Intel® Math Kernel Library

Vector Statistical Library Notes

Document Number: 310713-009US

World Wide Web: http://developer.intel.com

Version	Version Information	Date
1.0	Original version of the VSL Notes. Documents Intel® Math Kernel Library release 6.0 Gold	02/03
2.0	Original release of the VSL Notes. Documents Intel Math Kernel Library release 6.0 Gold.	03/03
3.0	Documents Intel Math Kernel Library release 6.1.	07/03
4.0	Documents Intel Math Kernel Library release 7.0 Beta.	11/03
5.0	Documents Intel Math Kernel Library release 7.0 Gold.	04/04
6.0	Documents Intel Math Kernel Library release 7.0.1.	07/04
7.0	Documents Intel Math Kernel Library release 8.0 Beta.	03/05
8.0	Documents Intel Math Kernel Library release 8.0 Gold.	08/05
-009	Documents Intel Math Kernel Library release 8.1 Gold.	03/06

Table of Contents

About This Library

Vector Statistical Library (VSL) is designed for the purpose of pseudorandom and quasi-random vector generation and for convolution and correlation mathematical operations. VSL is an integral part of Intel® Math Kernel Library (Intel® MKL).

VSL provides a number of generator subroutines implementing commonly used continuous and discrete distributions, all of which are based on the highly optimized Basic Random Number Generators (BRNGs) and VML, the library of vector transcendental functions, to help improve their performance.

About This Document

This document includes a brief conceptual overview of random numbers generation problems, the product and its capabilities, with focus on interpretation of results and the related generator figures of merit as well as task-oriented, procedural, and reference information. In contrast to Intel MKL Reference Manual, VSL Notes substantially expand on the concept of random number generation and its application as well as on the related notions and issues. The document provides extensive comparative analysis of the library generators and describes the basic tests applied. Apart from the VSL distribution generators and service subroutines, dealt with in the Intel MKL Reference Manual, the VSL Notes also describe testing of distribution generators.

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

VSL Structure section covers the concept underlying VSL, the library structure and potential for functionality enhancement. VSL is a library of high-performance random number generators. The section describes the factors that optimize the VSL generators for Intel® processors. Special attention is given to VSL ease of use and other advantages in parallel programming.

Testing of Basic Random Number Generators and Testing of Distