStream( stream[k], k, nstreams );
}
```

NOTE

Block-splitting and leapfrog methods make programming with vector random number generators easier both in parallel applications and in sequential programs.

Not all VS BRNGs support both these methods of generating independent subsequences. The Leapfrog method is supported only when a BRNG provides a more efficient implementation than generation of the full sequence to pick out a required subsequence. The following table specifies the methods supported by different BRNGs:

BRNG	Leapfrog	Block-Splitting
MCG31m1	Supported	Supported
R250	-	-
MRG32k3a	-	Supported
MCG59	Supported	Supported
WH	Supported	Supported
MT19937	-	Supported
SFMT19937	-	Supported
MT2203	-	-
SOBOL	Supported to pick out individual components of quasi-random vectors	Supported
NIEDERREITER	Supported to pick out individual components of quasi-random vectors	Supported
PHILOX4X32X10	-	Supported
ARS5	-	Supported
ABSTRACT	-	-
NON-DETERMINISTIC	-	-

To initialize `nstreams` independent streams for the MT2203 set of generators, you can use the following code sequence:

```
...
#define nstreams 6024
...
VSLStreamStatePtr stream[nstreams];
int k;
```

```
for ( k=0; k< nstreams; k++ )
{
  vslNewStream( &stream[k], VSL_BRNG_MT2203+k, seed );
}
...
```

5.3.6. Abstract Basic Random Number Generators. Abstract Streams

If you want to use a BRNG not included in VS together with VS distribution generators, you can register your generator using the `vslRegisterBrng` function. In this case, your own BRNG should meet VS BRNG interface requirements. Otherwise, you can use VS abstract BRNGs as a wrapper.

CAUTION

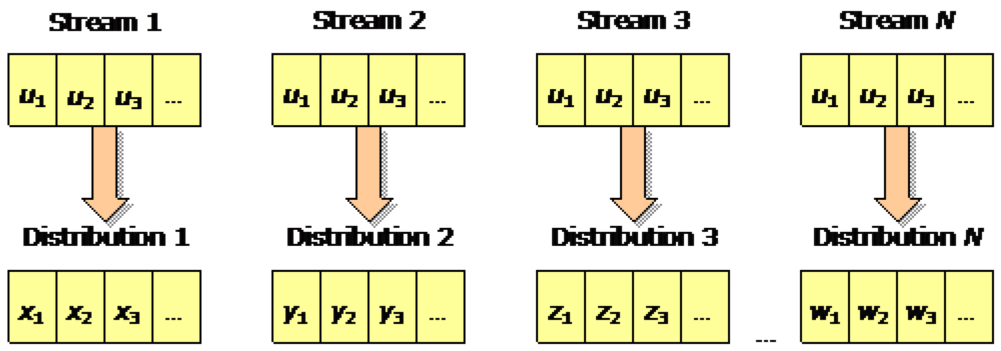
The Fortran flavor of the `vslRegisterBrng` function is not supported by VS. For Fortran applications, use VS abstract generator as a wrapper for your source of random numbers.

Abstract BRNGs are useful when you need to store random numbers in a buffer first and then use these numbers in distribution transformations. The reasons might be the following:

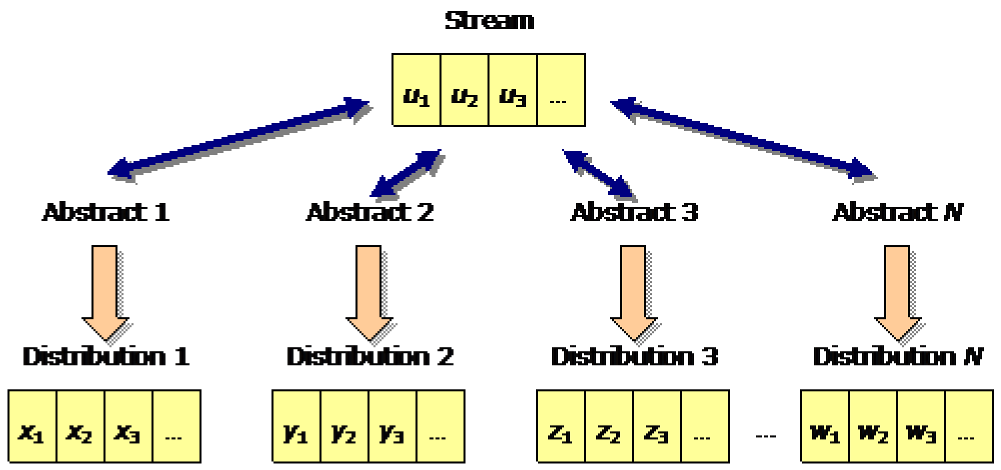
1. Random numbers are read from a file into a buffer.
2. Random numbers are taken from a physical device that stores numbers into a buffer.
3. You study the system behavior under different distribution generator parameters using the same BRNG sequence for each parameter set.
4. Your algorithm is sequential but you still want to use a vector random number generator.

While the first two cases do not require further discussion, the latter ones are considered below in more detail.

System is being studied under different parameters using the same BRNG sequence. One of the options is to create an identical stream for each parameter you want to study. Each of these streams is used with a particular distribution generator parameter set. In this case, a BRNG generates the underlying uniform sequence as many times as many distribution parameters are studied. See the following diagram for illustration:



If you use abstract BRNGs and respective abstract random streams instead, the underlying uniform sequence is generated only once and is stored in a buffer. Multiple copies of abstract random streams are associated with this buffer and are used with a particular distribution generator parameter set. See the following diagram for illustration:



The algorithm is essentially sequential. How to utilize vector random number generators? One of typical situations in Monte Carlo methods is when a mixture of distributions is used. Consider the following typical flowchart:

```
   For i from 1 to n, do:
/* Search for "good" candidate (u,v) */
   do
       u := Uniform() // get successive uniform random number from BRNG
       v := Uniform() // get successive uniform random number from BRNG
   until f(u,v)>a

    /* Get successive non-uniform random number */
   w := Nonuniform() // get successive uniform random number from BRNG
                     // and transform it to non-uniform random number

   /* Return i-th result */
   r[i] := g(u,v,w)
end do
```

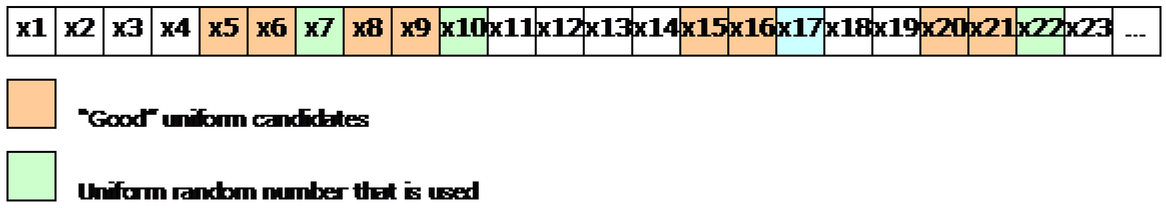
Minimization of control flow dependency is one of the valuable means to boost the performance on modern processor architectures. In particular, this means that you should try to generate and process random numbers as vectors rather than as scalars:

1. Generate vector *U* of pairs *(u, v)*
2. Applying "good candidate" criterion *f(u,v)>a*, form a new vector *V* that consists of "good" candidates only.
3. Get vector *W* of non-uniform random numbers *w*.
4. Get vector *R* of results *g(u,v,w)*.

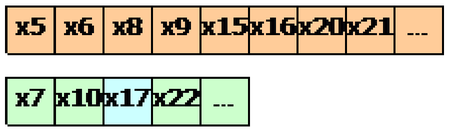
NOTE

Steps 1-4 do not preserve the original order of the underlying uniform random numbers utilization. Consider an example below, if you need to keep the original order.

Suppose that one underlying uniform random number is required per a non-uniform number. So, the underlying uniform random numbers are utilized as follows:


x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15 x16 x17 x18 x19 x20 x21 x22 x23 ...
"Good" uniform candidates
Uniform random number that is used

To keep the original order of the underlying uniform random number utilization and apply the vector random number generator effectively, pack "good" candidates into one buffer while packing random numbers to be used in non-uniform transformation into another buffer:



To apply non-uniform distribution transformation, that is, to use a VS distribution generator, for *x7*, *x10*, *x17*, *x22*, ... stored in a buffer *W*, you need to create an abstract stream that is associated with buffer *W*.

Types of Abstract Basic Random Number Generators

VS provides three types of abstract BRNGs intended for:

1. integer-valued buffers
2. single precision floating-point buffers
3. double precision floating-point buffers

The corresponding abstract stream initialization subroutines are:

```
vsliNewAbstractStream( &stream, n, ibuf, icallback );

vslsNewAbstractStream( &stream, n, sbuf, a, b, scallback );

vsldNewAbstractStream( &stream, n, dbuf, a, b, dcallback );
```

Each of these routines creates a new abstract stream `stream` and associates it with a corresponding cyclic buffer `[i,s,d]buf` of length *n*. Data in floating-point buffers is supposed to have uniform distribution over [*a*,*b*) interval. A mandatory parameter is a user-provided `callback` function `[i,s,d]callback` to update the associated buffer when the quantity of random numbers required in the distribution generator becomes insufficient in that buffer.

A user-provided `callback` function has the following format:

```
int MyUpdateFunc( VSLStreamStatePtr stream, int* n, <type> buf, int* nmin, int*
nmax, int* idx )
{
    ...
    /* Update buf[] starting from index idx */
    ...
    return nupdated;
}
```

For Fortran-interface compatibility, all parameters are passed by reference. The function renews the buffer `buf` of size *n* starting from position `idx`. The buffer is considered as cyclic and index `idx` varies from 0 to *n*-1. The minimal number of buffer entries to be updated is `nmin`. The maximum number of buffer entries that can be updated is `nmax`. To minimize callback call overheads, update as many entries as possible (that is, `nmax` entries), if the algorithm specifics permit this.

If you use multiple abstract streams, you do not need to create multiple `callback` functions. Instead, you may have one `callback` function and distinguish a particular abstract stream and a particular buffer using the `stream` and `buf` parameters, respectively.

The `callback` function should return the quantity of numbers that have been actually updated. Typically, the return value would be a number between `nmin` and `nmax`. If the `callback` function returns 0 or the number greater than `nmax`, the abstract BRNG reports an error. However, you can update less than `nmin` numbers (but greater than 0). In this case, the corresponding abstract generator calls the `callback` function again until at least `nmin` numbers are updated. Of course, this is inefficient but may be useful if no `nmin` numbers are available during the `callback` function call.

The respective pointers to the `callback` functions are defined as follows:

```
typedef int (*iUpdateFuncPtr)( VSLStreamStatePtr stream, int* n, unsigned int
ibuf[], int* nmin, int* nmax, int* idx );
typedef int (*dUpdateFuncPtr)( VSLStreamStatePtr stream, int* n, double dbuf[],
int* nmin, int* nmax, int* idx );
typedef int (*sUpdateFuncPtr)( VSLStreamStatePtr stream, int* n, float sbuf[],
 int* nmin, int* nmax, int* idx );
```

On the user level, an abstract stream looks like a usual random stream and can be used with any service and distribution generator routines. In many cases, more careful programming is required while using abstract streams. For example, you should check the distribution generator status to determine whether the `callback` function has successfully updated the buffer. Besides, a buffer associated with an abstract stream must not be updated manually, that is, outside of a `callback` function. In particular, this means that the buffer should not be filled with numbers before abstract stream initialization with `vsl[i,s,d]NewAbstractStream` function call.

You should also choose the type of the abstract stream to be created carefully. This type depends on a particular distribution generator routine. For instance, all single precision continuous distribution generator routines use abstract streams associated with single precision buffers, while double precision distribution generators use abstract streams associated with double precision buffers. Most of discrete distribution generators use abstract streams that are associated with either single or double precision abstract streams. See the following table to choose the appropriate type of an abstract stream:

Type of Discrete Distribution	**Type of Abstract Stream**
Uniform	Double precision
UniformBits	Integer
Bernoulli	Single precision
Geometric	Single precision
Binomial	Double precision
Hypergeometric	Double precision
Poisson (VSL_METHOD_IPOISSON_POISNORM)	Single precision
Poisson (VSL_METHOD_IPOISSON_PTPE)	Single and double precision
PoissonV	Single precision

NegBinomial	Double precision

The following example demonstrates generation of random numbers of the Poisson distribution with parameter *Λ = 3* using an abstract stream. Random numbers are assumed to be uniform integers from 0 to 2^{31} *ran_nums.txt* file. In the `callback` function, the numbers are transformed to double precision format and normalized to (0,1) interval.

```
#include <stdio.h>
#include "mkl_vsl.h"

#define METHOD        VSL_METHOD_IPOISSON_PTPE
#define N             4500
#define DBUFN         1000
#define M 0x7FFFFFFF /* 2^31-1 */

static FILE* fp;

int MydUpdateFunc(VSLStreamStatePtr stream, int* n, double dbuf[], int* nmin, int*
nmax, int* idx)
{
    int i;
    unsigned int num;
    double c;

    c = 1.0 / (double)M;
    for ( i = 0; i < *nmax; i++ )
    {
                  if ( fscanf(fp, "%u", &num) == EOF ) break;
        dbuf[(*idx+i) % (*n)] = num;
    }

    return i;
}

int main()
{
    int errcode;
    double lambda, a, b;
    double dBuffer[DBUFN];
    int r[N];
    VSLStreamStatePtr stream;

    /* Boundaries of the distribution interval */
    a  = 0.0;
    b = 1.0;

    /* Parameter of the Poisson distribution */
    lambda = 3.0;

    fp = fopen("ran_nums.txt", "r");

    /***** Initialize stream *****/
    vsldNewAbstractStream( &stream, DBUFN, dBuffer, a, b, MydUpdateFunc );

    /***** Call RNG *****/
    errcode = viRngPoisson(VSL_RNG_METHOD_POISSON_PTPE,stream,N,r,lambda);

    if (errcode == VSL_ERROR_OK)
    {
       /* Process vector of the Poisson distributed random numbers */
       ...
    }
    else
    {
       /* Process error */
       ...
   }
   ...

   vslDeleteStream( &stream );
   fclose(fp);

return 0;
}
```

5.4. Generating Methods for Random Numbers of Non-Uniform Distribution

You can use a source of uniformly distributed random numbers to generate both discrete and continuous distributions, which is implemented through a number of methods briefly described below.

5.4.1. Inverse Transformation

The probability distribution of a one-dimensional variate *X* may be most generally presented in terms of cumulative distribution function (CDF):

$$F(x) = \Pr(X \le x).$$

Any CDF is defined on the whole real axis and is monotonically increasing, where

$$F(-\infty) = 0; \quad F(+\infty) = 1.$$

In case of continuous distribution, the cumulative distribution function *F(x)* is a continuous one. In what follows, we assume that *F(x)* is steadily increasing, though assuming a non-steadily increasing function with a limited number of intervals where it steadily increases leads to trivial complications and generalizations of what follows.

Assuming the CDF steadily increases, the following single-valued inverse function should exist:

$$x = F^{-1}(u), \quad 0 \le u \le 1$$

It is easy to prove that, if *U* is a variate with a uniform distribution on the interval (0, 1), then the variate *X* is of *F(x)* distribution:

$$X = F^{-1}(U) \equiv G(U)$$

Thus, the inverse transformation method can be implemented as follows:

1. Generate a uniformly distributed random number meeting the requirements: $0 < u < 1$.
2. Assume $x = G(u)$ as a random number of the distribution *F(x)*.

The only drawback of this approach is that *G(u)* in a closed form is often hard to find, while numerical solution to the following equation is excessively time-consuming:

$$F(x) - u = 0$$

For discrete distributions, the CDF is a step function, the inverse transformation method still being applicable. For simplicity, assume that the distribution has probability mass points *k* = 0, 1, 2, ... with p_k probability. Then the distribution function is the sum:

$$F(x) = \sum_{k=0}^{\lfloor x \rfloor} p_k ,$$

where $\lfloor x \rfloor = \text{floor}(x)$ is the maximum integer that does not exceed *x*. If a continuous function *G(u)* exists in a closed form so that

$$G(F(k)) = k, \quad k = 0, 1, 2, \ldots ,$$

and *G(u)* is monotone, then generation of random numbers of the distribution *F(x)* can be implemented as follows:

1. Generate a uniformly distributed random number meeting the requirements: 0 < *u* < 1.
2. Assume *k = floor(G(u))* as a random number of the distribution *F(x)*.

For example, for the geometric distribution:

$$p_k = p \cdot (1-p)^k$$

Then *G(u)* does exist, as it is easy to prove:

$$G(u) = \frac{\ln(1-u)}{\ln(1-p)}$$

However, for most cases finding the closed form for *G(u)* function is too hard. An acceptable solution may be found using numerical search for *k* proceeding from

$$F(k-1) < u \le F(k)$$

With tabulated values of *F(k)*, the task is reduced to table lookup. As *F(k)* is a monotonically increasing function, you may use search algorithms that are considerably more efficient than exhaustive search. The efficiency is solely dependent on the size of the table.

You can apply an inverse transformation method to the s-dimensional quasi-random vectors. The resulting quasi-random sequence has the required s-dimensional non-uniform distribution.

5.4.2. Acceptance/Rejection

The cumulative distribution functions and their inverses may often be much more complex computationally than the probability density function (for continuous distributions) and the probability mass function (for discrete distributions).

$$F(x) = \int_{-\infty}^{x} f(t)dt, \quad f(x) - \text{probability density function}$$

$$F(x) = \sum_{k=0}^{\lfloor x \rfloor} p(k), \quad p(k) - \text{probability mass function}$$

Therefore, methods based on the use of density (mass) functions are often more efficient than the inverse transformation method.

Consider a case of continuous probability distribution:

Suppose, you need to generate random numbers *x* with distribution density *f*(*x*). Apart from the variate *X*, consider the variate *Y* with the density *g*(*x*), which has a fast method of random number generation and the constant *c* such that

$$f(x) \le cg(x), \quad -\infty < x < +\infty .$$

Then, it is easy to conclude that the following algorithm provides generation of random numbers *x* with the distribution *F(x)*:

1. Generate a random number *y* with the distribution density *g*(*x*).
2. Generate a random number *u* (independent of *y*) that is uniformly distributed over the interval (0, 1).
3. If *u≤f(y)/cg(y)*, accept *y* as a random number *x* with the distribution *F(x)*. Otherwise, go back to step 1.

The efficiency of this method depends on the degree of complexity of random number generation with distribution density *g(x)*, computational complexity for the functions *f(x)* and *g(x)*, as well as on the constant c value. The closer *c* is to 1, the lower the necessity to reject the generated *y*.

NOTE

Since quasi-random sequences are non-random, you should be careful when using quasi-random basic generators with the acceptance/rejection methods.

5.4.3. Mixture of Distributions

You can split the initial distribution into several simpler distributions:

$$F(x) = p_1F_1(x) + p_2F_2(x) + \ldots + p_kF_k(x), \quad \sum_{i=1}^{k} p_i = 1$$

In this case, random numbers for each of the distributions Fi(x) are easy to generate. An appropriate algorithm may be as follows:

1. Generate a random number *i* with the probability p_i.
2. Generate a random number *y* (independent of *i*) with the distribution $F_i(x)$.
3. Accept *y* as a random number *x* with the distribution *F*(*x*).

This technique is common in the acceptance/rejection method, when for the whole range of acceptable *x* values a density *g(x)*, which would approximate function *f(x)* well enough, is hard to find. In this case, the range is divided into sections so that *g(x)* looks relatively simple in each of the sub-ranges.

NOTE

Since quasi-random sequences are non-random, you should be careful when using quasi-random basic generators with the mixture methods.

5.4.4. Special Properties

To improve the efficiency of the algorithms based on the general methods described above, you may have to use special properties of distributions. For example, use of polar coordinates for a pair of independent normal variates enables you to develop an efficient method of random number generation based on 2D inverse transformation known as the Box-Muller method:

$$x_1 = \sqrt{-2\ln(u_1)}\sin 2\pi u_2$$
$$x_2 = \sqrt{-2\ln(u_1)}\cos 2\pi u_2$$

Generating s-dimensional normally distributed quasi-random sequences with 2D inverse transformation (the VS name is the Box-Muller2 method) is problematic when s is odd, because quasi-random numbers are generated in pairs. One of the options is to generate (s+1)-dimensional normally distributed quasi-random numbers from (s+1)-dimensional quasi-random numbers produced by a basic quasi-random generator and then ignore the last dimension.

Another option is to use the method that produces one normally distributed number from two uniform ones (the VS name is the Box-Muller method). In this case, to generate s-dimensional normally distributed quasi-random numbers, use 2s-dimensional quasi-random numbers produced by a basic quasi-random generator.

For a binomial distribution with parameters *m*, *p*, the probability mass function is as follows:

$$p_{m,p}(k) = C_m^k p^k (1-p)^{m-k}$$

For *p* > 0.5, it is convenient to use the fact that

$$p_{m,p}(k) = p_{m,1-p}(m-k)$$

Thus, you can convert a uniform distribution to a general distribution using a number of methods. Two different transformation techniques implemented for the same uniform distribution produce two different sequences of a general distribution, though possessing the same statistical properties.

Consider a simple example. If *U*1, *U*2 are two independent random values uniformly distributed over the interval (0, 1), that is, with the distribution function *F(x)* = *x* , 0 < *x* < 1, then the variate *X* = max(*U*1, *U*2) has the distribution *F(x)* ·*F(x)*. On the one hand, the random number x_1 with the maximum distribution from two independent uniform distributions may be derived either from a pair of uniformly distributed random numbers u_1, u_2 as x_1 = max(u_1, u_2) or from one uniform random number u_1 as x_1 = sqrt(u_1) by applying the inverse transformation method. It is obvious that applying two different methods to one and the same sequence u_1, u_2, u_3, ... gives two absolutely different sequences x_i.

You can perform transformations into non-uniform distribution sequences using various methods. The inverse transformation method may be preferable over the acceptance/rejection method for some applications and architectures, while the reverse preference is common for others. Taking this into account, the VS interface provides different options of random number generation for the same probability distribution. For example, a Poisson distribution may be transformed by two different methods: the first, known as `PTPE` [Schmeiser81], is based on acceptance/rejection and mixture of distributions techniques, while the second one is implemented through transformation of normally distributed random numbers.

The method number calls a method for a specified generator, for example:

```
viRngPoisson( VSL_METHOD_IPOISSON_PTPE, stream, n, r, lambda )
```

- calling `PTPE` method by passing the method number `VSL_METHOD_IPOISSON_PTPE`.

```

viRngPoisson( VSL_METHOD_IPOISSON_POISNORM, stream, n, r, lambda )
```

- calling transformation from normally distributed random numbers by passing the method number `VSL_METHOD_IPOISSON_POISNORM`.

For details on methods to be used for specific distributions, see Continuous Distribution Random Number Generators and Discrete Distribution Random Number Generators sections.

5.5. Accurate and Fast Modes of Random Number Generation

Using the distribution generators in the application, you can expect the obtained random numbers to belong to definitional domain of the corresponding distribution irrespective of its parameters. For example, random numbers x_i obtained as output of the relevant generator that are uniformly distributed on [*a*, *b*) are assumed to satisfy the following condition: $x_i \in [a, b)$ for all indices *i* and for all values of *a* and *b*. However, because of the specificity of floating-point calculations and rounding modes, some continuous distribution generators may produce random numbers lying beyond definitional domain for some particular values of distribution parameters. This is not acceptable in applications for which accuracy of calculations is critical.

To resolve this issue, VS defines two modes of random number generation: accurate and fast. A generation mode is initialized during the call of the distribution generator by specifying value of the method parameter. For example, an accurate generation of single precision floating-point numbers from the distribution uniform on [*a*, *b*) interval in C looks as follows:

```
...
status=vsRngUniform(VSL_METHOD_SUNIFORM_STD_ACCURATE, stream, n, r, a, b);
...
```

So, if a Monte Carlo application uses several distribution generators, each of them can be called in the preferable mode. When used in the accurate mode, the generators produce random numbers that belong to definitional domain for all parameter values of the distribution. The table below lists the generators that support the accurate mode of calculations:

Type of Distribution	**Data Types**
Uniform	s,d
Exponential	s,d
Weibull	s,d
Raleigh	s,d
Lognormal	s,d
Gamma	s,d
Beta	s,d

The distribution generators used in the fast mode produce numbers beyond the definitional domain in relatively rare cases. The application should set the accurate mode if all generated random numbers are expected to belong to the definitional domain irrespective of distribution parameter values. Use of the accurate mode may result in a slight performance degradation for random number generation.

5.6. Example of VS Use

A typical algorithm for VS generators is as follows:

1. Create and initialize a stream/streams. You can use functions `vslNewStream`, `vslNewStreamEx`, `vslCopyStream`, `vslCopyStreamState`, `vslLeapfrogStream`, `vslSkipAheadStream`.

2. Call one or more RNGs.
3. Process the output.
4. Delete the stream/streams. Use function `vslDeleteStream`.

You may reiterate steps 2-3. Random number streams may be generated for different threads.

The following example demonstrates generation of two random streams:

1. The first stream is the output of the basic generator MCG31m1. This stream is used to generate 1,000 normally distributed random numbers in blocks of 100 random numbers with parameters *a* = 5 and sigma = 2.
2. The second stream is the output of the basic generator R250. This stream is used to produce 1,000 exponentially distributed random numbers in blocks of 100 random numbers with parameters *a* = -3 and beta = 2.

For each of the streams, the seeds are equal to 1. Delete the streams after completing the generation. The purpose is to calculate the sample mean for normal and exponential distributions with the given parameters.

```
#include <stdio.h>
#include "mkl.h"
float rn[100], re[100];        /* buffers for random numbers */
float sn, se;                  /* averages */
VSLStreamStatePtr streamn, streame;
int i, j;
/* Initializing the streams*/
sn = 0.0f;
se = 0.0f;
vslNewStream( &streamn, VSL_BRNG_MCG31, 1 );
vslNewStream( &streame, VSL_BRNG_R250, 1 );
/* Generating */
for ( i=0; i<10; i++ )
{
  vsRngGaussian( VSL_METHOD_SGAUSSIAN_BOXMULLER2,
                 streamn, 100, rn, 5.0f, 2.0f );
  vsRngExponential(VSL_RNG_METHOD_EXPONENTIAL_ICDF,
                        streame, 100, re, -3.0f, 4.0f );
  for ( j=0; j<100; j++ )
  {
    sn += rn[j];
    se += re[j];
  }
}
sn /= 1000.0f;
se /= 1000.0f;
/* Deleting the streams */
vslDeleteStream( &streamn );
vslDeleteStream( &streame );
/* Printing the results */

printf( "Sample mean of normal distribution = %f\n", sn );
printf( "Sample mean of exponential distribution = %f\n", se );
```

When you call a generator of random numbers of normal (Gaussian) distribution, use the named constant `VSL_RNG_METHOD_GAUSSIAN_BOXMULLER2` to invoke the Box-Muller2 generation method. In the case of a generator of exponential distribution, assign the method by the named constant `VSL_RNG_METHOD_EXPONENTIAL_ICDF`.

The following example generates 100 three-dimensional quasi-random vectors in the $(2,3)^3$ hypercube using SOBOL BRNG.

```
#include <stdio.h>
#include "mkl.h"
float r[100][3]; /* buffer for quasi-random numbers */
VSLStreamStatePtr stream;

/* Initializing the streams*/
vslNewStream( &stream, VSL_BRNG_SOBOL, 3 );
/* Generating */
vsRngUniform( VSL_RNG_METHOD_UNIFORM_STD,
              stream, 100*3, (float*)r, 2.0f, 3.0f );
/* Deleting the streams */
vslDeleteStream( &stream );
```

The example below demonstrates the use of a non-deterministic random number generator for initialization of Mersenne Twister generator in parallel computations, so that each run of the application produces different random number sequences:

```
#define N 6024
#define METHOD1 VSL_RNG_METHOD_UNIFORMBITS_STD
#define METHOD2 VSL_RNG_METHOD_GAUSSIAN_ICDF
#define M 1024
int main()
{
    int i;
    VSLStreamStatePtr istream;
    VSLStreamStatePtr stream[N];
    unsigned seed[N];
    double buf[M];
    /* Initialize non-deterministic random number generator */
    errcode = vslNewStream( &stream, VSL_BRNG_NONDETERM, VSL_BRNG_RDRAND );
    if (errcode == VSL_RNG_ERROR_NONDETERM_NOT_SUPPORTED )
    {
         printf("CPU does not support non-deterministic generator");
         exit( 1 );
    }

    /* Generate seeds for MT2203 generator */
    errcode = viRngUniformBits( METHOD1, istream, N, seed );
    /* De-initialize */
    errcode = vslDeleteStream( &istream );
    for ( i = 0; i < N; i++ )
    {
        errcode = vslNewStream( &stream[i], VSL_BRNG_MT2203, seed[i] );
        errcode = vdRngGaussian( METHOD2, stream[i], M, buf, 0.0, 1.0 );
        /* Process Gaussian numbers in the buffer buf */
        ...
         errcode = vslDeleteStream( &stream[i] );
    }
    return 0;
}
```

The example below demonstrates the use of a non-deterministic random number generator in parallel Monte Carlo simulations:

```
#define NCORE 4
#define METHOD VSL_RNG_METHOD_GAUSSIAN_ICDF
#define M 1024
int main()
{
    int i;
    VSLStreamStatePtr stream[NCORE];
    for ( i = 0; i < NCORE; i++ )
    {
        /* Initialize non-deterministic random number generator */
        errcode = vslNewStream( &stream[i], VSL_BRNG_NONDETERM, VSL_BRNG_RDRAND );
        if (errcode == VSL_RNG_ERROR_NONDETERM_NOT_SUPPORTED )
        {
            printf("CPU does not support non-deterministic generator");
            exit( 1 );
        }
    }

    #pragma omp parallel for
    for ( i = 0; i < NCORE; i++ )
    {
        double buf[M];
        errcode = vdRngGaussian( METHOD, stream[i], M, buf, 0.0, 1.0 );
        /* Process Gaussian numbers in the buffer buf */
        ...
    }
    /* De-initialize */
    for ( i = 0; i < NCORE; i++ )
    {
        errcode = vslDeleteStream( &stream[i] );
    }
    return 0;
}
```

6. Testing of Basic Random Number Generators

This section provides information on testing the Basic Random Number Generators (BRNG), including details on BRNG properties and categories, as well as on interpretation of test results.

6.1. BRNG Implementations and Categories

Optimization Notice

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

Notice revision #20110804

Three implementations are available for every BRNG:

1. integer implementation (output is a 32-bit integer sequence)
2. real (single precision)
3. real (double precision)

You can use the BRNG integer output to obtain random bits or groups of bits. However, when you interpret the output of a generator, you should take into consideration the characteristics of each BRNG in general and its bit precision in particular. For detailed information on implementations of each BRNG, see Basic Random Generator Properties and Testing Results.

All VS BRNGs are tested by a number of specially designed empirical tests. These tests are applied for floating-point sequences or for integer-valued sequences.

The set of tests for BRNGs fall into the following categories:

1. tests to analyze the randomness of bits/groups of bits

2. tests to analyze the randomness of real random numbers normalized to the interval (0, 1)
3. tests to analyze conformance to the template

6.1.1. Randomness of Bits/Groups of Bits

Use the tests of this category to evaluate the BRNG integer implementation. The `viRngUniformBits` function corresponds to the integer implementation on the interface level. These tests analyze characteristics of each BRNG and its bit precision in particular. You can use the results of the tests to decide if you can apply this particular BRNG to obtain random bits or groups of bits. A failed test means that the interpretation of the integer output as the stream of random bits may result in an inadequate simulation outcome.

This category also includes a set of tests to determine the degree of randomness of upper, medium and lower bits. For example, upper bits may prove to be much more random than lower. Thus, some tests may indicate which bits or groups of bits are better for use as random ones.

6.1.2. Randomness of Real Random Numbers

This category contains different tests for BRNG normalized output. You can apply all these tests for real implementation of both single and double precision. Moreover, in most cases, the testing results are identical for both implementations, which proves that non-randomness of lower bits in the original integer sequence does not have practical influence on the randomness of the real BRNG output normalized to the (0, 1) interval. The `vsRngUniform` and `vdRngUniform` functions, for single and double precision, respectively, correspond to real implementations on the interface level.

6.1.3. Conformance to the Template

This category contains tests to check how a BRNG output conforms to the template. Template tests variations check if the leapfrog and skip-ahead methods generate subsequences of random numbers correctly. These tests are particularly important because, if any current member of the integer sequence differs from the template in a single bit only, the resulting sequence will be totally different from the template sequence. Also, the statistical properties of such a sequence are worse than those of the template sequence. This assumption is based on the fact that in a variety of sequences there are a very small number of "sufficiently random" sequences. As Knuth suggests, "random numbers should not be generated with a method chosen at random" [Knuth81].

6.2. Interpreting Test Results

Testing a generator for all possible seeds and sampling sizes is hardly practicable. Therefore, only a few subsequences of various lengths are actually tested.

Testing a random number sequence $u_1, u_2, \ldots, u_n$ gives a p-value that falls within the range from 0 to 1. Being a function of a random sampling, this p-value is a random number itself. For the sequence $u_1, u_2, \ldots, u_n$ of truly random numbers, the resulting p-value is supposed to be uniformly distributed over the interval (0, 1). Significant p-value deviation from the theoretical uniform distribution may indicate a defect in the tested sequence. For example, the sequence $u_1, u_2, \ldots, u_n$ may be considered suspicious if the resulting p-value falls outside the interval (0.01, 0.99). The chance to reject a "good" sequence in this case is 2%.

Multiple testing of different subsequences of a sequence substantiates the statistical conclusion about the sequence randomness.

6.2.1. One-Level (Threshold) Testing

To test K subsequences $u_1, u_2, ..., u_n; u_{n+1}, u_{n+2}, ..., u_{2n}; ...; u_{(K-1)n+1}, u_{(K-1)n+2}, ..., u_{Kn}$ of the original sequence, p-values $p_1, p_2, ..., p_K$ are computed. For a subsequence $u_{(j-1)n+1}, u_{(j-1)n+2}, ..., u_{jn}$ the test j fails if the value p_j falls outside the interval $(p_l, p_h) \subset (0, 1)$. The sequence $u_1, u_2, ..., u_{Kn}$ is considered suspicious when r or more test iterations failed.

Threshold testing for the VS generators was done with the following variable settings:

1. ten iterations (K=10)
2. the interval (p_l, p_h) equal to (0.05, 0.95)
3. $r = 5$.

The chance to reject a 'good' sequence in this case is 0.16349374% ≅ 0.2%.

6.2.2. Two-Level Testing

To test K subsequences $u_1, u_2, ..., u_n; u_{n+1}, u_{n+2}, ..., u_{2n}>; ...; u_{(K-1)n+1}, u_{(K-1)n+2}, ..., u_{Kn}$ of the original sequence, p-values $p_1, p_2, ..., p_K$ are computed. Since the resulting p-values for the sequence $u_1, u_2, ..., u_{Kn}$ of truly random numbers are supposed to be uniformly distributed over the interval (0, 1), a uniformity test can be performed for these p-values, returning p-value q_1 of the second level. Repeating this procedure L times results in obtaining L p-values of the second level $q_1, q_2, ... , q_L$ on which you can perform threshold testing.

We have conducted threshold second level testing for the VS generators with ten iterations (L=10) and applied the Kolmogorov-Smirnov goodness-of-fit test with Anderson-Darling statistics to evaluate $p_1, p_2, ..., p_K$ uniformity.

6.3. BRNG Tests Description

Most of empirical tests used for the VS BRNGs are well documented (for example, see [Mars95], [Ziff98]). This section describes these tests and the testing procedure in greater detail since tests may vary in their applicability and implementation for a particular BRNG. In addition, this section provides figures of merit that are used to decide on passing vs. failure in one- or two level testing. For the underlying criteria of the test, see Interpreting Test Results section.

NOTE

This section does not discuss non-deterministic RNGs. These generators are tested using a battery of NIST tests, a statistical test suite for random generators for cryptographic applications, [NIST800-22].

6.3.1. 3D Spheres Test

Test Purpose

The test uses simulation to evaluate the randomness of the triplets of sequential random numbers of uniform distribution. The stable response is the volume of the sphere. The radius of the sphere is equal to the minimal distance between the generated 3D points.

First Level Test

The test generates the vector u_i of 12,000 random numbers (i = 0, 1, ..., 11999), which are uniformly distributed in the (0, 1000) interval. The test forms 4,000 triplets of random numbers $x_k = (u_{3k}, u_3k_{+1}, u_{3k+2})(k = 0, 1, ..., 3999)$ situated in the cube $R = (0, 1000)x(0, 1000)x(0, 1000)$. Further, the test calculates $d_{min} = d(x_k, x_l)(l \neq k)$, where $d(x, y)$ is the Euclidean distance between x and y. In this case, the volume of the sphere with the d_{min}

radius should have the distribution close to the exponential one with $a = 0$, $b = 40p$ parameters. Thus, the distribution of the $p = 1 - \exp(-(d_{min})3/30)$ value should be close to the uniform distribution. The result of the first level test is the p-value.

Second Level Test

The second level test performs the first level test ten times. The p-value p_j, $j = 1, 2, ..., 10$ is the result of each first level test. The test applies the Kolmogorov-Smirnov goodness-of-fit test with Anderson-Darling statistic to the obtained set of p_j ($j = 1, 2, ..., 10$). If the resulting p-value is $p < 0.05$ or $p > 0.95$, the test fails.

Final Result Interpretation

The final result is the FAIL percentage for the failed first level tests. The test performs the second level test ten times. The acceptable result is the value of FAIL < 50%.

Tested Generators

Function Name	**Application**
`vsRngUniform`	applicable
`vdRngUniform`	applicable
`viRngUniform`	not applicable
`viRngUniformBits`	applicable

NOTE

The test transforms the integer output into the real output within the interval (0, 1) for the function `viRngUniformBits`. For detailed information about the normalization of the integer output see the description of the given BRNG.

6.3.2. Birthday Spacing Test

Test Purpose

The test uses simulation to evaluate the randomness of groups of 24 sequential bits of the integer output of a BRNG. The test analyzes all possible groups of the kind, that is, for example, from 0 to 23 bit, from 1 to 24 bit, and so on.

First Level Test

The first level test selects at random $m = 210$ "birthdays" from a "year" of $n = 224$ days. Then the test computes the spacing between the birthdays for each pair of sequential birthdays. The test then uses the spacings to determine the K value, that is, the number of pairs of sequential birthdays with the spacing of more than one day. In this case K should have the distribution close to the Poisson distribution with the $l = 16$ parameter. The first level test determines 200 values of Kj ($j = 1, 2, ..., 200$). To obtain the p-value p, the test applies the chi-square goodness-of-fit test to the determined values.

The integer output lists different interpretations for each BRNG. In the table below, NB stands for the number of bits required to represent a random number in integer arithmetic, WS stands for the machine word size, in bits, used in integer random number generation.

BRNG	**Integer Output Interpretation**
MCG31m1	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-30. NB=31, WS=32.
R250	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
MRG32k3a	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
MCG59	Array of 64-bit integers. Each 64-bit integer uses the following bits: 0-58. NB=59, WS=64.
WH	Array of quadruples of 32-bit integers. Each 32-bit integer uses the following bits: 0-23. NB=24, WS=32.
MT19937	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
MT2203	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
SFMT19937	Array of quadruples of 32-bit integers. Each 32-bit integer uses the following bits: 0-32. NB=32, WS=32.
PHILOX4X32X10	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
ARS5	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.

The test generates the dates of the birthdays in the following way:

1. Selects the b_s, b_{s+1}, ..., b_{s+23} bits from the next WS-bit integer of the integer output of `viRngUniformBits`.
2. Treats the selected bits as a 24-bit integer, that is, the number of the date on which the next birthday takes place and thus generates a birthday.
3. The test performs the steps 1 and 2 *m* times to generate *m* birthdays taken that the year consists of *n* days. The legitimate values *s* are different for each base generator (see the table above): 0 £ *s* £ NB - 24.

Second Level Test

The second level test performs the first level test ten times for the fixed *s*. The test applies the Kolmogorov-Smirnov goodness-of-fit test with Anderson-Darling statistics to the obtained set of $p_j (j = 1, 2 , ..., 10)$. If the resulting p-value is $p < 0.05$ or $p > 0.95$, the test fails for the given *s*.

Final Result Interpretation

The second level test performs ten times for each 0 £ *s* £ NB - 24. The test computes the FAILs percentage for the failed second level tests. The final result is the minimal percentage of the failed tests FAIL = min(FAIL_0, FAIL_1, ..., FAIL_{NB-24}) for 0 £ *s* £ NB - 24. The applicable result is the value of FAIL < 50%. Thus, the test determines if it is possible to select 24 random bits from every element of the integer output of the generator.

1. The integer output for the WH generator is the quadruples of the 32-bits values (x_i, y_i, z_i, w_i). In each 32-bit value only the lower 24 bits are significant.
2. The second level test performs ten times for the x_i element. Then the test computes the FAIL_x percentage the failed second level tests.
3. The second level test performs ten times for the y_i. Then the test computes the FAIL_y percentage for the failed second level tests.
4. The test performs the same procedure to compute the FAIL_z and FAIL_w values.

The final result is the minimal percentage of the failed tests FAIL = min(FAIL_x, FAIL_y, FAIL_z, FAIL_w). The acceptable result is the value of FAIL < 50%.

The test determines if it is possible to select 24 random bits from the fixed element *x*, *y*, *z* or *w* for each element of the integer output of the generator.

Tested Generators

Function Name	**Application**
`vsRngUniform`	not applicable
`vdRngUniform`	not applicable
`viRngUniform`	not applicable
`viRngUniformBits`	applicable

6.3.3. Bitstream Test

Test Purpose

The test uses simulation to check if it is possible to interpret the integer output of a BRNG as a sequence of random bits.

NOTE

The bit precision of a BRNG defines the sequence of random bits formation. For example, only 59 lower bits take part in the bit stream formation for the MCG59 generator, and only 31 lower bits for the MCG31 generator.

First Level Test

The first level test initially forms the sequence of bits $b_0, b_1, b_2, \ldots$ from the integer output of the BRNG and then forms 20-bit overlapping words $w_0 = b_0\, b_1 \ldots b_{19}$, $w_1 = b_1\, b_2 \ldots b_{20}, \ldots$ from the sequence. From the total number of 2021 formed words the test computes the quantity K of the missed 20-bit words. For the truly random sequence the K statistic distribution should be very close to normal with mean a = 141,909 and standard deviation s = 428. The test denotes the cumulative function of the normal distribution with these parameters as $F(x)$. The result is that the distribution of the p-value $p = F(K)$ should be uniform within the interval of (0, 1). In the table below, NB stands for the number of bits required to represent a random number in integer arithmetic, WS stands for the machine word size, in bits, used in integer random number generation.

BRNG	**Integer Output Interpretation**
MCG31m1	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-30. NB=31, WS=32.
R250	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
MRG32k3a	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
MCG59	Array of 64-bit integers. Each 64-bit integer uses the following bits: 0-58. NB=59, WS=64.
WH	Array of quadruples of 32-bit integers. Each 32-bit integer uses the following bits: 0-23. NB=24, WS=32.
MT19937	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
MT2203	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
SFMT19937	Array of quadruples of 32-bit integers. Each 32-bit

	integer uses the following bits: 0-31. NB=32, WS=32.
PHILOX4X32X10	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
ARS5	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.

The test selects:

1. NB of lower bits from each WS-bit integer to form a bit sequence
2. NB of lower bits from each of four WS-bit elements for WH generator

Second Level Test

The second level test performs the first level test 20 times. The result of each first level test is the p-value p_j, $j = 1, 2, ..., 20$. The test applies the Kolmogorov-Smirnov goodness-of-fit test with Anderson-Darling statistics to the obtained set of p_j, $j = 1, 2, ..., 20$. If the resulting p-value is $p < 0.05$ or $p > 0.95$, the test fails.

Final Result Interpretation

The final result of the test is the FAIL percentage of the failed second level tests. The second level test performs ten times. The acceptable result is the value of FAIL < 50%.

Tested Generators

Function Name	**Application**
`vsRngUniform`	not applicable
`vdRngUniform`	not applicable
`viRngUniform`	not applicable
`viRngUniformBits`	applicable

The lower bits are not random for multiplicative congruential generators where the module is the power of two (for example, MCG59), thus, the Bitstream Test fails for such generators.

6.3.4. Rank of 31x31 Binary Matrices Test

Test Purpose

The test evaluates the randomness of 31-bit groups of 31 sequential random numbers of the integer output. The stable response is the rank of the binary matrix composed of the random numbers. The test performs iterations for all possible 31-bit groups of bits (0-30, 1-31, ...) for generators with more than 31 bit precision.

First Level Test

The first level test selects, with *s* fixed, groups of bits b_s, b_{s+1}, ..., b_{s+30} from each element of the integer output and forms a binary matrix 31x31 in size from these 31 groups. The first level test composes 40000 of such matrices out of sequential elements of the integer output of the generator. Then the test computes the number of matrices with the rank of: 31, 30, 29, or less than 29. The following table shows the probability of these ranks in a truly random sequence:

Rank	**Probability in a Truly Random Sequence**
31	0.289
30	0.578
29	0.128
<29	0.005

Therefore, the test divides all possible matrix ranks into four groups. The test makes a V statistic with a chi-square distribution with three degrees of freedom for these four groups. Then the first level test applies the chi-square goodness-of-fit test to the groups. The testing result is the p-value.

In the table below, NB stands for the number of bits required to represent a random number in integer arithmetic, WS stands for the machine word size, in bits, used in integer random number generation.

NOTE

The acceptable values of 0 £ *s* £ NB - 31 are specific for each BRNG. The test is not applicable for the basic generator WH.

BRNG	**Integer Output Interpretation**
MCG31m1	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-30. NB=31, WS=32.
R250	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
MRG32k3a	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
MCG59	Array of 64-bit integers. Each 64-bit integer uses the following bits:

	0-58. NB=59, WS=64.
WH	Array of quadruples of 32-bit integers. Each 32-bit integer uses the following bits: 0-23. NB=24, WS=32.
MT19937	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
MT2203	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
SFMT19937	Array of quadruples of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
PHILOX4X32X10	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
ARS5	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.

The test selects only NB of lower bits from each WS-bit integer to form a bit sequence.

Second Level Test

The second level test performs the first level test ten times for the fixed *s*. The result is the set of p-values p_j, *j* = 1, 2, ..., 10.The test applies the Kolmogorov-Smirnov goodness-of-fit test with Anderson-Darling statistics to the obtained set of p_j, *j* = 1, 2, ..., 10. If the resulting p-value is $p < 0.05$ or $p > 0.95$, the test fails for the *s*.

Final Result Interpretation

The second level test performs ten times for each 0 £ *s* £ NB - 31. The test computes the FAIL percentage of the failed second level tests. The final result is the minimal percentage of the failed tests FAIL = min(FAIL_0, FAIL_1, ..., $\text{FAIL}_{\text{NB-31}}$) for 0 £ *s* £ NB - 31. The acceptable result is the value of FAIL < 50%. Therefore, the test indicates whether it is possible to single out at least 31 random bits out of each element of generator integer output such that 31 random numbers of 31 bits each have a random enough behavior under this particular test.

Tested Generators

Function Name	**Application**
`vsRngUniform`	not applicable
`vdRngUniform`	not applicable
`viRngUniform`	not applicable
`viRngUniformBits`	applicable

The Rank of 31x31 Binary Matrices Test cannot be applied to the generator WH as each element of this generator is only 24-bit.

6.3.5. Rank of 32x32 Binary Matrices Test

Test Purpose

The test evaluates the randomness of 32-bit groups of 32 sequential random numbers of the integer output. The stable response is the rank of the binary matrix composed of the random numbers. The test performs iterations for all possible 32-bit groups of bits (0-31, 1-32,...) for the generators with the bit precision of more than 32 bits.

First Level Test

The first level test selects, with *s* fixed, groups of bits b_s, b_{s+1}, ..., b_{s+31} from each element of the integer output. Then it forms a binary matrix 32x32 in size from these 32 groups. The first level test composes 40000 of such matrices out of sequential elements of the integer output of the generator.

Then the test computes the number of matrices with the rank of: 32, 31, 30, or less than 30. The following table shows the probability of these ranks in a truly random sequence:

Rank	**Probability in a Truly Random Sequence**
32	0.289
31	0.578
30	0.128
<30	0.005

Therefore, the test divides all possible matrix ranks into four groups. The test makes a V statistics with a chi-square distribution with three degrees of freedom for these three groups. Then the first level test applies the chi-square goodness-of-fit test to the groups. The testing result is the p-value.

In the table below, NB stands for the number of bits required to represent a random number in integer arithmetic, WS stands for the machine word size, in bits, used in integer random number generation.

NOTE

The acceptable values of 0 £ *s* £ NB - 32 are specific for each BRNG. The test cannot be applied to the WH generator as each element of this generator is only 24-bit. The test cannot be applied to the MCG31 generator as each element of this generator is only 31-bit.

BRNG	**Integer Output Interpretation**
MCG31m1	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-30. NB=31, WS=32.
R250	Array of 32-bit integers. Each 32-bit integer uses the following bits:

	0-31. NB=32, WS=32.
MRG32k3a	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
MCG59	Array of 64-bit integers. Each 64-bit integer uses the following bits: 0-58. NB=59, WS=64.
WH	Array of quadruples of 32-bit integers. Each 32-bit integer uses the following bits: 0-23. NB=24, WS=32.
MT19937	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
MT2203	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
SFMT19937	Array of quadruples of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
PHILOX4X32X10	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
ARS5	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.

The test selects only NB of lower bits from each WS-bit integer to form a bit sequence.

Second Level Test

The second level test performs the first level test ten times for the fixed *s*. The result is the set of p-values p_j, $j = 1, 2, ..., 10$.The test applies the Kolmogorov-Smirnov goodness-of-fit test with Anderson-Darling statistics to the obtained set of p_j, $j = 1, 2, ..., 10$. If the resulting p-value is $p < 0.05$ or $p > 0.95$, the test fails for the *s*.

Final Result Interpretation

The second level test performs ten times for each 0 £ *s* £ NB - 32. The test computes the FAIL percentage of the failed second level tests. The final result is the minimal percentage of the failed tests FAIL = min(FAIL_0, FAIL_1, ..., $\text{FAIL}_{\text{NB-32}}$) for 0 £ *s* £ NB - 32. The acceptable result is the value of FAIL < 50%. Therefore the test indicates whether it is possible to single out at least 32 random bits out of each element of generator integer output such that 32 random numbers of 32 bits each have a random enough behavior under this particular test.

Tested Generators

Function Name	**Application**
`vsRngUniform`	not applicable

`vdRngUniform`	not applicable
`viRngUniform`	not applicable
`viRngUniformBits`	applicable

6.3.6. Rank of 6x8 Binary Matrices Test

Test Purpose

The test evaluates the randomness of the 8-bit groups of 6 sequential random numbers of the integer output. The stable response is the rank of the binary matrix composed of the random numbers. The test checks all possible 8-bit groups: 0-7, 1-8, ...

First Level Test

The first level test selects, with s fixed, groups of bits b_s, b_{s+1}, ..., b_{s+7} from each element of the integer output and forms a binary matrix 6x8 in size from these 6 groups. The first level test composes 100000 of such matrices out of sequential elements of the integer output of the generator. Then the test computes the number of matrices with the rank of: 6, 5, or less than 5. The following table shows the probability of these ranks in a truly random sequence:

Rank	**Probability in a Truly Random Sequence**
6	0.773
5	0.217
<5	0.010

Therefore, the test divides all possible matrix ranks into three groups. The test makes a V statistic with a chi-square distribution with two degrees of freedom for these three groups. Then the first level test applies the chi-square goodness-of-fit test to the groups. The testing result is the p-value.

In the table below, NB stands for the number of bits required to represent a random number in integer arithmetic, WS stands for the machine word size, in bits, used in integer random number generation.

NOTE

The acceptable values of 0 £ *s* £ NB - 8 are specific for each BRNG. The test checks each of the four elements of the integer output for the WH and SFMT19937 basic generators.

BRNG	**Integer Output Interpretation**
MCG31m1	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-30. NB=31, WS=32.

R250	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
MRG32k3a	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
MCG59	Array of 64-bit integers. Each 64-bit integer uses the following bits: 0-58. NB=59, WS=64.
WH	Array of quadruples of 32-bit integers. Each 32-bit integer uses the following bits: 0-23. NB=24, WS=32.
MT19937	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
MT2203	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
SFMT19937	Array of quadruples of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
PHILOX4X32X10	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
ARS5	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.

The test selects only NB of lower bits from each WS-bit integer to form a bit sequence.

Second Level Test

The second level test performs the first level test ten times for the fixed s. The result is a set of p-values p_j, j = 1, 2, ..., 10. The test applies the Kolmogorov-Smirnov goodness-of-fit test with Anderson-Darling statistics to the obtained set of p_j, j = 1, 2, ..., 10. If the resulting p-value is p < 0.05 or p > 0.95, the test fails for the s.

Final Result Interpretation

The second level test performs ten times for each 0 £ s £ NB - 8. The test computes the FAIL percentage of the failed second level tests. The final result is the minimal percentage of the failed tests FAIL = min(FAIL$_0$, FAIL$_1$, ..., FAIL$_{NB-8}$) for 0 £ s £ NB - 31. The acceptable result is the value of FAIL < 50%. Therefore the test indicates whether it is possible to single out at least 8 random bits out of each element of generator integer output such that six random numbers of eight bits each have a random enough behavior under this particular test.

Tested Generators

Function Name	**Application**

`vsRngUniform`	not applicable
`vdRngUniform`	not applicable
`viRngUniform`	not applicable
`viRngUniformBits`	applicable

The Rank of 6x8 Binary Matrices Test checks each element of the WH generator separately as different multiplicative generators produce its elements.

6.3.7. Count-the-1's Test (Stream of Bits)

Test Purpose

The test evaluates the randomness of the overlapping random five-letter words sequence. The five-letter words have the specified distribution of the probabilities of obtaining the specified letter. The test forms the random letters from the integer output of a BRNG. The test treats the integer output as a sequence of bits.

First Level Test

The first level test assumes that the integer output is a sequence of random bits. The test interprets this bit sequence as a sequence of bytes, that is, a sequence of 8-bit integer numbers. The number of 1's in every random byte should have a binominal distribution with $m = 8$, $p = 1/2$ parameters. Therefore, the probability of getting k 1's in a byte is equal to $2^{-8}C_8^k$. The first level test regards a random variable c that takes five possible values:

$c = 0$, if the number of 1's in a random byte is less than three,

$c = 1$, if the number of 1's in a random byte is three,

$c = 2$, if the number of 1's in a random byte is four,

$c = 3$, if the number of 1's in a random byte is five,

$c = 4$, if the number of 1's in a random byte is more than five.

The probability distribution of c is the following:

$$q_0 = 2^{-8}\left(C_8^0 + C_8^1 + C_8^2\right); q_1 = 2^{-8}C_8^3; q_2 = 2^{-8}C_8^4; q_3 = 2^{-8}C_8^5; q_4 = 2^{-8}\left(C_8^6 + C_8^7 + C_8^8\right)$$

The test interprets c as a selection of a random letter from the alphabet {*a, b, c, d, e*} with the probabilities q_0, q_1, q_3, q_4 respectively. Thus, the sequence of random bytes $b_0, b_1, b_2, \ldots$ corresponds with the defined sequence of random letters $l_0, l_1, l_2, \ldots$. The test forms overlapping words of length four: $v_1 = l_1\, l_2\, l_3\, l_4$, $v_2 = l_2\, l_3\, l_4\, l_5$, ... and length five: $w_1 = l_1\, l_2\, l_3\, l_4\, l_5$, $w_2 = l_2\, l_3\, l_4\, l_5\, l_6$, ... from this sequence. The test computes the frequencies of getting each of 625 of possible four-letter words and of 3,125 of possible five-letter words for 2,560,000 of the obtained words. According to these frequencies, the test makes the chi-square statistics V1 and V2 for the four- and five-letter words respectively. The test takes into account the covariance of the frequencies of the fallouts of four-letter and five-letter words and performs the chi-square test for the V2 - V1 statistic. The V2 - V1 statistic is asymptotically normal with a mean $a = 2500$ and standard deviation $s = 70.71$. The result of the first level test is the p-value.

In the table below, NB stands for the number of bits required to represent a random number in integer arithmetic, WS stands for the machine word size, in bits, used in integer random number generation.

BRNG	Integer Output Interpretation
MCG31m1	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-30. NB=31, WS=32.
R250	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
MRG32k3a	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
MCG59	Array of 64-bit integers. Each 64-bit integer uses the following bits: 0-58. NB=59, WS=64.
WH	Array of quadruples of 32-bit integers. Each 32-bit integer uses the following bits: 0-23. NB=24, WS=32.
MT19937	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
MT2203	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
SFMT19937	Array of quadruples of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
PHILOX4X32X10	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
ARS5	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.

The test selects only NB of lower bits from each WS-bit integer to form a bit sequence.

Second Level Test

The second level test performs the first level test ten times. The test applies the Kolmogorov-Smirnov goodness-of-fit test with Anderson-Darling statistics to the obtained p-values of p_j, j = 1, 2, ..., 10. If the resulting p-value is p < 0.05 or p > 0.95, the test fails.

Final Result Interpretation

The second level test performs ten times. The test computes the FAIL percentage of the failed second level tests. The acceptable result is the value of FAIL < 50%.

Tested Generators

Function Name	**Application**
`vsRngUniform`	not applicable
`vdRngUniform`	not applicable
`viRngUniform`	not applicable
`viRngUniformBits`	applicable

The WH and SFMT19937 generators use all the four elements to form a bit sequence.

6.3.8. Count-the-1's Test (Stream of Specific Bytes)

Test Purpose

The test evaluates the randomness of the overlapping random five-letter words sequence. The five-letter words have the specified distribution of the probabilities of obtaining the specified letter. The test forms the random letters from the integer output of a BRNG. The test selects only 8 sequential bits from each element, starting with a certain fixed bit *s*.

First Level Test

The test selects the d_s, d_{s+1}, ..., d_{s+7} bits determining the next random byte from each element of the integer output, where 0 £ *s* £ NB - 8 (see the table below). The number of 1's in every random byte should have a binominal distribution with $m = 8$, $p = 1/2$ parameters. Therefore, the probability of getting k 1's in a byte is equal to $2^{-8}C_8^k$. The first level test regards a random number that takes five possible values:

$c = 0$, if the number of 1's in a random byte is less than three,

$c = 1$, if the number of 1's in a random byte is three,

$c = 2$, if the number of 1's in a random byte is four,

$c = 3$, if the number of 1's in a random byte is five,

$c = 4$, if the number of 1's in a random byte is more than five.

The probability distribution of c is the following:

$$q_0 = 2^{-8}\left(C_8^0 + C_8^1 + C_8^2\right); q_1 = 2^{-8}C_8^3; q_2 = 2^{-8}C_8^4; q_3 = 2^{-8}C_8^5; q_4 = 2^{-8}\left(C_8^6 + C_8^7 + C_8^8\right).$$

The test interprets c as a selection of a random letter from the alphabet {*a, b, c, d, e*} with the respective probabilities q_0, q_1, q_2, q_3, q_4. Thus, the sequence of random bytes b_0, b_1, b_2, ... corresponds with the defined sequence of random letters l_0, l_1, l_2, ... The test forms overlapping words of length four: $v_1 = l_1 l_2 l_3 l_4$, $v_2 = l_2 l_3 l_4 l_5$, ... and length five: $w_1 = l_1 l_2 l_3 l_4 l_5$, $w_2 = l_2 l_3 l_4 l_5 l_6$, ... from this sequence. The test computes the frequencies of getting each of 625 of possible four-letter words and of 3,125 of possible five-letter words for 256,000 of the obtained words. According to these frequencies, the test makes the chi-square statistics V1 and V2 for the four- and five-letter words respectively. The test takes into account the covariance of the frequencies of the fallouts of four-letter and five-letter words and performs the chi-square test for the V2 -V1 statistic. The V2 -

V1 statistic is asymptotically normal with a mean a = 2500 and standard distribution s = 70.71. The result of the first level test is the p-value.

In the table below, NB stands for the number of bits required to represent a random number in integer arithmetic, WS stands for the machine word size, in bits, used in integer random number generation.

BRNG	**Integer Output Interpretation**
MCG31m1	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-30. NB=31, WS=32.
R250	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
MRG32k3a	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
MCG59	Array of 64-bit integers. Each 64-bit integer uses the following bits: 0-58. NB=59, WS=64.
WH	Array of quadruples of 32-bit integers. Each 32-bit integer uses the following bits: 0-23. NB=24, WS=32.
MT19937	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
MT2203	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
SFMT19937	Array of quadruples of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
PHILOX4X32X10	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.
ARS5	Array of 32-bit integers. Each 32-bit integer uses the following bits: 0-31. NB=32, WS=32.

Second Level Test

The second level test performs the first level test ten times for the fixed 0 £ s £ NB - 8. The test applies the Kolmogorov-Smirnov goodness-of-fit test with Anderson-Darling statistics to the obtained p-values of p_j, j = 1, 2, ..., 10. If the resulting p-value is p < 0.05 or p > 0.95, the test fails for s.

Final Result Interpretation

The second level test performs ten times for each of 0 £ s £ NB - 8. The test computes the FAIL percentage of the failed second level tests. The final result is the minimal for 0 £ s £ NB - 8 percentage of the failed tests FAIL

= min($FAIL_0$, $FAIL_1$, ..., $FAIL_{NB-8}$). The acceptable result is the value of FAIL < 50%. Therefore, the test determines whether it is possible to select at least 8 random bits from each element of the integer output of the generator.

Tested Generators

Function Name	**Application**
`vsRngUniform`	not applicable
`vdRngUniform`	not applicable
`viRngUniform`	not applicable
`viRngUniformBits`	applicable

The test checks each of the four elements separately for the WH and SFMT19937 generators.

6.3.9. Craps Test

Test Purpose

The test evaluates the randomness of the output sequence of random numbers of the uniform distribution that imitates the process of dice tossing when gambling Craps. The stable response is the number of tosses of the pair of dice necessary to complete the game and the frequency of wins in the game.

First Level Test

The test forms a sequence of random numbers equiprobably taking the values from 1 to 6 from the output sequence of random numbers. The test treats every number as a number of spots on the face of a die. Thus the test regards a pair of numbers as the result of a toss of two dice. If on the first throw of dice the sum of the spots on the faces of dice equals to 7 or 11, it is a win; if the sum equals 2, 3 or 12, it is a loss. In other cases it is necessary to make additional throws to define the result of the game.

The test performs additional throws until the sum of the spots equals to 7 or coincides with the sum thrown on the first throw. If the sum equals to 7, it is a loss, otherwise, it is a win.

The theoretical probability of the win is 244/495, that is, a little less than 0.5. Further, the frequency of wins with the K-multiple repeats of the game, when K = 200,000, has a very close to normal distribution with mean a = K*244/495 and standard deviation s = a*251/495.

The number of throws necessary to complete the game can take the 1,2, ... values. On K-multiple iterations of the game, the test computes the frequencies of getting $c = 1$, $c = 2$, ..., $c = 20$, $c > 20$. Based on these frequencies, the test makes the chi-square statistics V with the chi-square distribution with 20 degrees of freedom.

The result of the first level test is the pair of p-values p and q for the number of tosses and the frequency of wins respectively.

Second Level Test

The test performs the first level test ten times. The result of each iteration of the first level test is the pair of p-values p_j and q_j, j = 1, 2, ..., 10. The test applies the Kolmogorov-Smirnov goodness-of-fit test with Anderson-

Darling statistics to the obtained p-values of p_j, $j = 1, 2, ..., 10$. If the resulting p-value is $p < 0.05$ or $p > 0.95$, the test fails. Similarly, the test applies the Kolmogorov-Smirnov goodness-of-fit test with Anderson-Darling statistics to the obtained p-values of q_j, $j = 1, 2, ..., 10$. If the resulting p-value is $q < 0.05$ or $q > 0.95$, the *s* test fails. The test passes in all other cases.

Final Result Interpretation

The final result of the test is the percentage FAIL of the failed second level tests. The test performs the second level test ten times. The acceptable result is the value of FAIL < 50%.

Tested Generators

Function Name	Application
`vsRngUniform`	applicable
`vdRngUniform`	applicable
`viRngUniform`	applicable
`viRngUniformBits`	applicable

6.3.10. Parking Lot Test

Test Purpose

The test evaluates the randomness of two-dimensional random points uniformly distributed in a square with a side length of 100. This is achieved by calculating the number of successfully "parked" points from the 12,000 random two-dimensional points.

First Level Test

The test assumes a next random point (x, y) successfully "parked", if it is far enough from every previous successfully "parked" point. The sufficient distance between the points (x_1, y_1) and (x_2, y_2) is $\min(|x_1 - x_2|, |y_1 - y_2|)>1$. Numerous experiments prove that out of 12,000 of truly random points only 3,523 points park successfully in average. Moreover, the number K of points successfully parked after 12,000 attempts has a close to normal distribution with:

1. mean $a = 3{,}523$
2. standard deviation $s = 21.9$

Consequently, $(K - a)/s$ should have a close to standard normal distribution with the $\Phi(x)$ cumulative distribution function. The result of the test is the p-value $p = \Phi((K - a)/s)$.

Second Level Test

The test performs the first level test ten times. The result of each iteration of the first level test is the p-value p_j, $j = 1, 2, ..., 10$. The test applies the Kolmogorov-Smirnov goodness-of-fit test with Anderson-Darling statistics to the obtained p-values of p_j, $j = 1, 2, ..., 10$. If the resulting p-value is $p < 0.05$ or $p > 0.95$, the test fails.

Final Result Interpretation

The final result of the test is the percentage of the failed second level tests. The test performs the second level test ten times. The acceptable result is the value of FAIL < 50%.

Tested Generators

Function Name	**Application**
`vsRngUniform`	applicable
`vdRngUniform`	applicable
`viRngUniform`	not applicable
`viRngUniformBits`	applicable

6.3.11. Self-Avoiding Random Walk Test

Test Purpose

The test evaluates the randomness of the output vector of the generator. The stable response is the frequency of achieving the upper side of the lattice by the point walking randomly along the sites.

First Level Test

A random particle walks along the sites of a square lattice. With each new step, the particle moves in one of possible directions one step forward corner-wise. A square lattice has two types of sides: the lower and left-hand sides are totally reflecting, while the upper and right-hand sides are totally adsorbing. Reaching the lower and left-hand sides, the vector of the movement direction makes a 90-degree bend. The upper and right-hand sides adsorb the particle when it reaches them and the walking process completes. The particle starts its movement from the lower left-hand site of the lattice in the northeast direction. If the particle encounters an unvisited site, it changes the direction vector with a 1/2 probability clockwise or counter-clockwise by 90 degrees and continues the walking process. If the particle encounters an already visited site of the lattice, it defines the movement direction according to the conditions of inadmissibility of re-tracing at least a part of the passed path.

Due to the symmetry of the task, either upper or the right-hand side should equiprobably adsorb the particle. The test determines the frequency of the achievement of the upper side of the lattice by the result of 500 iterations of the walking process. If M is the number of attempts when the particle reaches the upper side, then $K = (2M - 500)/\sqrt{500}$ has the close to standard normal distribution $\Phi(x)$. The result of the first level test is the p-value $p = \Phi(K)$.

Second Level Test

The test performs the first level test ten times. The result of each iteration of the first level test is the p-value p_j, $j = 1, 2, \ldots, 10$. The test applies the Kolmogorov-Smirnov goodness-of-fit test with Anderson-Darling statistics to the obtained p-values of p_j, $j = 1, 2, \ldots, 10$. If the resulting p-value is $p < 0.05$ or $p > 0.95$, the test fails.

Final Result Interpretation

The final result of the test is the percentage FAIL of the failed second level tests. The test performs the second level test ten times. The acceptable result is the value of FAIL < 50%.

Tested Generators

Function Name	Application
`vsRngUniform`	applicable
`vdRngUniform`	applicable
`viRngUniform`	not applicable
`viRngUniformBits`	applicable

6.3.12. Template Test

Test Purpose

The test evaluates the conformity of the generator output with the template sequence of random numbers. The test forms the specified output integer sequence $x_1, x_2, ..., x_k, x_{k+1}, ...$ from the recurrence specifying initial conditions. The parameters of the recurrences are selected such that the output sequences possess "good" properties (good multidimensional uniformity, large period, etc.). If the test computes any member of sequence x_k incorrectly, that results in incorrect computing of the other members $x_{k+1}, ...$ of the sequence. Moreover, if x_k differs from the correct (template) sequence in one bit, the subsequent members of sequence may differ significantly from the template sequence. The quality of the obtained sequence is probably much worse than the quality of the template sequence. That is why all VS BRNGs undergo thorough tests for template sequences conformity.

The test also checks the BRNGs with the random output numbers $u_1, u_2, ..., u_k, u_{k+1}, ...$, uniformly distributed over the (a, b) interval for the template output conformity.

Obviously, the output sequences are different for real arithmetic of single and double precision. For the integer output every number should coincide bitwisely with the template number. For the real output numbers, such coincidence is not necessary. The upper bits of the mantissa of the real output determine the quality of the output sequence; the lower bits do not influence randomness. For example, the coincidence of the upper binary digits of the mantissa is sufficient for most applications. (See the chapter Spectral Test in [Knuth81]).

This test is also used to validate VS basic quasi-random number generators.

Final Result Interpretation

The final result is the number of the sequence members that do not coincide with the template members. The value should be equal to 0.

For real sequences the test assumes that the sequence member coincides with the template member, if at least 8 upper binary digits of the mantissa coincide.

Tested Generators

Function Name	Application
`vsRngUniform`	applicable
`vdRngUniform`	applicable
`viRngUniform`	not applicable
`viRngUniformBits`	applicable

6.4. Basic Random Generator Properties and Testing Results

This section contains the empirical testing results for the VS BRNGs described in the BRNG Tests Description section and other information on the properties of BRNGs and the rules of the output vector interpretation.

6.4.1. MCG31m1

This is a 31-bit multiplicative congruential generator:

$$x_n = ax_{n-1}(\mathrm{mod}\, m)$$
$$u_n = x_n / m$$
$$a = 1132489760,\ m = 2^{31} - 1$$

MCG31m1 belongs to linear congruential generators with the period length of approximately 2^{31}. Such generators are still used as default random number generators in various software systems, mainly due to the simplicity of the portable versions implementation, speed and compatibility with the earlier systems versions. However, their period length does not meet the requirements for modern basic generators. Still, the MCG31m1 generator possesses good statistic properties and you may successfully use it to generate random numbers of different distributions for small samplings.

Real Implementation (Single and Double Precision)

The output vector is the sequence of the floating-point values u_0, u_1, ...

Integer Implementation

The output vector of 32-bit integers x_0, x_1, ...

Stream Initialization by Function `vslNewStream`

MCG31m1 generates the stream and initializes it specifying the input 32-bit parameter seed :

1. Assume `x`$_0$ `= seed mod 0x7FFFFFFF`
2. If $x_0 = 0$, assume $x_0 = 1$.

Stream Initialization by Function `vslNewStreamEx`

MCG31m1 generates the stream and initializes it specifying the array `params[]` of *n* 32-bit integers:

1. If $n = 0$, assume $x_0 = 1$
2. Otherwise, assume `x₀ = params[0] mod 0x7FFFFFFF`
   If $x_0 = 0$, assume $x_0 = 1$.

Subsequences Selection Methods

`vslSkipAheadStream`	supported
`vslLeapfrogStream`	supported

Generator Period

$$\rho = 2^{31} - 2 \approx 2.1 \times 10^{9}$$

Lattice Structure

$M_8 = 0.72771$, $M_{16} = 0.61996$, $M_{32} = 0.61996$ (for more details see [L'Ecu94]).

Empirical Testing Results Summary

Test Name	**`vsRngUniform`**	**`vdRngUniform`**	**`viRngUniform`**	**`viRngUniformBits`**
3D Spheres Test	OK (10% errors)	OK (10% errors)	Not applicable	OK (10% errors)
Birthday Spacing Test	Not applicable	Not applicable	Not applicable	OK (0% errors)
Bitstream Test	Not applicable	Not applicable	Not applicable	OK (10% errors)
Rank of 31x31 Binary Matrices Test	Not applicable	Not applicable	Not applicable	OK (10% errors)
Rank of 32x32 Binary Matrices Test	Not applicable	Not applicable	Not applicable	Not applicable
Rank of 6x8 Binary Matrices Test	Not applicable	Not applicable	Not applicable	OK (0% errors)
Counts-the-1's Test (stream of bits)	Not applicable	Not applicable	Not applicable	OK (20% errors)

Counts-the-1's Test (stream of specific bytes)	Not applicable	Not applicable	Not applicable	OK (0% errors)
Craps Test	OK (20% errors)	OK (20% errors)	OK (20% errors)	OK (20% errors)
Parking Lot Test	OK (10% errors)	OK (10% errors)	Not applicable	OK (10% errors)
2D Self-Avoiding Random Walk Test	OK (20% errors)	OK (20% errors)	Not applicable	OK (20% errors)

NOTE

1. The tabulated data is obtained using the one-level (threshold) testing technique. The OK result indicates FAIL < 50%. The run fails when p-value falls outside the interval [0.05, 0.95].
2. The stream tested is generated by calling the function `vslNewStream` with seed=7,777,777.

6.4.2. R250

This is a generalized feedback shift register generator:

$$x_n = x_{n-103} \oplus x_{n-250}$$
$$u_n = x_n / 2^{32}$$

Feedback shift register generators possess ample theoretical foundation and were initially intended for cryptographic and communication applications. Physicists widely use R250 generator, as it is simple and fast in implementation. However, this generator fails in some types of tests, one of which is the 2D Self-Avoiding Random Walk Test.

Real Implementation (Single and Double Precision)

The output vector is the sequence of the floating-point values u_0, u_1, ...

Integer Implementation

The output vector of 32-bit integers x_0, x_1, ...

Stream Initialization by Function `vslNewStream`

R250 generates the stream and initializes it specifying the input 32-bit integer parameter seed. The stream state is the array of 250 32-bit integers x_{-250}, x_{-249}, x_{-1}, initialized in the following way:

1. If seed = 0, assume seed = 1. Assume x_{-250} = seed.

2. Initialize x_{-249}, ..., x_0 according to recurrent correlation $x_{n+1} = 69069x_n(\bmod 2^{32})$.
3. Interpret the values x_{7k-247}, k = 0, 1, ..., 31 as a binary matrix of size 32x32 and perform the following: set the diagonal bits to 1, and the under-diagonal bits to 0.

Stream Initialization by Function `vslNewStreamEx`

R250 generates a stream and initializes it specifying the array n of 32-bit integer `params[]`:

1. If $n \geq 0$, assume x_{k-250} `= params[k]`, k=0,1,...,249.

If n = 0, assume seed = 1, and perform the initialization as described in the above section on stream initialization by the function `vslNewStream`.

Subsequences Selection Methods

`vslSkipAheadStream`	not supported
`vslLeapfrogStream`	not supported

Generator Period

$\rho = 2^{250} \approx 1.8 \times 10^{75}$

Empirical Testing Results Summary

Test Name	**`vsRngUniform`**	**`vdRngUniform`**	**`viRngUniform`**	**`viRngUniformBits`**
3D Spheres Test	OK (0% errors)	OK (0% errors)	Not applicable	OK (0% errors)
Birthday Spacing Test	Not applicable	Not applicable	Not applicable	OK (0% errors)
Bitstream Test	Not applicable	Not applicable	Not applicable	OK (25% errors)
Rank of 31x31 Binary Matrices Test	Not applicable	Not applicable	Not applicable	OK (10% errors)
Rank of 32x32 Binary Matrices Test	Not applicable	Not applicable	Not applicable	OK (0% errors)
Rank of 6x8 Binary Matrices Test	Not applicable	Not applicable	Not applicable	OK (0% errors)
Counts-the-1's Test (stream of bits)	Not applicable	Not applicable	Not applicable	OK (30% errors)

Counts-the-1's Test (stream of specific bytes)	Not applicable	Not applicable	Not applicable	OK (0% errors)
Craps Test	OK (20% errors)	OK (20% errors)	OK (20% errors)	OK (20% errors)
Parking Lot Test	OK (0% errors)	OK (0% errors)	Not applicable	OK (0% errors)
2D Self-Avoiding Random Walk Test	FAIL (70% errors)	FAIL (80% errors)	Not applicable	FAIL (80% errors)

NOTE

1. The tabulated data is obtained using the one-level (threshold) testing technique. The OK result indicates FAIL < 50%. The run fails when p-value falls outside the interval [0.05, 0.95].
2. The stream tested is generated by calling the function `vslNewStream` with seed=7,777,777.

6.4.3. MRG32k3a

This is a 32-bit combined multiple recursive generator with two components of order 3:

$$x_n = a_{11}x_{n-1} + a_{12}x_{n-2} + a_{13}x_{n-3} (\mathrm{mod}\, m_1)$$

$$y_n = a_{21}y_{n-1} + a_{22}y_{n-2} + a_{23}y_{n-3} (\mathrm{mod}\, m_2)$$

$$z_n = x_n - y_n (\mathrm{mod}\, m_1)$$

$$u_n = z_n / m_1$$

$$a_{11} = 0,\ a_{12} = 1403580,\ a_{13} = -810728,\ m_1 = 2^{32} - 209$$

$$a_{21} = 527612,\ a_{22} = 0,\ a_{23} = -1370589,\ m_2 = 2^{32} - 22853$$

MRG32k3a combined generator meets the requirements for modern RNGs, such as good multidimensional uniformity, or a long period. Optimization for various Intel® architectures makes it competitive with the other VS BRNGSs in terms of speed.

Real Implementation (Single and Double Precision)

The output vector is the sequence of the floating-point values u_0, u_1, ...

Integer Implementation

The output vector of 32-bit integers z_0, z_1, ...

Stream Initialization by Function `vslNewStream`

MRG32k3a generates the stream and initializes it specifying the 32-bit input integer parameter seed. The stream state is the two triplets of 32-bit integers ($x_{-1}, x_{-2}, ..., x_{-3}$ and $y_{-1}, y_{-2}, ..., y_{-3}$), initialized in the following way:

1. Assume x_{-3} = seed `mod` m_1.
2. Assume the other values equal to 1, that is, $x_{-2} = x_{-1} = y_{-3} = y_{-2} = y_{-1} = 1$.

Stream Initialization of Function `vslNewStreamEx`

MRG32k3a generates the stream and initializes it specifying the array n of 32-bit integer `params[]`:

1. If $n = 0$, assume $x_{-3} = x_{-2} = x_{-1} = y_{-3} = y_{-2} = y_{-1} = 1$.
2. If $n = 1$, assume x_{-3} = `params[0] mod m`$_1$, $x_{-2} = x_{-1} = y_{-3} = y_{-2} = y_{-1} = 1$.
3. If $n = 2$, assume x_{-3} = `params[0] mod m`$_1$, x_{-2} = `params[1] mod m`$_1$, $x_{-1} = y_{-3} = y_{-2} = y_{-1} = 1$.
4. If $n = 3$, assume x_{-3} = `params[0] mod m`$_1$, x_{-2} = `params[1] mod m`$_1$, x_{-1} = `params[2] mod m`$_1$, $y_{-3} = y_{-2} = y_{-1} = 1$.
   If the values prove to be $x_{-3} = x_{-2} = x_{-1} = 0$, assume $x_{-3} = 1$.
5. If $n = 4$, assume x_{-3} = `params[0] mod m`$_1$,x_{-2} = `params[1] mod m`$_1$, x_{-1} = `params[2] mod m`$_1$, y_{-3} = `params[3] mod m`$_2$, $y_{-2} = y_{-1} = 1$.
   If the values prove to be $x_{-3} = x_{-2} = x_{-1} = 0$, assume $x_{-3} = 1$.
6. If $n = 5$, assume x_{-3} = `params[0] mod m`$_1$, x_{-2} = `params[1] mod m`$_1$, x_{-1} = `params[2] mod m`$_1$, y_{-3} = `params[3] mod m`$_2$, y_{-2} = `params[4] mod m`$_2$, $y_{-1} = 1$.
   If the values prove to be $x_{-3} = x_{-2} = x_{-1} = 0$, assume $x_{-3} = 1$.
7. If $n \ge 6$, assume x_{-3} = `params[0] mod m`$_1$, x_{-2} = `params[1] mod m`$_1$, x_{-1} = `params[2] mod m`$_1$, y_{-3} = `params[3] mod m`$_2$, y_{-2} = `params[4] mod m`$_2$, y_{-1} = `params[5] mod m`$_2$.
   If the values prove to be $x_{-3} = x_{-2} = x_{-1} = 0$, assume $x_{-3} = 1$.
   If the values prove to be $y_{-3} = y_{-2} = y_{-1} = 0$, assume $y_{-3} = 1$.

Subsequences Selection Methods

`vslSkipAheadStream`	Supported
`vslLeapfrogStream`	Not supported

Generator Period

$$\rho \approx 2^{191} \approx 3.1 \times 10^{57}$$

Lattice Structure

$M_8 = 0.68561$, $M_{16} = 0.63940$, $M_{32} = 0.63359$.

Empirical Testing Results Summary

NOTE

1. The tabulated data is obtained using the one-level (threshold) testing technique. The OK result indicates FAIL < 50%. The run fails when p-value falls outside the interval [0.05, 0.95].
2. The stream tested is generated by calling the function `vslNewStream` with seed=7,777,777.

6.4.4. MCG59

This is a 59-bit multiplicative congruential generator:

$$x_n = ax_{n-1} (\bmod m)$$
$$u_n = x_n / m$$
$$a = 13^{13}, m = 2^{59}$$

Multiplicative congruential generator MCG59 is one of the two basic generators implemented in the NAG Numerical Libraries. As the module of the generator is not prime, the length of its period is not 2^{59} but only 2^{57}, if the initial value (seed) is not an even number. The drawback of these generators is well known, (see [Cram46], [Ent98]):

1. The lower bits of the generated sequence of pseudo-random numbers are not random and thus breaking numbers down into their bit patterns and using individual bits may cause failure of random tests.
2. Block-splitting an entire period sequence into 2d identical blocks leads to their full identity in d lower bits.

Real Implementation (Single and Double Precision)

The output vector is the sequence of the floating-point values u_0, u_1, ...

Integer Implementation

The output vector of the 32-bit integers is $x_0 \bmod 2^{32}$, $\lfloor x_0/2^{32} \rfloor$, $x_1 \bmod 2^{32}$, $\lfloor x_1/2^{32} \rfloor$, ...

Thus, the output vector stores practically every 59-bit member of the integer output as two 32-bit integers. For example, to get a vector from *n* 59-bit integers the size of the output array should be large enough to store 2n 32-bit numbers.

Stream Initialization by Function `vslNewStream`

MCG59 generates the stream and initializes it specifying the 32-bit input integer parameter seed.

1. Assume x_0 = seed mod 2^{59}.
2. If $x_0 = 0$, assume $x_0 = 1$.

Stream Initialization of Function `vslNewStreamEx`

MCG59 generates the stream and initializes it specifying the the array `params[]` of *n* 32-bit integers:

1. If $n = 0$, assume $x_0 = 1$.
2. If $n = 1$, assume `seed = params[0]`, follow the instructions described in the above section on stream initialization by the function `vslNewStream`.
3. Otherwise assume `seed = params[0]+2`^{32}`*params[1]`, follow the instructions described in the above section on stream initialization by the function `vslNewStream`.

Subsequences Selection Methods

`vslSkipAheadStream`	Supported
`vslLeapfrogStream`	Supported

Generator Period

$\rho \approx 2^{57} \approx 1.4 \times 10^{17}$

Lattice Structure

$S_2 = 0.84$; $S_3 = 0.73$; $S_4 = 0.74$; $S_5 = 0.58$; $S_6 = 0.63$; $S_7 = 0.52$; $S_8 = 0.55$; $S_9 = 0.56$.

Empirical Testing Results Summary

Test Name	**`vsRngUniform`**	**`vdRngUniform`**	**`viRngUniform`**	**`viRngUniformBits`**
3D Spheres Test	OK (10% errors)	OK (10% errors)	Not applicable	OK (10% errors)
Birthday Spacing Test	Not applicable	Not applicable	Not applicable	OK (0% errors)[1]
Bitstream Test	Not applicable	Not applicable	Not applicable	OK (45% errors)
Rank of 31x31 Binary Matrices Test	Not applicable	Not applicable	Not applicable	OK (0% errors)[2]
Rank of 32x32 Binary Matrices Test	Not applicable	Not applicable	Not applicable	OK (0% errors)[3]
Rank of 6x8 Binary Matrices Test	Not applicable	Not applicable	Not applicable	OK (0% errors)[4]
Counts-the-1's Test (stream of bits)	Not applicable	Not applicable	Not applicable	FAIL (100% errors)
Counts-the-1's Test (stream of specific bytes)	Not applicable	Not applicable	Not applicable	OK (0% errors)[5]
Craps Test	OK (10% errors)	OK (10% errors)	OK (10% errors)	OK (10% errors)
Parking Lot	OK (20% errors)	OK (20% errors)	Not applicable	OK (20% errors)

Test				
2D Self-Avoiding Random Walk Test	OK (20% errors)	OK (10% errors)	Not applicable	OK (10% errors)

NOTE

1. The tabulated data is obtained using the one-level (threshold) testing technique. The OK result indicates FAIL < 50%. The run fails when p-value falls outside the interval [0.05, 0.95].
2. The stream tested is generated by calling the function `vslNewStream` with seed=7,777,777.

[1] The generator fails the test for bit groups 0-23, 1-24, 2-25, 3-26, 5-28.

[2] The generator fails the test for bit groups 0-30, 1-31.

[3] The generator fails the test for bit groups 0-31, 1-32.

[4] The generator fails the test for bit groups 0-7, ..., 9-16, 11-18, 32-39, ..., 37-44, 39-46, ..., 41-48.

[5] The generator fails the test for bit groups 0-7, ..., 11-18, 13-20, ..., 15-22.

6.4.5. WH

This is a set of 273 Wichmann-Hill's combined multiplicative congruential generators (j = 1, 2, ..., 273):

$$x_n = a_{1,j} x_{n-1} (\bmod m_{1,j})$$

$$y_n = a_{2,j} y_{n-1} (\bmod m_{2,j})$$

$$z_n = a_{3,j} z_{n-1} (\bmod m_{3,j})$$

$$w_n = a_{4,j} w_{n-1} (\bmod m_{4,j})$$

$$u_n = \left(x_n / m_{1,j} + y_n / m_{2,j} + z_n / m_{3,j} + w_n / m_{4,j} \right) \bmod 1$$

WH is a set of 273 different basic generators. This generator is the second basic generator in the NAG libraries. The constants $a_{i,j}$ range from 112 to 127, the constants $m_{i,j}$ are prime numbers ranging from 16,718,909 to 16,776,971, close to 2^{24}. These constants should show good results in the spectral test (see Knuth [Knuth81] and MacLaren [MacLaren89]). The period of each Wichmann-Hill generator may be equal to 2^{92} if not for common factors between ($m_{1,j}$-1), ($m_{2,j}$-1), ($m_{3,j}$-1) and ($m_{4,j}$-1). However, each generator should still have a period of at least 2^{80}. The generated pseudo-random sequences are essentially independent of one another according to the spectral test (for detailed information about properties of these generators see [MacLaren89]).

Real Implementation (Single and Double Precision)

The output vector is the sequence of the floating-point values u_0, u_1, ...

Integer Implementation

The output vector of 32-bit integers x_0, y_0, z_0, w_0, x_1, y_1, z_1, w_1

Thus, the output vector stores practically every quadruple (*x, y, z, w*) of members of the integer output as four 32-bit integers. For example, to get a vector from n quadruples (*x, y, z, w*), the size of the output array should be large enough to store 4n 32-bit numbers.

Stream Initialization by Function `vslNewStream`

WH generates the stream and initializes it specifying the 32-bit input integer parameter seed :

1. Assume x_0 = `seed mod m₁`. If $x_0 = 0$, assume $x_0 = 1$.
2. Assume $y_0 = 1$, $z_0 = 1$, $w_0 = 1$.

The test selects a WH generator adding an offset to the named constant `VSL_BRNG_WH: VSL_BRNG_WH+0, VSL_BRNG_WH+1, ... , VSL_BRNG_WH+272`. The following example illustrates the initialization of the seventh (of 273) WH generator:

```
vslNewStream (&stream, VSL_BRNG_WH+6, seed);
```

Stream Initialization of Function `vslNewStreamEx`

WH generates the stream and initializes it specifying the the array `params[]` of *n* 32-bit integers:

1. If $n = 0$, assume $x_0 = 1$, $y_0 = 1$, $z_0 = 1$, $w_0 = 1$.
2. If $n = 1$, assume x_0 = `params[0] mod m₁`, $y_0 = 1$, $z_0 = 1$, $w_0 = 1$.
   If $x_0 = 0$, assume $x_0 =1$.
3. If $n = 2$, assume x_0 = `params[0] mod m₁`, y_0 = `params[1] mod m₂`, $z_0 = 1$, $w_0 = 1$. If $x_0 = 0$, assume $x_0 = 1$.
   If $y_0 = 0$, assume $y_0 = 1$.
4. If $n = 3$, assume x_0 = `params[0] mod m₁`, y_0 = `params[1] mod m₂`, z_0 = `params[2] mod m₃`, $w_0 = 1$.
   If $x_0 = 0$, assume $x_0 = 1$.
   If $y_0 = 0$, assume $y_0 = 1$.
   If $z_0 = 0$, assume $z_0 = 1$.
5. If $n \geq 4$, assume x_0 = `params[0] mod m₁`, y_0 = `params[1] mod m₂`, z_0= `params[2] mod m₃`, w_0 = `params[3] mod m₄`.
   If $x0 = 0$, assume $x_0 = 1$.
   If $y_0 = 0$, assume $y_0 = 1$.
   If $z_0 = 0$, assume $z_0 = 1$.
   If $w_0 = 0$, assume $w_0 = 1$.

Subsequences Selection Methods

`vslSkipAheadStream`	Supported
`vslLeapfrogStream`	Supported

Generator Period

$$\rho \geq 2^{80} \approx 1.2 \times 10^{24}$$

Empirical Testing Results Summary

Test Name	`vsRngUniform`	`vdRngUniform`	`viRngUniform`	`viRngUniformBits`
3D Spheres Test	OK (0% errors)	OK (0% errors)	Not applicable	OK (0% errors)
Birthday Spacing Test	Not applicable	Not applicable	Not applicable	FAIL (60% errors)
Bitstream Test	Not applicable	Not applicable	Not applicable	OK (10% errors)
Rank of 31x31 Binary Matrices Test	Not applicable	Not applicable	Not applicable	N/A
Rank of 32x32 Binary Matrices Test	Not applicable	Not applicable	Not applicable	N/A
Rank of 6x8 Binary Matrices Test	Not applicable	Not applicable	Not applicable	OK (0% errors)[6]
Counts-the-1's Test (stream of bits)	Not applicable	Not applicable	Not applicable	OK (10% errors)
Counts-the-1's Test (stream of specific bytes)	Not applicable	Not applicable	Not applicable	OK (0% errors)
Craps Test	OK (20% errors)	OK (20% errors)	OK (20% errors)	OK (10% errors)
Parking Lot Test	OK (10% errors)	OK (10% errors)	Not applicable	OK (10% errors)
2D Self-Avoiding Random Walk Test	OK (10% errors)	OK (0% errors)	Not applicable	OK (20% errors)

NOTE

1. The tabulated data is obtained using the one-level (threshold) testing technique. The OK result indicates FAIL < 50%. The run fails when p-value falls outside the interval [0.05, 0.95].
2. The stream tested is generated by calling the function `vslNewStream` with seed=7,777,777.

6.4.6. MT19937

This is a Mersenne Twister pseudorandom number generator:

$$x_n = x_{n-(624-397)} \oplus ((x_{n-624} \& 0x80000000) | (x_{n-624+1} \& 0x7FFFFFFF))A,$$

$$y_n = x_n,$$

$$y_n = y_n \oplus (y_n >> 11),$$

$$y_n = y_n \oplus ((y_n << 7) \& 0x9D2C5680),$$

$$y_n = y_n \oplus ((y_n << 15) \& 0xEFC60000),$$

$$y_n = y_n \oplus (y_n >> 18),$$

$$u_n = y_n / 2^{32}.$$

Matrix A (32x32) has the following format:

$$A = \left|\begin{array}{ccccc} 0 & 1 & 0 & & \\ 0 & 0 & \cdots & 0 & \\ & & & \cdots & \\ & & 0 & 0 & 1 \\ a_{31} & a_{30} & \cdots & \cdots & a_0 \end{array}\right|,$$

where the 32-bit vector $a = a_{31} \ldots a_0$ has the value a = 0x9908B0DF.

Mersenne Twister pseudorandom number generator MT19937 is a modification of twisted generalized feedback shift register generator [Matsum92], [Matsum94]. MT19937 has the period length of 2^{19937}-1 and is 623-dimensionally equidistributed with up to 32-bit accuracy. These properties make the generator applicable for simulations in various fields of science and engineering. The initialization procedure is essentially the same as described in [MT2002]. The state of the generator is represented by 624 32-bit unsigned integer numbers.

Real Implementation (Single and Double Precision)

The output vector is the sequence of the floating-point values u_0, u_1, ...

Integer Implementation

The output vector of 32-bit integers y_0, y_1, ...

Stream Initialization by Function `vslNewStream`

MT19937 generates the stream and initializes it specifying the input 32-bit unsigned integer parameter seed. The stream state, that is, the array of 624 32-bit integers x_0, ..., x_{623}, is initialized by the procedure described in [MT2002] and based on the seed value.

Stream Initialization of Function `vslNewStreamEx`

MT19937 generates the stream and initializes it specifying the array the array `params[]` of *n* 32-bit integers:

1. If *n* ≥ 1, perform initialization as described in [MT2002] using array `params[]` on input.
2. If *n* = 0, assume params[0] = 1, *n* = 1 and perform initialization as described in the previous item.

Subsequences Selection Methods

`vslSkipAheadStream`	Supported
`vslLeapfrogStream`	Not supported

Skip-ahead method supported by MT19937 is based on algorithms described in [Haram08].

Generator Period

$$\rho = 2^{19937} - 1 \approx 4.3 \times 10^{6001}$$

Empirical Testing Results Summary

Test Name	**`vsRngUniform`**	**`vdRngUniform`**	**`viRngUniform`**	**`viRngUniformBits`**
3D Spheres Test	OK (0% errors)	OK (0% errors)	Not applicable	OK (0% errors)
Birthday Spacing Test	Not applicable	Not applicable	Not applicable	OK (10% errors)
Bitstream Test	Not applicable	Not applicable	Not applicable	OK (10% errors)
Rank of 31x31 Binary Matrices Test	Not applicable	Not applicable	Not applicable	OK (10% errors)
Rank of 32x32 Binary Matrices Test	Not applicable	Not applicable	Not applicable	OK (0% errors)
Rank of 6x8 Binary Matrices Test	Not applicable	Not applicable	Not applicable	OK (0% errors)
Counts-the-1's Test (stream of bits)	Not applicable	Not applicable	Not applicable	OK (20% errors)
Counts-the-1's Test (stream of specific bytes)	Not applicable	Not applicable	Not applicable	OK (0% errors)

Craps Test	OK (30% errors)	OK (30% errors)	OK (30% errors)	OK (30% errors)
Parking Lot Test	OK (0% errors)	OK (0% errors)	Not applicable	OK (0% errors)
2D Self-Avoiding Random Walk Test	OK (0% errors)	OK (10% errors)	Not applicable	OK (10% errors)

NOTE

1. The tabulated data is obtained using the one-level (threshold) testing technique. The OK result indicates FAIL < 50%. The run fails when p-value falls outside the interval [0.05, 0.95].
2. The stream tested is generated by calling the function `vslNewStream` with seed=7,777,777.

6.4.7. SFMT19937

This is a SIMD-oriented Fast Mersenne Twister pseudorandom number generator:

$$w_n = w_0 A \oplus w_M B \oplus w_{n-2} C \oplus w_{n-1} D$$

where w_0, w_M, w_{n-2}, ... are 128-bit integers, and the wA, wB, wC, wD operations are defined as follows:

$wA := (w \overset{128}{<<} 8) \oplus w$, left shift of 128-bit integer *w* by *a* followed by exclusive-or operation

$wB := (w \overset{32}{>>} 11, 32) \& mask$, right shift of each 32-bit integer in quadruple *w* followed by and-operation with quadruple of 32-bit masks *mask*,

mask=(0xBFFFFFF6 0xBFFAFFFF 0xDDFECB7F 0xDFFFFFFEF)

$wC := (w \overset{128}{>>} 8)$, right shift of 128-bit integer *w*

$wD := (w \overset{32}{<<} 18, 32) \& mask$, left shift of each 32-bit integer in quadruple *w*

Integer output: $r_{4n+k} = w_n(k)$, where $w_n(k)$ is the *k*-th 32-bit integer in quadruple w_n, $k = 0, 1, 2, 3$

$u_n = (int)r_n/2^{32} + ½$

Real Implementation (Single and Double Precision)

The output vector is the sequence of the floating-point values u_0, u_1, ...

Integer Implementation

The output vector of 32-bit integers r_0, r_1...

Stream Initialization by Function `vslNewStream`

SFMT19937 generates the stream and initializes it specifying the input 32-bit unsigned integer parameter seed. The stream state, that is, the array of 156 128-bit integers (624 32-bit integers x_0, ..., x_{623}), is initialized by the procedure described in [Saito08] and based on the seed value.

Stream Initialization of Function `vslNewStreamEx`

SFMT19937 generates the stream and initializes it specifying the array `params[]` of *n* 32-bit unsigned integers:

1. If $n \geq 1$, perform initialization as described [Saito08] using array `params[]` on input.
2. If $n = 0$, assume params[0] = 1, $n = 1$ and perform initialization as described in the previous item.

Subsequences Selection Methods

`vslSkipAheadStream`	Supported
`vslLeapfrogStream`	Not supported

Skip-ahead method supported by SFMT19937 is based on algorithms described in [Haram08].

Generator Period

$$\rho = 2^{19937} - 1 \approx 4.3 \times 10^{6001}$$

Empirical Testing Results Summary

Test Name	**`vsRngUniform`**	**`vdRngUniform`**	**`viRngUniform`**	**`viRngUniformBits`**
3D Spheres Test	OK (30% errors)	OK (30% errors)	Not applicable	OK (40% errors)
Birthday Spacing Test	Not applicable	Not applicable	Not applicable	OK (0% errors)
Bitstream Test	Not applicable	Not applicable	Not applicable	OK (10% errors)
Rank of 31x31 Binary Matrices Test	Not applicable	Not applicable	Not applicable	OK (0% errors)
Rank of 32x32 Binary Matrices Test	Not applicable	Not applicable	Not applicable	OK (0% errors)
Rank of 6x8 Binary Matrices	Not applicable	Not applicable	Not applicable	OK (0% errors)

Test				
Counts-the-1's Test (stream of bits)	Not applicable	Not applicable	Not applicable	OK (10% errors)
Counts-the-1's Test (stream of specific bytes)	Not applicable	Not applicable	Not applicable	OK (0% errors)
Craps Test	OK (20% errors)	OK (20% errors)	OK (20% errors)	OK (10% errors)
Parking Lot Test	OK (30% errors)	OK (30% errors)	Not applicable	OK (0% errors)
2D Self-Avoiding Random Walk Test	OK (0% errors)	OK (20% errors)	Not applicable	OK (10% errors)

NOTE

1. The tabulated data is obtained using the one-level (threshold) testing technique. The OK result indicates FAIL < 50%. The run fails when p-value falls outside the interval [0.05, 0.95].
2. The stream tested is generated by calling the function `vslNewStream` with seed=7,777,777.

6.4.8. MT2203

This is a set of 6024 Mersenne Twister pseudorandom number generators (*j* = 1, ..., 6024):

$$x_{n,j} = x_{n-(69-34),j} \oplus ((x_{n-69,j} \,\&\, 0\text{xFFFFFFE0}) \,|\, (x_{n-69+1,j} \,\&\, 0\text{x1F}))A_j,$$

$$y_{n,j} = x_{n,j},$$

$$y_{n,j} = y_{n,j} \oplus (y_{n,j} >> 12),$$

$$y_{n,j} = y_{n,j} \oplus ((y_{n,j} << 7) \,\&\, b_j),$$

$$y_{n,j} = y_{n,j} \oplus ((y_{n,j} << 15) \,\&\, c_j),$$

$$y_{n,j} = y_{n,j} \oplus (y_{n,j} >> 18),$$

$$u_n = y_{n,j} / 2^{32}.$$

Matrix A_j(32x32) has the following format:

$$A_j = \begin{vmatrix} 0 & 1 & 0 & & \\ 0 & 0 & \dots & 0 & \\ & & & \dots & \\ & & 0 & 0 & 1 \\ a_{31,j} & a_{30,j} & \dots & \dots & a_{0,j} \end{vmatrix},$$

with the 32-bit vector $a_j = a_{31,j} \dots a_{0,j}$.

The set of 6024 basic pseudorandom number generators MT2203 is a natural addition to MT19937 generator. MT2203 generators are intended for use in large scale Monte Carlo simulations performed on multi-processor computer systems. These generators possess a smaller period length but the number of 2^{2203} to the method described in [Matsum2000].

Real Implementation (Single and Double Precision)

The output vector is the sequence of the floating-point values u_0, u_1, ...

Integer Implementation

The output vector of 32-bit integers $y_{0,j}$, $y_{1,j}$, ...

Stream Initialization by Function `vslNewStream`

MT2203 generates the stream and initializes it specifying the input 32-bit unsigned integer parameter seed. The stream state, that is, the array of 69 32-bit integers x_0, ..., x_{68}, is initialized by the procedure described in [MT2002] and based on the seed value.

MT2203 generator is a set of 6024 basic generators. To select an MT2203 generator, add an offset to the named constant `VSL_BRNG_MT2203`, for example, `VSL_BRNG_MT2203+0, VSL_BRNG_ MT2203+1, ....` The following example illustrates the initialization of the 10th (of 6024) MT2203 generator:

```
vslNewStream (&stream, VSL_BRNG_MT2203+9, seed);
```

Stream Initialization of Function `vslNewStreamEx`

MT2203 generates the stream and initializes it specifying the array `params[]` of *n* 32-bit unsigned integers:

1. If *n* ≥ 1, perform initialization as described in [MT2002] using array `params[]` on input.
2. If *n* = 0, assume params[0] = 1, *n* = 1 and perform initialization as described in the previous item.

Subsequences Selection Methods

`vslSkipAheadStream`	Not supported
`vslLeapfrogStream`	Not supported

Generator Period

$$\rho = 2^{2203} - 1 \approx 1.48 \times 10^{663}$$

Empirical Testing Results Summary

Test Name	`vsRngUniform`	`vdRngUniform`	`viRngUniform`	`viRngUniformBits`

3D Spheres Test	OK (20% errors)	OK (20% errors)	Not applicable	OK (20% errors)
Birthday Spacing Test	Not applicable	Not applicable	Not applicable	OK (0% errors)
Bitstream Test	Not applicable	Not applicable	Not applicable	OK (15% errors)
Rank of 31x31 Binary Matrices Test	Not applicable	Not applicable	Not applicable	OK (10% errors)
Rank of 32x32 Binary Matrices Test	Not applicable	Not applicable	Not applicable	OK (0% errors)
Rank of 6x8 Binary Matrices Test	Not applicable	Not applicable	Not applicable	OK (0% errors)
Counts-the-1's Test (stream of bits)	Not applicable	Not applicable	Not applicable	OK (0% errors)
Counts-the-1's Test (stream of specific bytes)	Not applicable	Not applicable	Not applicable	OK (0% errors)
Craps Test	OK (20% errors)	OK (20% errors)	OK (20% errors)	OK (20% errors)
Parking Lot Test	OK (0% errors)	OK (0% errors)	Not applicable	OK (0% errors)
2D Self-Avoiding Random Walk Test	OK (10% errors)	OK (0% errors)	Not applicable	OK (0% errors)

NOTE

1. The tabulated data is obtained using the one-level (threshold) testing technique. The OK result indicates FAIL < 50%. The run fails when p-value falls outside the interval [0.05, 0.95].

2. The stream tested is generated by calling the function `vslNewStream` with seed=7,777,777.

6.4.9. SOBOL

This is a 32-bit Gray code-based quasi-random number generator:

$$\mathbf{x}_n = \mathbf{x}_{n-1} \oplus \mathbf{v}_c$$

$$\mathbf{u}_n = \mathbf{x}_n / 2^{32}$$

NOTE

The value *c* is the rightmost zero bit in n-1; x_n is s-dimensional vector of 32-bit values. The s-dimensional vectors (calculated during random stream initialization) v_i, $i = 1,32$ are called direction numbers. The vector u_n is the generator output normalized to the unit hypercube $(0,1)^s$.

Bratley and Fox [Brat87] provide an implementation of the SOBOL quasi-random number generator. VS implementation allows generating SOBOL's low-discrepancy sequences with the length of up to 2^{32}. This implementation also accepts registration of user-defined parameters (direction numbers and primitive polynomials) during the initialization, which permits obtaining quasi-random vectors of any dimension. If you do not supply user-defined parameters, the default values are used for generation of quasi-random vectors. The default dimension of quasi-random vectors can vary from 1 to 40 inclusive.

Real Implementation (Single and Double Precision)

The output vector is the sequence of the floating-point values $u_1, u_2, \ldots$, where elements $u_1, u_2, \ldots, u_s$ correspond to the $\boldsymbol{u_1}$, $u_{s+11}, u_{s+2}, \ldots, u_{2s}$ correspond to the $\boldsymbol{u_2}$, and so on.

Integer Implementation

The output vector of 32-bit integers $x_1, x_2, \ldots$, where elements $x_1, x_2, \ldots, x_s$ correspond to the $\boldsymbol{x_1}$, $x_{s+11}, x_{s+2}, \ldots, x_{2s}$ correspond to the $\boldsymbol{x_2}$, and so on.

Stream Initialization by Function `vslNewStream`

SOBOL generates the stream and initializes it specifying the input 32-bit parameter seed (dimension dimen of a quasi-random vector):

1. Assume `dimen` = seed
2. If `dimen` < 1 or `dimen` > 40, assume `dimen` = 1.

Stream Initialization by Function `vslNewStreamEx`

SOBOL generates the stream and initializes it, specifying the array `params[]` of *n* 32-bit integers to set the dimension dimen of a quasi-random vector as well as pass other generator related parameters. For example, initial direction numbers and primitive polynomials. Direction numbers can also be passed using the array. The general interface for passing stream initialization parameters of SOBOL via the `params[]` array has the following format:

Position in `params[]`	0	1	2	`3...2+dimen`	`3+dimen`	`4+dimen...dimen*(maxdeg+1)+3`
	`dimen`	Parameter Class Indicators	Initial Values Subclass Indicators	Primitive polynomials	Maximum degree of primitive polynomial, maxdeg	Initial direction numbers

The dimension parameter params[0] is mandatory, and can be initialized as follows:

```
params[0] = dimen;
```

The other elements of params intended for passing additional user-supplied data are optional. For example, if they are not presented, then the default tables of direction numbers are used for generation of quasi-random vectors. VS default tables of direction numbers allow generating quasi-random sequences for dimensions up to 40.

If you want to generate quasi-random vectors of greater dimension or obtain another sequence you may register a set of your own primitive polynomials and/or a table of initial direction numbers. In order to do this, you need to set the Parameter Class Indicators field (`params[1]`) to `VSL_USER_QRNG_INITIAL_VALUES`:

```
params[1] = VSL_USER_QRNG_INITIAL_VALUES;
```

Further, you should specify in Initial Values Subclass Indicators field (`params[2]`) whether you want to supply primitive polynomials, initial direction numbers, or both, by setting corresponding indicators. In the example below both direction numbers and primitive polynomials indicators are set:

```
params[2] = VSL_USER_INIT_DIRECTION_NUMBERS | VSL_USER_PRIMITIVE_POLYMS;
```

If you want to provide just initial direction numbers, do it as follows:

```
params[2] = VSL_USER_INIT_DIRECTION_NUMBERS;
```

Similarly, you can indicate that only primitive polynomials are passed to the library:

```
params[2] = VSL_USER_PRIMITIVE_POLYMS;
```

NOTE

For dimensions greater than 40, both the primitive polynomials and the table of initial direction numbers must be provided.

The remainder of the params array is used to pass primitive polynomials and/or initial direction numbers. Primitive polynomials are packed as unsigned integers, initial direction numbers for SOBOL are assumed to be a two-dimensional table. In the matrix i-th row corresponds to i-th dimension, and number of columns equals the maximum degree of primitive polynomials maxdeg. The number of polynomials (and the number of rows in the table) depends on the initialization mode for the first dimension. In the default initialization mode (see [Brat88] for details) it is enough to pass into the library `dimen -1` primitive polynomials (correspondingly, the number of rows in the table of initial direction numbers also equals `dimen -1`). To override default initialization for the first dimension, `set VSL_QRNG_OVERRIDE_1ST_DIM_INIT` indicator in `params[2]`:

```
params[2] = params[2] | VSL_QRNG_OVERRIDE_1ST_DIM_INIT;
```

and pass a complete set of polynomials and/or initial direction numbers (dimen primitive polynomials and the table of initial direction numbers with dimen rows). If you pass just primitive polynomials or initial direction numbers for dimensions $1 \le s \le 40$, the default initialization for the first dimension is always assumed (the number of polynomials and the number of rows in the table of initial direction numbers equals s-1).

If both arrays are passed to the generator you should organize data in correct order: first - polynomials, second - maximum degree of primitive polynomials and, finally, initial direction numbers as shown in the example below:

```
unsigned int uSobolIrredPoly[dimen] = {...};
unsigned int uSobolMInit[dimen][maxdeg] = {...};
...
params[0] = dimen;
params[1] = VSL_USER_QRNG_INITIAL_VALUES;
params[2] = VSL_USER_INIT_DIRECTION_NUMBERS|VSL_USER_PRIMITIVE_POLYMS;
params[2] = params[2] | VSL_QRNG_OVERRIDE_1ST_DIM_INIT;

for ( i = 0; i < dimen; i++ ) params[i+3] = uSobolIrredPoly[i];
params[3+dimen] = maxdeg;
```

```
k = 4+dimen;
for ( i = 0; i < dimen; i++ )
{
     for ( j = 0; j < maxdeg; j++ )
     {
          params[k++] = uSobolMInit[i][j];
     }
}
```

The following example illustrates replacement of the default initial values for SOBOL with user-provided values:

```
...
// dimen = 10
unsigned int uSobolMInit[dimen-1][maxdeg] = {...};
params[0] = dimen;
params[1] = VSL_USER_QRNG_INITIAL_VALUES;
params[2] = VSL_USER_INIT_DIRECTION_NUMBERS;
params[3] = maxdeg;
k = 4;
for ( i = 0; i < dimen-1; i++ )
{
     for ( j = 0; j < maxdeg; j++ )
     {
          params[k++] = uSobolMInit[i][j];
     }
       }
```

You can also calculate a table of direction numbers using your own initial direction numbers and primitive polynomials and pass this array to the generator. The interface for registration of the direction numbers is as follows:

Position in `params[]`	0	1	2	`3...dimen*32+2`
	`dimen`	Parameter Class Indicators	Initial Values Subclass Indicators	Direction numbers

As earlier, the dimension parameter params[0] and Parameter Class Indicators field (params[1]) can be initialized as follows:

```
params[0] = dimen;
params[1] = VSL_USER_QRNG_INITIAL_VALUES;
Further, you need to initialize Initial Values Subclass Indicators field
(params[2]):
params[2] = VSL_USER_DIRECTION_NUMBERS;
```

Direction numbers are assumed to be a dimen x 32 table of unsigned integers and can be passed to the generator in the following way:

```
unsigned int uSobolV[dimen][32] = {...};
params[0] = dimen;
params[1] = VSL_USER_QRNG_INITIAL_VALUES;
params[2] = VSL_USER_DIRECTION_NUMBERS;
k = 3;
for ( i = 0; i < dimen; i++ )
{
     for ( j = 0; j < 32; j++ )
     {
          params[k++] = uSobolV[i][j];
     }
}
```

In short, the SOBOL stream initialization is as follows:

1. If *n* = 0, assume `dimen` = 1 and initialize the stream using the default table of primitive polynomials and initial direction numbers.
2. If *n* = 1, `dimen = params[0]`, initialize the stream using the default table of primitive polynomials and initial direction numbers; (if dimen < 1 or dimen > 40, assume dimen = 1).
3. If *n* > 1, dimen = params[0]
   a. If dimen < 1, assume dimen = 1 and initialize the stream using the default table of primitive polynomials and initial direction numbers.
   b. If the externally defined parameters of the generator are packed incorrectly, initialize the stream using the default table of primitive polynomials and initial direction numbers; (if dimen > 40, assume dimen = 1).
   c. Initialize the SOBOL quasi-random stream by means of the user-defined primitive polynomials and initial direction numbers or direction numbers.

Subsequences Selection Methods

`vslSkipAheadStream`	supported
`vslLeapfrogStream`	supported

NOTE

1. The skip-ahead method skips individual components of quasi-random vectors rather than whole s-dimensional vectors. Hence, to skip N s-dimensional quasi-random vectors, call `vslSkipAheadStream` subroutine with parameter `nskip` equal to the Nxs.
2. The leapfrog method works with individual components of quasi-random vectors rather than with s-dimensional vectors. In addition, its functionality allows picking out a fixed quasi-random component only. In other words, nstreams parameter should be equal to the predefined constant `VSL_QRNG_LEAPFROG_COMPONENTS`, and *k* parameter should indicate the index of a component of s-dimensional quasi-random vectors to be picked out (0 ≤ *k* < *s*).

Generator Period

$$\rho = 2^{32} \approx 4.2 \times 10^{9}$$

Dimensions

$1 \le s \le 40$ is the default set of dimensions; user-defined dimensions are accepted.

6.4.10. NIEDERREITER

This is a 32-bit Gray code-based quasi-random number generator:

$$\mathbf{x}_n = \mathbf{x}_{n-1} \oplus \mathbf{v}_c$$

$$\mathbf{u}_n = \mathbf{x}_n / 2^{32}$$

NOTE

The value *c* is the right-most zero bit in n-1; x_n is s-dimensional vector of 32-bit values. The s-dimensional vectors (calculated during random stream initialization) v_i, $i = 1,32$ are called direction numbers. The vector $u_n$is the generator output normalized to the unit hypercube $(0,1)^s$.

According to the results of Bratley, Fox, and Niederreiter [Brat92] Niederreiter sequences have the best known theoretical asymptotic properties. VS implementation allows generating Niederreiter low-discrepancy sequences of length up to 2^{32}. This implementation also allows for registration of user-defined parameters (irreducible polynomials or direction numbers), which allows obtaining quasi-random vectors of any dimension. If you do not supply user-defined parameters, the default values are used for generation of quasi-random vectors. The default dimension of quasi-random vectors can vary from 1 to 318 inclusive.

Real Implementation (Single and Double Precision)

The output vector is the sequence of the floating-point values u_1, u_2, ..., where elements u_1, u_2, ..., u_s correspond to the $\mathbf{u_1}$, u_{s+11}, u_{s+2}, ..., u_{2s} correspond to the $\mathbf{u_2}$, and so on.

Integer Implementation

The output vector of 32-bit integers x_1, x_2, ..., where elements x_1, x_2, ..., x_s correspond to the $\mathbf{x_1}$, x_{s+11}, x_{s+2}, ..., x_{2s} correspond to the $\mathbf{x_2}$, and so on.

Stream Initialization by Function `vslNewStream`

NIEDERREITER generates the stream and initializes it specifying the input 32-bit parameter seed (dimension dimen of a quasi-random vector):

1. Assume `dimen` = seed
2. If `dimen` < 1 or `dimen` > 318, assume `dimen` = 1.

Stream Initialization by Function `vslNewStreamEx`

NIEDERREITER generates the stream and initializes it specifying the array `params[]` of *n* 32-bit integers to set the dimension `dimen` of a quasi-random vector as well as pass other generator related parameters, for example, irreducible polynomials, or direction numbers (matrix of the generator). The general interface for passing stream the polynomials via the `params[]` array has the following format:

Position in `params[]`	0	1	2	`3...2+dimen`
	`dimen`	Parameter Class Indicators	Initial Values Subclass Indicators	Irreducible polynomials

The dimension parameter params[0] is mandatory, and can be initialized as follows:

```
params[0] = dimen;
```

The other elements of params intended for passing additional user-supplied data are optional. For example, if they are not presented, then the default table of irreducible polynomials is used for generation of quasi-random vectors. VS default tables of the polynomials allow generating quasi-random sequences for dimensions up to 318.

If you want to generate quasi-random vectors of greater dimension or obtain another sequence, you can register a set of your own irreducible polynomials. In order to do this, you need to set the Parameter Class Indicators field (`params[1]`) to `VSL_USER_QRNG_INITIAL_VALUES`:

```
params[1] = VSL_USER_QRNG_INITIAL_VALUES;
```

Further, you should indicate in Initial Values Subclass Indicators field (`params[2]`) that you want to supply irreducible polynomials:

```
params[2] = VSL_USER_IRRED_POLYMS;
```

The remainder of the params array is used to pass irreducible polynomials. They are packed as unsigned integers and serially set into corresponding positions of the params array as shown in the example below (number of the polynomials equals the dimension `dimen`):

```
unsigned int uNiederrIrredPoly[dimen] = {...};
...
params[0] = dimen;
params[1] = VSL_USER_QRNG_INITIAL_VALUES;
params[2] = VSL_USER_IRRED_POLYMS;

for ( i = 0; i < dimen; i++ ) params[i+3] = uNiederrIrredPoly[i];
```

You can also calculate direction numbers (matrix of the generator) using your own irreducible polynomials and pass this table to the generator. The interface for registration of the direction numbers is as follows:

Position in `params[]`	0	1	2	`3...dimen*32+2`
	`dimen`	Parameter Class Indicators	Initial Values Subclass Indicators	Direction numbers

The dimension parameter `params[0]` and Parameter Class Indicators field (`params[1]`) can be initialized as follows:

```
params[0] = dimen;
params[1] = VSL_USER_QRNG_INITIAL_VALUES;
```

Further, you need to initialize Initial Values Subclass Indicators field (`params[2]`):

```
params[2] = VSL_USER_DIRECTION_NUMBERS;
```

Direction numbers are assumed to be a dimen x 32 table of unsigned integers and can be passed to the generator in the following way:

```
unsigned int uNiederrCJ[dimen][32] = {...};
params[0] = dimen;
params[1] = VSL_USER_QRNG_INITIAL_VALUES;
params[2] = VSL_USER_DIRECTION_NUMBERS;
k = 3;
for ( i = 0; i < dimen; i++ )
{
     for ( j = 0; j < 32; j++ )
     {
          params[k++] = uNiederrCJ[i][j];
     }
                     }
```

In short, the NIEDERREITER stream initialization is as follows:

1. If *n* = 0, assume `dimen` = 1 and initialize the stream using the default table of irreducible polynomials.
2. If *n* = 1, dimen = params[0], initialize the stream using the default table of irreducible polynomials; (if dimen < 1 or dimen > 318, assume dimen = 1).
3. If *n* > 1, dimen = params[0]
    a. If dimen < 1, assume dimen = 1 and initialize the stream using the default table of irreducible polynomials.

b. If the externally defined parameters of the generator are packed incorrectly, initialize the stream using the default table of irreducible polynomials; (if dimen > 318, assume dimen = 1).

c. Initialize the NIEDERREITER quasi-random stream by means of the table of user-defined irreducible polynomials.

Subsequences Selection Methods

`vslSkipAheadStream`	supported
`vslLeapfrogStream`	supported

NOTE

1. The skip-ahead method skips individual components of quasi-random vectors rather than whole s-dimensional vectors. Hence, to skip N s-dimensional quasi-random vectors, call `vslSkipAheadStream` subroutine with parameter `nskip` equal to the N$\times$s.
2. The leapfrog method works with individual components of quasi-random vectors rather than with s-dimensional vectors. In addition, its functionality allows picking out a fixed quasi-random component only. In other words, nstreams parameter should be equal to the predefined constant `VSL_QRNG_LEAPFROG_COMPONENTS`, and k parameter should indicate the index of a component of s-dimensional quasi-random vectors to be picked out ($0 \le k < s$).

Generator Period

$$\rho = 2^{32} \approx 4.2\times 10^{9}$$

Dimensions

$1 \le s \le 318$ is the default set of dimensions; user-defined dimensions are accepted.

6.4.11. Philox4x32-10

This is a keyed family of counter-based BRNGs. The state consists of 128-bit integer counter c and two 32-bit keys k_0 and k_1. The generator has 32-bit integer output obtained in the following way [Salmon2011]:

1. $c_n = c_{n-1} + 1$
2. $w_n = f(c_n)$, where f is a function that takes a 128-bit argument and returns a 128-bit number. The returned number is obtained as follows:
   1. The argument c is interpreted as four 32-bit numbers $c = \overline{L_1R_1L_0R_0}$, where $\overline{ABCD} = A \cdot 2^{96} + B \cdot 2^{64} + C \cdot 2^{32} + D$, put $k_0^0 = k_0$ and $k_1^0 = k_1$.
   2. The following recurrence is calculated:

$$L_1^{i+1} = \mathrm{mullo}\left(R_1^i, 0xD2511F53\right)$$
$$R_1^{i+1} = \mathrm{mulhi}\left(R_0^i, 0xCD9E8D57\right) xor\ k_0\ xor\ L_0$$
$$L_0^{i+1} = \mathrm{mullo}\left(R_0^i, 0xCD9E8D57\right)$$
$$R_0^{i+1} = \mathrm{mulhi}\left(R_1^i, 0xD2511F53\right)\ xor\ k_1\ xor\ L_1$$
$$k_0^{i+1} = k_0^i + 0xBB67AE85$$
$$k_1^{i+1} = k_1^i + 0x9E3779B9$$

Where `mulhi(a,b)` and `mullo(a,b)` are high and low 32-bit parts of the a*b product, respectively.

3. Put $f(c) = \overline{L_1^N R_1^N L_0^N R_0^N}$, where $N = 10$

3. Integer output: $r_{4n+k} = w_n(k)$, where $w_n(k)$ is the k-th 32-bit integer in quadruple w_n, $k = 0, 1, 2, 3$
4. Real output: $u_n = (int)r_n/2^{32} + ½$

Real Implementation (Single and Double Precision)

The output vector is the sequence of the floating-point values u_0, u_1, ...

Integer Implementation

The output vector of 32-bit integers r_0, r_1, ...

Stream Initialization by Function `vslNewStream`

Philox4x32-10 generates the stream and initializes it specifying the 32-bit input integer parameter seed. The stream state is a 128-bit number c and a pair of 32-bit integers k_0 and k_1 initialized in the following way:

1. Assume k_0 = seed.
2. Assume the other values are equal to 0, that is $k_1 = 0$ and $c = 0$.

Stream Initialization by Function `vslNewStreamEx`

Philox4x32-10 generates the stream and initializes it specifying the array `params[]` of n 32-bit integers:

1. If $n = 0$, assume $c = k_0 = k_1 = 0$.
2. If $n = 1$, assume k = params[0], $c = 0$.
3. If $n = 2$, assume k = params[0] + params[1]*2^{32}, $c = 0$.
4. If $n = 3$, assume k = params[0] + params[1]*2^{32}, c = params[2].
5. If $n = 4$, assume k = params[0] + params[1]*2^{32}, c = params[2] + params[3]*2^{32}.
6. If $n = 5$, assume k = params[0] + params[1]*2^{32}, c = params[2] + params[3]*2^{32}+ params[4]*2^{64}.
7. If $n >= 6$, assume k = params[0] + params[1]*2^{32}, c = params[2] + params[3]*2^{32}+ params[4]*2^{64}+ params[5]*2^{96}.

Subsequences Selection Methods

`vslSkipAheadStream`	supported
`vslLeapfrogStream`	not supported

Generator Period

$$\rho = 2^{130} \approx 1.4 \times 10^{39}$$

Empirical Testing Results Summary

NOTE

1. The tabulated data is obtained using the one-level (threshold) testing technique. The OK result indicates FAIL < 50%. The run fails when p-value falls outside the interval [0.05, 0.95].
2. The stream tested is generated by calling the function `vslNewStream` with seed=7,777,777.

6.4.12. ARS5

This is a keyed family of counter-based BRNGs. The state consists of 128-bit integer counter *c* and a 128-bit key *k*. The BRNG is based on the AES encryption algorithm [FIPS-197]. The 32-bit output is obtained in the following way [Salmon2011]:

1. The *i*-th number is defined by the following formula: $r_i = (f(i/4) >> ((i \bmod 4) * 32) \bmod 2^{32}$
2. Function *f(c)* takes 128-bit input and produces 128-bit result obtained in the following way:
    1. Put $c_0 = c$ *xor* k and $k_0 = k$.
    2. The following recurrence is calculated N times:
        1. c_{i+1} = `SubBytes(`c`)`
        2. c_{i+1} = `ShiftRows(`c_{i+1}`)`
        3. c_{i+1} = `MixColumns(`c_{i+1}`)`, this step is omitted if $i + 1 = N$
        4. c_{i+1} = `AddRoundKey(`c_{i+1}, k_j`)`
        5. $Lo(k_{i+1}) = Lo(k)$ + 0x9E3779B97F4A7C15

           $Hi(k_{i+1}) = Hi(k)$ + 0xBB67AE8584CAA73B
    3. Put $f(c) = c_N$, where $N = 5$
3. Real output: $u_n = (int)r_n/2^{32} + ½$

Specification for the `SubBytes`, `ShiftRows`, `MixColumns` and `AddRoundKey` functions can be found in [FIPS-197].

Real Implementation (Single and Double Precision)

The output vector is the sequence of the floating-point values u_0, u_1, ...

Integer Implementation

The output vector of 32-bit integers r_0, r_1, ...

Stream Initialization by Function `vslNewStream`

ARS5 generates the stream and initializes it specifying the 32-bit input integer parameter seed. The stream state is two 128-bit numbers *c* and *k* initialized in the following way:

1. Assume k = seed.
2. Assume $c = 0$.

Stream Initialization by Function `vslNewStreamEx`

ARS5 generates the stream and initializes it specifying the array `params[]` of *n* 32-bit integers:

1. If $n = 0$, assume $c = k = 0$.
2. If $n = 1$, assume k = params[0], $c = 0$.
3. If $n = 2$, assume k = params[0] + params[1]*2^{32}, $c = 0$.

4. If *n* = 3, assume *k* = params[0] + params[1]*2^{32} + params[2]*2^{64}, *c* = 0.
5. If *n* = 4, assume *k* = params[0] + params[1]*2^{32} + params[2]*2^{64} + params[3]*2^{96}, *c* = 0.
6. If *n* = 5, assume *k* = params[0] + params[1]*2^{32} + params[2]*2^{64} + params[3]*2^{96}, *c* = params[4].
7. If *n* = 6, assume *k* = params[0] + params[1]*2^{32} + params[2]*2^{64} + params[3]*2^{96}, *c* = params[4] + params[5]*2^{32}.
8. If *n* = 7, assume *k* = params[0] + params[1]*2^{32} + params[2]*2^{64} + params[3]*2^{96}, *c* = params[4] + params[5]*2^{32} + params[6]*2^{64}.
9. If *n* >= 8, assume *k* = params[0] + params[1]*2^{32} + params[2]*2^{64} + params[3]*2^{96}, *c* = params[4] + params[5]*2^{32} + params[6]*2^{64} + params[7]*2^{96}.

Subsequences Selection Methods

`vslSkipAheadStream`	supported
`vslLeapfrogStream`	not supported

Generator Period

$\rho = 2^{130} \approx 1.4 \times 10^{39}$

Empirical Testing Results Summary

Test Name	**`vsRngUniform`**	**`vdRngUniform`**	**`viRngUniform`**	**`viRngUniformBits`**
3D Spheres Test	OK (20% errors)	OK (20% errors)	Not applicable	OK (20% errors)
Birthday Spacing Test	Not applicable	Not applicable	Not applicable	OK (0% errors)
Bitstream Test	Not applicable	Not applicable	Not applicable	OK (15% errors)
Rank of 31x31 Binary Matrices Test	Not applicable	Not applicable	Not applicable	OK (0% errors)
Rank of 32x32 Binary Matrices Test	Not applicable	Not applicable	Not applicable	OK (0% errors)
Rank of 6x8 Binary Matrices Test	Not applicable	Not applicable	Not applicable	OK (0% errors)
Counts-the-1's Test (stream of bits)	Not applicable	Not applicable	Not applicable	OK (0% errors)

Counts-the-1's Test (stream of specific bytes)	Not applicable	Not applicable	Not applicable	OK (0% errors)
Craps Test	OK (30% errors)	OK (30% errors)	OK (30% errors)	OK (30% errors)
Parking Lot Test	OK (10% errors)	OK (10% errors)	Not applicable	OK (10% errors)
2D Self-Avoiding Random Walk Test	OK (20% errors)	OK (10% errors)	Not applicable	OK (10% errors)

NOTE

1. The tabulated data is obtained using the one-level (threshold) testing technique. The OK result indicates FAIL < 50%. The run fails when p-value falls outside the interval [0.05, 0.95].
2. The stream tested is generated by calling the function `vslNewStream` with seed=7,777,777.

7. Testing of Distribution Random Number Generators

VS generators are tested with a set of tests to control the quality of random number sequences of general discrete and continuous distributions.

Random numbers of discrete and continuous distributions are generated by transforming random numbers of uniform distribution. A source of uniformly distributed random numbers is a random stream produced by a basic generator. Quality of the random number sequences with non-uniform distribution greatly depends on the quality of the respective basic generator. Therefore, generators of discrete and continuous distributions are tested for each individual basic generator.

VS can provide several methods of random number generation for any probability distribution. For example, two methods are implemented for Poisson distribution: PTPE acceptance/rejection algorithm and PoisNorm inverse transformation algorithm, based on transformation of normal distribution. The generator is tested for each of the implemented methods.

VS offers two different implementations for each of continuous distributions:

1. single-precision real arithmetic
2. double-precision real arithmetic

As a rule, a single-precision version of the generator is faster than a double-precision one. Moreover, single-precision version is quite sufficient for most applications. VS offers only one version for discrete distributions.

Apart from the above-mentioned factors, RNGs are dependent for their quality on distribution parameters. For example, different transformation techniques may be used for different parameters. Therefore, generators are also tested for different parameter sets.

7.1. Interpreting Test Results

Test results for general distribution generators are interpreted similarly to basic generators. For reliable results, either one-level (threshold) or two-level testing is performed.

7.2. Description of Distribution Generator Tests

This section describes the available Distribution Generator Tests:

1. Confidence Test

2. Distribution Moments Test
3. Chi-Squared Goodness-of-Fit Test

7.2.1. Confidence Test

Test Purpose

The test checks how well each output member corresponds to the valid range of possible values. For example, for an exponential distribution with parameters a and b all the output members xi should lie within the range $a \leq x_i < \infty$. A value $x_i < 1$ is impossible, that is, the fact that the variate X of exponential distribution with parameters a and b acquires a value less than a is an impossible event (not to be confused with a null event). Any output member lying outside the valid range causes an error.

Such a test is necessary because statistical tests (for example, distribution moments test or chi-square test) cannot detect a small number (if compared with the total sample size) of x_i values falling outside the valid range.

Interpreting Final Results

The test gives a certain quantity K of random numbers that lie outside the valid range of values. The test is considered passed, if $K = 0$, and failed otherwise.

7.2.2. Distribution Moments Test

Test Purpose

The test verifies that sample moments of a given distribution agree with theoretical moments. Sample mean (first order moment) and sample variance (central moment of the second order) are considered as stable responses.

First Level Test

The generated random number sequence is used to compute the sample mean M and the sample variance D that are of an asymptotically normal distribution. Proceeding from this asymptotic, p-values p^M and p^D are found using the values of M and D.

Second Level Test

The first level test is run ten times, each run producing a pair of p-values p^M and p^D, $j = 1, 2, \dots, 10$. The Kolmogorov-Smirnov goodness-of-fit test with Anderson-Darling's statistics is applied to the obtained p-values p_j^M, $j = 1, 2, \dots, 10$. If the resulting p-value $p^M < 0.05$ or $p^M > 0.95$, the test is considered failed for the sample mean. The same procedure is performed for p-values p_j^D, $j = 1, 2, \dots, 10$, and if p-value $p^D < 0.05$ or $p^D > 0.95$, the test is considered failed for the sample variance.

Interpreting Final Results

Ten runs of the second level test provide the percentage FAILM of failed tests for the sample mean and the percentage FAILD of failed tests for the sample variance. The final result of the test is the percentage FAIL = max(FAILM, FAILD). The value of FAIL < 50% is considered acceptable.

7.2.3. Chi-Squared Goodness-of-Fit Test

Test Purpose

The test verifies that the sample distribution function agrees with the hypothesized distribution. A chi-squared *V* statistic with the number of degrees of freedom that is minus one from the number of the intervals of partition is considered a stable response.

First Level Test

For a given parameter set and a given sample size, the test computes the partition of the distribution domain into disjoint intervals, so that the a priori quantity of random numbers from each interval is of order 100.

The test computes the actual number of random values within each interval of the generated sample and then calculates chi-square of the statistic *V*. The statistic *V* is asymptotically of chi-squared distribution $F_{k-1}(x)$ with k - 1 degrees of freedom, where k is the number of the intervals. Thus, the p-value equal to $F_{k-1}(V)$ should be of a distribution that is close to uniform.

Second Level Test

The first level test is run ten times, each run producing a p-value p_j, j = 1, 2, ... , 10. The Kolmogorov-Smirnov goodness-of-fit test with Anderson-Darling's statistics is applied to the obtained p-values p_j j = 1, 2, ... , 10. If the resulting p-value p^M

Interpreting Final Results

The final result of the test is the percentage FAIL of failed second level tests. The second level test is run ten times. The value of FAIL < 50% is considered acceptable.

7.2.4. Performance

The following factors influence the performance of an RNG of a given distribution:

1. architecture and configuration of the hardware and software
2. performance of the underlying BRNG
3. method of transformation
4. number of random numbers to be generated (size of the output vector)
5. parameters of a given probability distribution

VS random number generators are optimized for Intel® Xeon® Processor X7560 and Intel® Xeon® Processor X5670. For more details on performance, see Vector Statistics (VS) Performance Data document available at http://software.intel.com/en-us/articles/intel-math-kernel-library-documentation/. For earlier Intel processors, VS generators are fully functional, but not specifically optimized.

The value of Clocks Per Element (CPE), which is independent from the processor clock rate, is selected as a unit of measurement.

For example, if the generator performance is equal to 10 CPE and the processor rate is 1 GHz, then the generator produces 108 random numbers per second.

The VS BRNGs differ from each other in speed, therefore data on performance of general (discrete and continuous) distribution generators is given separately for each BRNG used as an underlying generator to produce uniformly distributed random numbers.

Performance of a general distribution generator also depends on a method chosen for transforming a uniform distribution to a given non-uniform one. This requires specifying the applied transformation method as well.

The length of a generated vector is another factor influencing the performance of the VS vector type generators. Calling generators on short vector lengths may prove highly ineffective. See the figure for the typical interdependence between the generator performance and the vector length.

Finally, the generator performance may vary according to probability distribution parameters. The tables provide performance data only for fixed parameter values (or fixed intervals of parameter variations). Table footnotes contain parameters with which a given performance is obtained. For some transformation methods the performance is approximately the same on a wide range of parameters, such methods being called uniformly fast, while for others the performance may vary considerably with variation in the distribution parameters, for example, in PTPE method for an RNG of Poisson distribution. When the latter is the case, graphs of interdependence between the performance and the distribution parameters are provided.

7.3. Continuous Distribution Functions

This section describes VS Continuous Distribution Random Number Generators:

1. Uniform (VSL_RNG_METHOD_UNIFORM_STD/VSL_RNG_METHOD_UNIFORM_STD_ACCURATE)
2. Gaussian (VSL_RNG_METHOD_GAUSSIAN_BOXMULLER)
3. Gaussian (VSL_RNG_METHOD_GAUSSIAN_BOXMULLER2)
4. Gaussian (VSL_RNG_METHOD_GAUSSIAN_ICDF)
5. GaussianMV (VSL_RNG_METHOD_GAUSSIANMV_BOXMULLER)
6. GaussianMV (VSL_RNG_METHOD_GAUSSIANMV_BOXMULLER2)
7. GaussianMV (VSL_RNG_METHOD_GAUSSIANMV_ICDF)
8. Exponential (VSL_RNG_METHOD_EXPONENTIAL_ICDF/VSL_RNG_METHOD_EXPONENTIAL_ICDF_ACCURATE)
9. Laplace (VSL_RNG_METHOD_LAPLACE_ICDF)
10. Weibull (VSL_RNG_METHOD_WEIBULL_ICDF/VSL_RNG_METHOD_WEIBULL_ICDF_ACCURATE)
11. Cauchy (VSL_RNG_METHOD_CAUCHY_ICDF)
12. Rayleigh (VSL_RNG_METHOD_RAYLEIGH_ICDF/VSL_RNG_METHOD_RAYLEIGH_ICDF_ACCURATE)
13. Lognormal (VSL_RNG_METHOD_LOGNORMAL_ BOXMULLER2/VSL_RNG_METHOD_LOGNORMAL_ BOXMULLER2_ACCURATE)
14. Lognormal (VSL_RNG_METHOD_LOGNORMAL_ICDF/VSL_RNG_METHOD_LOGNORMAL_ICDF_ACCURATE)
15. Gumbel (VSL_RNG_METHOD_GUMBEL_ICDF)
16. Gamma (VSL_RNG_METHOD_GAMMA_GNORM/VSL_RNG_METHOD_GAMMA_GNORM_ACCURATE)
17. Beta (VSL_RNG_METHOD_BETA_CJA/VSL_RNG_METHOD_BETA_CJA_ACCURATE)
18. ChiSquare (VSL_RNG_METHOD_CHISQUARE_CHI2GAMMA)

7.3.1. Uniform (VSL_RNG_METHOD_UNIFORM_STD/VSL_RNG_METHOD_UNIFORM_STD_ACCURATE)

Random number generator of uniform distribution over the real interval [*a*,*b*). You may identify the underlying BRNG by passing the random stream descriptor `stream` as a parameter. Then the Uniform function calls real implementation (of single precision for `vsRngUniform` and of double precision for `vdRngUniform`) of this BRNG.

See Intel® MKL Vector Statistics Random Number Generator Performance Data for test results summary.

7.3.2. Gaussian (VSL_RNG_METHOD_GAUSSIAN_BOXMULLER)

Random number generator of normal (Gaussian) distribution with parameters *a* and *s*. You can obtain any successive random number *x* of the standard normal distribution according to the formula

$$x = \sqrt{-2\ln u_1}\,\sin 2\pi u_2,$$

where u_1, u_2 are a pair of successive random numbers uniformly distributed over the interval (0, 1). For details, see [Box58].

The normal distribution with the parameters *a* and *s* is transformed to the random number *y* by scaling and the shift *y = sx+a*.

See Intel® MKL Vector Statistics Random Number Generator Performance Data for test results summary.

7.3.3. Gaussian (VSL_RNG_METHOD_GAUSSIAN_BOXMULLER2)

Random number generator of normal (Gaussian) distribution with parameters *a* and *s*. You can produce a successive pair of the random numbers x_1, x_2 of the standard normal distribution according to the formula

$$x_1 = \sqrt{-2\ln u_1}\,\sin 2\pi u_2$$

$$x_2 = \sqrt{-2\ln u_1}\,\cos 2\pi u_2$$

where u_1, u_2 are a pair of successive random numbers uniformly distributed over the interval (0, 1). For details, see [Box58].

The normal distribution with the parameters *a* and *s* is transformed to the random number *y* by scaling and the shift *y = sx+a*.

You can safely call this VS method even when the random numbers are generated in blocks with the size aliquant to 2.

For example, you use the VSL_METHOD_DGAUSSIAN_BOXMULLER2 method to generate a pair of random numbers of the standard normal distribution.

Option 1

Single call of method VSL_METHOD_DGAUSSIAN_BOXMULLER2 with the vector length equal to 2:

```
...
double x[2];
...
vdRngGaussian(VSL_RNG_METHOD_GAUSSIAN_BOXMULLER2, stream, 2, x, 0.0, 1.0);
...
```

In this case, you generate the random numbers *x*[0], *x*[1] by the formula:

$$x[0] = \sqrt{-2\ln u_1}\,\sin 2\pi u_2$$

$$x[1] = \sqrt{-2\ln u_1}\,\cos 2\pi u_2$$

Option 2

Double call of the method VSL_METHOD_DGAUSSIAN_BOXMULLER2 with the vector length equal to 1:

```
...
double x[2];
...
vdRngGaussian(VSL_RNG_METHOD_GAUSSIAN_BOXMULLER2, stream, 1, &x[0], 0.0, 1.0);
vdRngGaussian(VSL_RNG_METHOD_GAUSSIAN_BOXMULLER2, stream, 1, &x[1], 0.0, 1.0);
...
```

At the first call of vdRngGaussian you produce the random number *x*[0] by the formula:

$$x[0]=\sqrt{-2\ln u_1}\sin 2\pi u_2$$

At the second call of `vdRngGaussian` the vector length, over which you initially called the function to generate the random stream, is recognized as odd (equal to 1 in this case). Then the random number *x*[1] is generated by the formula:

$$x[1]=\sqrt{-2\ln u_1}\cos 2\pi u_2$$

and not by the formula:

$$x[1]=\sqrt{-2\ln u_3}\sin 2\pi u_4$$

See Intel® MKL Vector Statistics Random Number Generator Performance Data for test results summary.

7.3.4. Gaussian (VSL_RNG_METHOD_GAUSSIAN_ICDF)

Random number generator of normal (Gaussian) distribution with parameters *a* and *s*. You can obtain any successive random number *x* of the standard normal distribution by the inverse transformation method from the following formula:

$$x=\sqrt{2}\,erf^{-1}(u)$$

where *u* is a random number uniformly distributed over the interval (-1, 1), and $erf^{-1}(u)$ is inverse to the error function

$$erf(u)=\frac{2}{\sqrt{\pi}}\int_0^u \exp(-t^2)\,dt$$.

The normal distribution with parameters *a* and *s* is transformed to the random number *y* by scaling and the shift *y* = *sx*+*a*.

See Intel® MKL Vector Statistics Random Number Generator Performance Data for test results summary.

7.3.5. GaussianMV (VSL_RNG_METHOD_GAUSSIANMV_BOXMULLER)

Random number generator of d-variate (correlated) normal distribution with parameters *a* and *T*. You can obtain any successive random vector **x** according to the formula

$$\mathbf{x}_n=\mathbf{T}\mathbf{z}_n+\mathbf{a}$$

where:

1. $\mathbf{z}_n$ is a d-dimensional vector of random numbers from standard normal distribution
2. **T** is a lower triangular dxd matrix - Cholesky factor of variance-covariance matrix

Random numbers from standard normal distribution are generated by method

VSL_RNG_METHOD_GAUSSIAN_BOXMULLER.

See Intel® MKL Vector Statistics Random Number Generator Performance Data for test results summary and performance graphs.

7.3.6. GaussianMV (VSL_RNG_METHOD_GAUSSIANMV_BOXMULLER2)

Random number generator of d-variate (correlated) normal distribution with parameters *a* and *T*. You can obtain any successive random vector **x** according to the formula

$$\mathbf{x}_n = \mathbf{T}\mathbf{z}_n + \mathbf{a}$$

where:

1. $\mathbf{z}_n$ is a d-dimensional vector of random numbers from standard normal distribution
2. **T** is a lower triangular dxd matrix - Cholesky factor of variance-covariance matrix

Random numbers from standard normal distribution are generated by method

VSL_RNG_METHOD_GAUSSIAN_BOXMULLER2.

See Intel® MKL Vector Statistics Random Number Generator Performance Data for test results summary and performance graphs.

7.3.7. GaussianMV (VSL_RNG_METHOD_GAUSSIANMV_ICDF)

Random number generator of d-variate (correlated) normal distribution with parameters *a* and *T*. You can obtain any successive random vector **x** according to the formula

$$\mathbf{x}_n = \mathbf{T}\mathbf{z}_n + \mathbf{a}$$

where:

1. $\mathbf{z}_n$is a d-dimensional vector of random numbers from standard normal distribution
2. **T** is a lower triangular dxd matrix - Cholesky factor of variance-covariance matrix

Random numbers from standard normal distribution are generated by method

VSL_RNG_METHOD_GAUSSIAN_ICDF.

See Intel® MKL Vector Statistics Random Number Generator Performance Data for test results summary and performance graphs.

7.3.8. Exponential (VSL_RNG_METHOD_EXPONENTIAL_ICDF/VSL_RNG_METHOD_EXPONENTIAL_ICDF_ACCURATE)

Random number generator of the exponential distribution with parameters *a* and *β*. You can generate any successive random number *x* of the exponential distribution by the inverse transformation method from the formula:

$$x = -\beta \ln(u) + a$$

where *u* is a successive random number of a uniform distribution over the interval (0, 1).

See Intel® MKL Vector Statistics Random Number Generator Performance Data for test results summary.

7.3.9. Laplace (VSL_RNG_METHOD_LAPLACE_ICDF)

Random number generator of the Laplace distribution with parameters *a* and *β*. You can generate any successive random number *x* of the Laplace distribution by the inverse transformation method from the formula:

$$x = \begin{cases} -\beta\ln(u_1)+a, & u_2 \le 1/2 \\ \beta\ln(u_1)+a, & u_2 > 1/2 \end{cases}$$

where u_1, u_2 is a pair of successive random numbers of a uniform distribution over the interval (0, 1).

See Intel® MKL Vector Statistics Random Number Generator Performance Data for test results summary.

7.3.10. Weibull (VSL_RNG_METHOD_WEIBULL_ICDF/ VSL_RNG_METHOD_WEIBULL_ICDF_ACCURATE)

Random number generator of the Weibull distribution with the parameters *α*, *a* and *β*. You can generate any successive random number *x* of the Weibull distribution by the inverse transformation method from the formula

$$x = \beta\left(-\ln(u)\right)^{1/\alpha} + a$$

where *u* is a successive random number of a uniform distribution over the interval (0, 1).

See Intel® MKL Vector Statistics Random Number Generator Performance Data for test results summary.

7.3.11. Cauchy (VSL_RNG_METHOD_CAUCHY_ICDF)

Random number generator of the Cauchy distribution with parameters *a* and *β*. You can generate any successive random number *x* of the Cauchy distribution by the inverse transformation method from the formula

$$x = \beta\tan u + a$$

where *u* is a successive random number of a uniform distribution over the interval (-p/2, p/2).

See Intel® MKL Vector Statistics Random Number Generator Performance Data for test results summary.

7.3.12. Rayleigh (VSL_RNG_METHOD_RAYLEIGH_ICDF/ VSL_RNG_METHOD_RAYLEIGH_ICDF_ACCURATE)

Random number generator of the Rayleigh distribution with the parameters *a* and *β*. You can generate any successive random number *x* of the Rayleigh distribution by the inverse transformation method from the formula

$$x = \beta\sqrt{-\ln u} + a,$$

where *u* is a successive random number of a uniform distribution over the interval (0, 1).

See Intel® MKL Vector Statistics Random Number Generator Performance Data for test results summary.

7.3.13. Lognormal (VSL_RNG_METHOD_LOGNORMAL_BOXMULLER2/VSL_RNG_METHOD_LOGNORMAL_BOXMULLER2_ACCURATE)

Random number generator of the lognormal distribution with parameters *a*, *σ*, *b* and *β*. You can generate any successive random number *x* of the lognormal distribution by the inverse transformation method from the formula

$$x = \beta \exp(y) + b$$

where *y* is a successive random number of a normal (Gaussian) distribution with parameters *a* and *σ*.

The random numbers of the normal distribution are generated using method VSL_RNG_METHOD_GAUSSIAN_BOXMULLER2.

See Intel® MKL Vector Statistics Random Number Generator Performance Data for test results summary.

7.3.14. Lognormal (VSL_RNG_METHOD_LOGNORMAL_ICDF/VSL_RNG_METHOD_LOGNORMAL_ICDF_ACCURATE)

Random number generator of the lognormal distribution with parameters *a*, *σ*, *b* and *β*. You can generate any successive random number *x* of the lognormal distribution by the inverse transformation method from the formula

$$x = \beta \exp(y) + b$$

where *y* is a successive random number of a normal (Gaussian) distribution with parameters *a* and *σ*.

The random numbers of the normal distribution are generated using method VSL_RNG_METHOD_GAUSSIAN_ICDF.

See Intel® MKL Vector Statistics Random Number Generator Performance Data for test results summary.

7.3.15. Gumbel (VSL_RNG_METHOD_GUMBEL_ICDF)

Random number generator of the Gumbel distribution with parameters *a* and *β*. You can generate any successive random number *x* of the Gumbel distribution by the inverse transformation method from the formula

$$x = \beta \ln(y) + a$$

where *y* is a successive random number of an exponential distribution with the parameters *a* = 0 and *β* = 0.

The random numbers of the exponential distribution are generated using method VSL_RNG_METHOD_EXPONENTIAL_ICDF.

See Intel® MKL Vector Statistics Random Number Generator Performance Data for test results summary.

7.3.16. Gamma (VSL_RNG_METHOD_GAMMA_GNORM/ VSL_RNG_METHOD_GAMMA_GNORM_ACCURATE)

Random number generator of the gamma distribution with parameters shape *α*, offset *a*, and scale factor *β*. You can generate any successive random number y_α of the standard gamma distribution (*a* = 0, *β* = 1) as follows:

1. If $\alpha > 1$, a gamma distributed random number can be generated as a cube of properly scaled normal random number [Mars2000]. The algorithm is based on the acceptance/rejection method using squeeze technique.
2. If $\alpha < 1$, a gamma distributed random number is generated using two acceptance/rejection based algorithms:
   - If $\alpha < 0.6$, a gamma distributed random number is obtained by transformation of exponential power distributed random number [Dev86],
   - Otherwise, rejection method from Weibull distribution is used [Vad77], [Dev86].

When $\alpha = 1$ gamma distribution is reduced to exponential distribution with parameters a, β. The random numbers of the exponential distribution are generated using method VSL_RNG_METHOD_EXPONENTIAL_ICDF. The gamma distributed random number γ with parameters α, a, and β is transformed from γ_α using scale and shift $\gamma = a + \beta\gamma_\alpha$.

See Intel® MKL Vector Statistics Random Number Generator Performance Data for test results summary.

7.3.17. Beta (VSL_RNG_METHOD_BETA_CJA/ VSL_RNG_METHOD_BETA_CJA_ACCURATE)

Random number generator of the beta distribution with two shape parameters p and q, offset a, and scale factor β. You can generate any successive random number $\Theta(p,q)$ of the standard gamma distribution ($a = 0$, $\beta = 1$) as follows:

1. If $min(p,q) > 1$, use Cheng algorithm. For details, see [Cheng78].
2. If $max(p,q) < 1$, apply a composition of two algorithms:
   a. If $q + K{*}P^2 + C \le 0$, where $K = 0.852...$, $C = - 0.956...$, use Jöhnk algorithm. For details, see [Jöhnk64].
   b. Otherwise, use Atkinson switching algorithm. For details, see [Atkin79].
3. If $min(p,q) < 1$ and $max(p,q) > 1$, use the switching algorithm of Atkinson to generate random numbers. For details, see [Atkin79].
4. If $p = 1$ or $q = 1$, use the inverse transformation method.
5. If $p = 1$ and $q = 1$, standard beta distribution is reduced to the uniform distribution over the interval (0,1). The random numbers of the uniform distribution are generated using the `VSL_RNG_METHOD_UNIFORM_STD` method.

The algorithms of Cheng and Atkinson use acceptance/rejection technique. The beta distributed random number γ with the parameters p, q, a, and β is transformed from $\Theta(p,q)$ as follows: $\gamma = a + \beta\Theta(p,q)$

See Intel® MKL Vector Statistics Random Number Generator Performance Data for test results summary.

7.4. Discrete Distribution Functions

This section describes VS Discrete Distribution Random Number Generators:

1. Uniform (VSL_RNG_METHOD_UNIFORM_STD)
2. UniformBits (VSL_RNG_METHOD_UNIFORMBITS_STD)
3. UniformBits32 (VSL_RNG_METHOD_UNIFORMBITS32_STD)
4. UniformBits64 (VSL_RNG_METHOD_UNIFORMBITS64_STD)
5. Bernoulli (VSL_RNG_METHOD_BERNOULLI_ICDF)
6. Geometric (VSL_RNG_METHOD_GEOMETRIC_ICDF)

7. Binomial (VSL_RNG_METHOD_BINOMIAL_BTPE)
8. Hypergeometric (VSL_RNG_METHOD_HYPERGEOMETRIC_H2PE)
9. Poisson (VSL_RNG_METHOD_POISSON_PTPE)
10. Poisson (VSL_RNG_METHOD_POISSON_POISNORM)
11. PoissonV (VSL_RNG_METHOD_POISSONV_POISNORM)
12. NegBinomial (VSL_RNG_METHOD_NEGBINOMIAL_NBAR)
13. Multinomial (VSL_RNG_METHOD_MULTINOMIAL_MULTPOISSON)

7.4.1. Uniform (VSL_RNG_METHOD_UNIFORM_STD)

Uniform discrete distribution over the integer interval [*a*,*b*). You can generate any successive random number *k* of the uniform distribution by the formula:

$$k = \lfloor u \rfloor,$$

where *u* is a successive random number of a uniform (continuous) distribution over the interval [*a*,*b*) and $\lfloor x \rfloor$ stands for the operation floor(*x*) that produces the maximum integer, which does not exceed *x*.

See Intel® MKL Vector Statistics Random Number Generator Performance Data for test results summary.

7.4.2. UniformBits (VSL_RNG_METHOD_UNIFORMBITS_STD)

A random number generator of uniform distribution that produces an integer (non-normalized to the interval (0, 1)) sequence. You can identify the underlying BRNG by passing the random stream descriptor `stream` as a parameter. Then `UniformBits` function calls integer implementation of this basic generator.

Basic generators differ in bit capacity and structure of the integer output, therefore you should interpret the output integer array of the function `viRngUniformBits` correctly. The following list provides rules for interpreting 32-bit integer output `r[i]` for each VS BRNG.

MCG31m1

Integer Recurrence

$$x_i = ax_{i-1} (\mathrm{mod}\, m)$$

$$u_i = x_i / m$$

$$a = 1132489760,\ m = 2^{31} - 1$$

Interpretation of 32-bit integer output array `r[i]` after calling `viRngUniformBits`

$$r[i] = x_i$$

R250

Integer Recurrence

$$x_i = x_{i-103} \oplus x_{i-250}$$

$$u_i = x_i / 2^{32}$$

Interpretation of 32-bit integer output array `r[i]` after calling `viRngUniformBits`

$$r[i] = x_i$$

MRG32k3a

Integer Recurrence

$$x_i = a_{11}x_{i-1} + a_{12}x_{i-2} + a_{13}x_{i-3} \pmod{m_1}$$
$$y_i = a_{21}y_{i-1} + a_{22}y_{i-2} + a_{23}y_{i-3} \pmod{m_2}$$
$$z_i = x_i - y_i \pmod{m_1}$$
$$u_i = z_i / m_1$$
$$a_{11} = 0,\ a_{12} = 1403580,\ a_{13} = -810728,\ m_1 = 2^{32} - 209$$
$$a_{21} = 527612,\ a_{22} = 0,\ a_{23} = -1370589,\ m_2 = 2^{32} - 22853$$

Interpretation of 32-bit integer output array `r[i]` after calling `viRngUniformBits`

$$r[i] = z_i$$

MCG59

Integer Recurrence

$$x_n = ax_{n-1} \pmod{m}$$
$$u_n = x_n / m$$
$$a = 13^{13},\ m = 2^{59}$$

Interpretation of 32-bit integer output array `r[i]` after calling `viRngUniformBits`

$$r[2i] = Lo(x_i),$$
$$r[2i+1] = Hi(x_i)$$

WH

Integer Recurrence

$$x_n = a_{1,j}x_{n-1} \pmod{m_{1,j}}$$
$$y_n = a_{2,j}y_{n-1} \pmod{m_{2,j}}$$
$$z_n = a_{3,j}z_{n-1} \pmod{m_{3,j}}$$
$$w_n = a_{4,j}w_{n-1} \pmod{m_{4,j}}$$
$$u_n = \left(x_n/m_{1,j} + y_n/m_{2,j} + z_n/m_{3,j} + w_n/m_{4,j} \right) \bmod 1$$

Interpretation of 32-bit integer output array `r[i]` after calling `viRngUniformBits`

$$r[4i] = x_i$$
$$r[4i+1] = y_i$$
$$r[4i+2] = z_i$$
$$r[4i+3] = w_i$$

MT19937

Integer Recurrence

$$x_n = x_{n-(624-397)} \oplus ((x_{n-624}\ \&\ 0\text{x}80000000) \mid (x_{n-624+1}\ \&\ 0\text{x}7\text{FFFFFFF}))A$$

$y_n = x_n$

$y_n = y_n \oplus (y_n >> 11)$

$y_n = y_n \oplus ((y_n << 7) \mathbin{\&} 0x9D2C5680)$,

$y_n = y_n \oplus ((y_n << 15) \mathbin{\&} 0xEFC60000)$,

$y_n = y_n \oplus (y_n >> 18)$,

$u_n = y_n / 2^{32}$,

where

$$A = \begin{vmatrix} 0 & 1 & 0 & & \\ 0 & 0 & \ldots & 0 & \\ & & & \ldots & \\ & & 0 & 0 & 1 \\ a_{31} & a_{30} & \ldots & \ldots & a_0 \end{vmatrix},$$

with $a = a_{31} \ldots a_0 = 0x9908B0DF$.

Interpretation of 32-bit integer output array `r[i]` after calling `viRngUniformBits`

$r[i] = y_i$

MT2203

Integer Recurrence

$x_{i,j} = x_{i-(69-34),j} \oplus ((x_{i-69,j} \mathbin{\&} 0xFFFFFFE0) \mid (x_{i-69+1,j} \mathbin{\&} 0x1F)) A_j$

$y_{i,j} = x_{i,j}$

$y_{i,j} = y_{i,j} \oplus (y_{i,j} >> 12)$

$y_{i,j} = y_{i,j} \oplus ((y_{i,j} << 7) \mathbin{\&} b_j)$

$y_{i,j} = y_{i,j} \oplus ((y_{i,j} << 15) \mathbin{\&} c_j)$

$y_{i,j} = y_{i,j} \oplus (y_{i,j} >> 18)$

$u_i = y_{i,j} / 2^{32}$,

where

$$A_j = \begin{vmatrix} 0 & 1 & 0 & & \\ 0 & 0 & \ldots & 0 & \\ & & & \ldots & \\ & & 0 & 0 & 1 \\ a_{31,j} & a_{30,j} & \ldots & \ldots & a_{0,j} \end{vmatrix},$$

with $a_j = a_{31,j} \ldots a_{0,j}$, $j = 1, \ldots, 1024$.

Interpretation of 32-bit integer output array `r[i]` after calling `viRngUniformBits`

$r[i] = y_{i,j}$

SFMT19937

Integer Recurrence

$$w_n = w_0 A \oplus w_M B \oplus w_{n-2} C \oplus w_{n-1} D,$$

$$wA := (w \overset{128}{<<} 8) \oplus w,$$

$$wB := (w \overset{32}{>>} 8) \,\&\, (\text{0xBFFFFFF6 0xBFFAFFFF 0xDDFECB7F 0x DFFFFFEF}),$$

$$wC := (w \overset{128}{>>} 8),$$

$$wD := (w \overset{32}{<<} 18)$$

$$u_{4n+k} = w_n(k) / 2^{32}, w_n(k) - \text{ k } \text{32-bit integer in quadruple } w_n, k = 0,\ldots,3$$

Interpretation of 32-bit integer output array `r[i]` after calling `viRngUniformBits`

r[*i*] = $w_{i/4}$(*i* % 4)

SOBOL

Integer Recurrence

$$\mathbf{x}_n = \mathbf{x}_{n-1} \oplus \mathbf{v}_c$$

$$\mathbf{u}_n = \mathbf{x}_n / 2^{32},$$

where

$$\mathbf{x}_n = (x_{s(n-1)+1}, x_{s(n-1)+2}, \ldots, x_{sn}), \ \mathbf{u}_n = (u_{s(n-1)+1}, u_{s(n-1)+2}, \ldots, u_{sn})$$

and *s* is the dimension of quasi-random vector.

Interpretation of 32-bit integer output array `r[i]` after calling `viRngUniformBits`

$r[i-1] = x_i$

NIEDERR

Integer Recurrence

$$\mathbf{x}_n = \mathbf{x}_{n-1} \oplus \mathbf{v}_c$$

$$\mathbf{u}_n = \mathbf{x}_n / 2^{32},$$

where

$$\mathbf{x}_n = (x_{s(n-1)+1}, x_{s(n-1)+2}, \ldots, x_{sn}), \ \mathbf{u}_n = (u_{s(n-1)+1}, u_{s(n-1)+2}, \ldots, u_{sn})$$

and *s* is the dimension of quasi-random vector.

Interpretation of 32-bit integer output array `r[i]` after calling `viRngUniformBits`

$r[i-1] = x_i$

PHILOX4X32X10

Integer Recurrence

$c_n = c_{n-1} + 1$

$w_n = f(c_n)$

f(c) is computed as follows:

$c = \overline{L_1 R_1 L_0 R_0}$

$k_0^0 = k_0$

$k_1^0 = k_1$

$L_1^{i+1} = \mathrm{mullo}\left(R_1^i, 0xD2511F53\right)$

$R_1^{i+1} = \mathrm{mulhi}\left(R_0^i, 0xCD9E8D57\right) xor\ k_0\ xor\ L_0$

$L_0^{i+1} = \mathrm{mullo}\left(R_0^i, 0xCD9E8D57\right)$

$R_0^{i+1} = \mathrm{mulhi}\left(R_1^i, 0xD2511F53\right)\ xor\ k_1\ xor\ L_1$

$k_0^{i+1} = k_0^i + 0xBB67AE85$

$k_1^{i+1} = k_1^i + 0x9E3779B9$

f(c)= $\overline{L_1^N R_1^N L_0^N R_0^N}$, *N*= 10

Interpretation of 32-bit integer output array `r[i]` after calling `viRngUniformBits`

$r[i] = w_{i/4}(i \% 4)$

$w_i(k)$ is the *k*-th 32-bit integer in quadruple w_n, *k*= 0, 1, 2, 3

ARS5

Integer Recurrence

$c_n = c_{n-1} + 1$

$w_n = f(c_n)$

f(c) is computed as follows:

$c_0 = c\ xor\ k$ and $k_0 = k$

c_{i+1}= `SubBytes(`c`)`

c_{i+1}= `ShiftRows(`c_{i+1}`)`

c_{i+1}= `MixColumns(`c_{i+1}`)`, omitted if $i + 1 = N$

c_{i+1}= `AddRoundKey(`c_{i+1}, k_j`)`

$Lo(k_{i+1}) = Lo(k)$+ 0x9E3779B97F4A7C15

$Hi(k_{i+1}) = Hi(k)$+ 0xBB67AE8584CAA73B

$f(c) = c_N$, *N*= 5

Interpretation of 32-bit integer output array `r[i]` after calling `viRngUniformBits`

$r[i] = w_{i/4}(i \% 4)$

$w_i(k)$ is the *k*-th 32-bit integer in quadruple w_n, *k*= 0, 1, 2, 3

NON-DETERMINISTIC

Integer Recurrence

Non-deterministic random generator [BMT] available in the latest Intel® CPUs [AVX].

Interpretation of 32-bit integer output array `r[i]` after calling `viRngUniformBits`

$r[i] = y_i$

NOTE

1. *Lo(x)* means obtaining lower 32 bits of the 64-bit unsigned integer *x*, that is, *Lo(x)* = $x \bmod 2^{32}$.
2. *Hi(x)* means obtaining upper 32 bits of the 64-bit unsigned integer *x*, that is, *Hi(x)* = $\lfloor x/2^{32} \rfloor$.
3. *ABCD* means a 128-bit number composed of four 32-bit numbers A, B, C and D, that is $\overline{ABCD} = A \cdot 2^{96} + B \cdot 2^{64} + C \cdot 2^{32} + D$

So, when you generate an integer sequence of *n* elements, the output array `r[i]` of the function `viRngUniformBits` comprises:

1. *n* elements for the basic generators MCG31m1, R250, MRG32k3a, MT19937, MT2203, SOBOL, NIEDERR, Philox4x32-10 and ARS5.
2. 2*n* elements for the basic generator MCG59.
3. 4*n* elements for the basic generators WH and SFMT19937.

You may use the integer output, in particular, for fast generation of bit vectors. However, in this case some bits (or groups of them) might be non-random. For example, lower bits produced by linear congruential generators are less random than their higher bits. Note that quasi-random numbers are not random at all. Thoroughly check the integer output bits and bit groups for randomness before forming bit vectors from `r[i]` array.

See Intel® MKL Vector Statistics Random Number Generator Performance Data for test results summary.

7.4.3. UniformBits32 (VSL_RNG_METHOD_UNIFORMBITS32_STD)

A random number generator that produces uniformly distributed bits in 32-bit chunks.

Some basic random number generators produce integers in which not all of the bits are uniformly distributed, for example:

1. The least significant bits in the integers produced by MCG59 BRNG are less random. For example, the lower four bits form a congruential sequence with the period of, at most, 16; and the least significant bit is either constant or strictly alternating. For example, see [Knuth81].
2. By design, BRNGs do not produce the most significant bits setting them to zero. For example, MCG31m1 is a 31-bit generator, and MCG59 is a 59-bit generator.

The UniformBits32 function transforms the underlying BRNG integer recurrence so that all bits in 32-bit chunks are uniformly distributed.

This function does not support the following VS BRNGs:

1. VSL_BRNG_MCG31
2. VSL_BRNG_R250
3. VSL_BRNG_MRG32K3A
4. VSL_BRNG_WH
5. VSL_BRNG_SOBOL
6. VSL_BRNG_NIEDERR
7. VSL_BRNG_IABSTRACT
8. VSL_BRNG_DABSTRACT
9. VSL_BRNG_SABSTRACT

7.4.4. UniformBits64 (VSL_RNG_METHOD_UNIFORMBITS64_STD)

A random number generator that produces uniformly distributed bits in 64-bit chunks.

The generator addresses the same BRNG issues as UniformBits32,its 32-bit counterpart.

The UniformBits64 function transforms the underlying BRNG integer recurrence so that all bits in 64-bit chunks are uniformly distributed.

This function does not support the following VS BRNGs:

1. VSL_BRNG_MCG31
2. VSL_BRNG_R250
3. VSL_BRNG_MRG32K3A
4. VSL_BRNG_WH
5. VSL_BRNG_SOBOL
6. VSL_BRNG_NIEDERR
7. VSL_BRNG_IABSTRACT
8. VSL_BRNG_DABSTRACT
9. VSL_BRNG_SABSTRACT

7.4.5. Bernoulli (VSL_RNG_METHOD_BERNOULLI_ICDF)

Bernoulli distribution with parameter *p*. You may generate any successive random number *k* of the Bernoulli distribution by the formula:

$$k = \begin{cases} 1, & u \le p \\ 0, & u > p \end{cases},$$

where *u* is a successive random number of a uniform distribution over the interval [0, 1).

See Intel® MKL Vector Statistics Random Number Generator Performance Data for test results summary.

7.4.6. Geometric (VSL_RNG_METHOD_GEOMETRIC_ICDF)

Geometrical distribution with parameter *p*. You may generate any successive random number *k* of the geometrical distribution by the formula:

$$k = \left\lfloor \frac{\ln u}{\ln(1-p)} \right\rfloor,$$

where *u* is a successive random number of a uniform distribution over the interval [0, 1).

See Intel® MKL Vector Statistics Random Number Generator Performance Data for test results summary.

7.4.7. Binomial (VSL_RNG_METHOD_BINOMIAL_BTPE)

Binomial distribution with parameters `ntrial` and *p*. If ntrial·min(*p*,1 - *p*) ≥ 30, random numbers of the binomial distribution are generated by the BTPE method (see [Kach88] for details). Otherwise, a combination of inverse transformation and table lookup methods is used. The BTPE method is a variation of the acceptance/rejection method that uses linear (on the fractions close to the distribution mode) and exponential (at the distribution tails) functions as majorizing functions. To avoid time-consuming acceptance/rejection checks, areas with zero probability of rejection are introduced and a squeezing technique is applied.

See Intel® MKL Vector Statistics Random Number Generator Performance Data for test results summary and performance graphs.

7.4.8. Hypergeometric (VSL_RNG_METHOD_HYPERGEOMETRIC_H2PE)

Hypergeometric distribution with parameters l, s, and m. If $M - k_L > 40$ and $k_L < k_H$, where $M = \lfloor \min(s + 1, l - s + 1) \cdot \min(m + 1, l - m + 1)/(l + 2) \rfloor$, $k_L = \max(0, \min(s.l - s) - (\max(m, l - m))$, $k_H = \min(\min(m, l - m), \min(s, l - s))$, the random numbers are generated by the H2PE method (see [Kach85] for details). Otherwise, they are produced using the inverse transformation method in combination with the table lookup method. The H2PE method is a variation of the acceptance/rejection method that uses constant (on the fraction close to the distribution mode) and exponential (at the distribution tails) functions as majorizing functions. To avoid time-consuming acceptance/rejection checks, a squeezing technique is applied.

See Intel® MKL Vector Statistics Random Number Generator Performance Data for test results summary and performance graphs.

7.4.9. Poisson (VSL_RNG_METHOD_POISSON_PTPE)

Poisson distribution with parameter Λ. If $\Lambda \geq 27$, random numbers are generated by PTPE method (see [Schmeiser81] for details). Otherwise, a combination of inverse transformation and table lookup methods is used. The PTPE method is a variation of the acceptance/rejection method that uses linear (on the fraction close to the distribution mode) and exponential (at the distribution tails) functions as majorizing functions. To avoid time-consuming acceptance/rejection checks, areas with zero probability of rejection are introduced and a squeezing technique is applied.

See Intel® MKL Vector Statistics Random Number Generator Performance Data for test results summary and performance graphs.

7.4.10. Poisson (VSL_RNG_METHOD_POISSON_POISNORM)

Poisson distribution with parameter Λ. If $\Lambda < 1$, the random numbers are generated by combination of inverse transformation and table lookup methods. Otherwise, they are produced through transformation of the normally distributed random numbers.

The VSL_RNG_METHOD_GAUSSIAN_BOXMULLER2 method is used to generate random numbers of normal distribution.

See Intel® MKL Vector Statistics Random Number Generator Performance Data for test results summary and performance graphs.

7.4.11. PoissonV (VSL_RNG_METHOD_POISSONV_POISNORM)

Poisson distribution with parameter Λ. If $\Lambda < 0.0625$, the random numbers are generated by inverse transformation method. Otherwise, they are produced through transformation of normally distributed random numbers.

The VSL_RNG_METHOD_GAUSSIAN_BOXMULLER2 method is used to generate random numbers of normal distribution.

See Intel® MKL Vector Statistics Random Number Generator Performance Data for test results summary and performance graphs.

7.4.12. NegBinomial (VSL_RNG_METHOD_NEGBINOMIAL_NBAR)

Negative binomial distribution with parameters *a* and *p*. If $(a - 1)(1 - p)/p \geq 100$, the random numbers are generated by NBAR method. Otherwise, the random numbers are generated by combination of inverse transformation and table lookup methods. The NBAR method is a variation of the acceptance/rejection method that uses constant and linear functions (on the fraction close to the distribution mode) and exponential functions (at the distribution tails) as majorizing functions. To ensure that the majorizing functions are close to the normalized probability mass function, five 2D figures are formed from the majorizing and minorizing functions as well as from other auxiliary curves. To avoid time-consuming acceptance/rejection checks, areas with zero probability of rejection are introduced.

See Intel® MKL Vector Statistics Random Number Generator Performance Data for test results summary and performance graphs.

7.4.13. Multinomial (VSL_RNG_METHOD_MULTINOMIAL_MULTPOISSON)

Multinomial distribution with parameters *m*, *k*, and a probability vector *p*. Random numbers of the multinomial distribution are generated by Poisson Approximation method (see [Charles93] for details).

1. In the first stage, *k* independent Poisson values $(X_1 \ldots X_k)$ are generated by the POISSNORM method.
2. Let m^* denote sum of the generated *k* Poisson variates:
   - If $m^*=m$, the first-stage sample has the required distribution.
   - If $m^*>m$, the sample is discarded and the first stage is repeated.
   - If $m^*<m$, m^*-m observations are generated by the Direct method (see [Charles93] for details):
     a. m^*-m uniformly distributed independent random variates U_i are generated on the interval (0, 1).
     b. The component X_i is incremented by 1 if

        $$p_{i-1}^* < U_i \leq p_i^*, \text{where } p_i^*, = \sum_{j=1}^{i} p_j \, , p_0^* = 0$$

See Intel® MKL Vector Statistics Random Number Generator Performance Data for test results summary and performance graphs.

8. Bibliography

[Ant79] Antonov, I.A., and Saleev, V.M. An economic method of computing LPt-sequences. USSR Comput. Math. Math. Phys., 19, 252-256, 1979.

[Atkin79] Atkinson A.C. A family of switching algorithms for the computer generation of beta random variables, Biometrika, 66, 1, 141-145, 1979.

[AVX] Intel® Advanced Vector Extensions Programming Reference, http://software.intel.com/file/36945

[BMT] Intel Bull Mountain Technology, Software Implementation Kit (SIK), http://software.intel.com/file/37157

[Box58] Box, G. E. P. and Muller, M. E. A Note on the Generation of Random Normal Deviates. Ann. Math. Stat. 28, 610-611, 1958.

[Brat87] Bratley, P., Fox, B.L., and Schrage, L.E.. A Guide to Simulation, 2nd Edition, Springer-Verlag, New York, 1987.

[Brat88] Bratley, P. and Fox, B.L. ALGORITHM 659: Implementing Sobol's Quasirandom Sequence Generator. ACM Transactions on Modeling and Computer Simulation, Vol. 14, No. 1, 88-100, March 1988.

[Brat92] Bratley, P., Fox, B.L., and Niederreiter, H. Implementation and Tests of Low-Discrepancy Sequences. ACM Transactions on Modeling and Computer Simulation, Vol. 2, No. 3, 195-213, July 1992.

[Charles93] Charles S. Davis, The computer generation of multinomial random variates. Computational Statistics & Data Analysis, V. 16, 205-217, Aug. 1993.

[Cheng78] Cheng, R. C. H., Generating Beta variates with Nonintegral Shape Parameters, Communications of the ACM, 21, 4, 317-322, 1978.

[Cram46] Cramer, H. Mathematical Methods of Statistics. Cambridge, 1946.

[Dev86] Devroye, L. Non-Uniform Random Variate Generation, Springer-Verlag, New York, 1986.

[Ent98] Entacher, Karl. Bad Subsequences of Well-Known Linear Congruential Pseudorandom Number Generators. ACM Transactions on Modeling and Computer Simulation, Vol. 8, No. 1, 61-70, January 1998.

[FIPS-197] Federal Information Processing Standards Publication 197, ADVANCED ENCRYPTION STANDARD (AES)

[Haram08] Haramoto H., Matsumoto M., Nishimura T., Panneton F., and L'Ecuyer P. Efficient Jump Ahead for F2-Linear Random Number Generators, INFORMS Journal on Computing, Vol.20, No 3, Summer 2008, pp 385-390.

[IntelSWMan] Intel® 64 and IA-32 Architectures Software Developer's Manual. 3 vols.

(http://www.intel.com/content/www/us/en/processors/architectures-softwaredeveloper-

manuals.html)

[Jöhnk64] Jöhnk, M.D. Erzeugung von Betaverteilten und Gammaverteilten Zufallszahlen, Metrika, 8, 5-15, 1964.

[Jun99] Jun, B., and Kocher, P. The Intel Random Number Generator. White paper prepared for Intel Corp., Cryptography Research, Inc., April 1999.

[Kach88] Kachitvichyanukul, V. and Schmeiser, B.W. Binomial random variate generation. Communications of the ACM, Volume 31, Issue 2, February 1988.

[Kach85] Kachitvichyanukul, V. and Schmeiser, B.W. Computer generation of hypergeometric random variates. J. Stat. Comput. Simul. 22, 1, 127-145, 1985.

[KIM04] Song-Ju Kim, Ken Umeno, and Akio Hasegawa, Corrections of the NIST Statistical Test Suite for Randomness, http://arxiv.org/PS_cache/nlin/pdf/0401/0401040v1.pdf

[Kirk81] Kirkpatrick, S., and E. Stoll. A Very Fast Shift-Register Sequence Random Number Generator. Journal of Computational Physics, V. 40, 517-526, 1981.

[Knuth81] Knuth, Donald E. The Art of Computer Programming, Volume 2, Seminumerical Algorithms, 2nd edition, Addison-Wesley Publishing Company, Reading, Massachusetts, 1981.

[L'Ecu94] L'Ecuyer, Pierre. Uniform Random Number Generators, Annals of Operations Research, 53, 77-120, 1994.

[L'Ecu99] L'Ecuyer, P. Good Parameter Sets for Combined Multiple Recursive Random Number Generators. Operations Research, 47, 1, 159-164, 1999.

[L'Ecuyer99] L'Ecuyer, Pierre. Tables of Linear Congruential Generators of Different Sizes and Good Lattice Structure. Mathematics of Computation, 68, 249-260, 1999.

[MacLaren89] MacLaren, N.M. The Generation of Multiple Independent Sequences of Pseudorandom Numbers. Applied Statistics, 38, 351-359, 1989.

[Mars95] Marsaglia, G. The Marsaglia Random Number CDROM, including the DIEHARD Battery of Tests of Randomness, Department of Statistics, Florida State University, Tallahassee, Florida, 1995.

[Mars2000] Marsaglia, G., and Tsang, W. W. A simple method for generating gamma variables, ACM Transactions on Mathematical Software, Vol. 26, No. 3, Pages 363-372, September 2000.

[Matsum92] Matsumoto, M., and Kurita, Y. Twisted GFSR generators, ACM Transactions on Modeling and Computer Simulation, Vol. 2, No. 3, Pages 179-194, July 1992.

[Matsum94] Matsumoto, M., and Kurita, Y. Twisted GFSR generators II, ACM Transactions on Modeling and Computer Simulation, Vol. 4, No. 3, Pages 254-266, July 1994.

[Matsum98] Matsumoto, M., and Nishumira T. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator, ACM Transactions on Modeling and Computer Simulation, Vol. 8, No. 1, Pages 3-30, January 1998.

[Matsum2000] Matsumoto, M., and Nishimura T. Dynamic Creation of Pseudorandom Number Generators, 56-69, in: Monte Carlo and Quasi-Monte Carlo Methods 1998, Ed. Niederreiter, H. and Spanier, J., Springer 2000, http://www.math.sci.hiroshima-u.ac.jp/%7Em-mat/MT/DC/dc.html.

[Mikh2000] Mikhailov, G.A. Weight Monte Carlo Methods, Novosibirsk: SB RAS Publ., 2000 (In Russian).

[MT2002] http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html

[NAG] Numerical Algorithms Group, www.nag.co.uk.

[NIST800-22] NIST Special Publication 800-22 A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications.

[NIST800-90] NIST Special Publication 800-90 Recommendation for Random Number Generation Using Deterministic Random Bit Generators (Revised), March 2007. http://csrc.nist.gov/publications/nistpubs/800-90/SP800-90revised_March2007.pdf

[Ripley87] Ripley, B.D. Stochastic Simulation, Wiley, New York, 1987.

[Saito08] Saito, M., and Matsumoto, M. *SIMD-oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator*,Monte Carlo and Quasi-Monte Carlo Methods 2006, Springer, pp. 607-622, 2008, http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/earticles.html

[Salmon2011] Salmon, J.K., Moraes M.A., Dror R. O., and Shaw, D.E. New York, 2011. http://www.thesalmons.org/john/random123/papers/random123sc11.pdf

[Schmeiser81] Schmeiser, Bruce, and Kachitvichyanukul, Voratas. Poisson Random Variate Generation. Research Memorandum 81-4, School of Industrial Engineering, Purdue University, 1981.

[Vad77] Vaduva, I. On computer generation of gamma random variables by rejection and composition procedures. Mathematische Operationsforschung und Statistik, Series Statistics, vol. 8, 545-576, 1977.

[Ziff98] Ziff, Robert M. Four-tap shift-register-sequence random-number generators. Computers in Physics, Vol. 12, No. 4, Jul/Aug 1998.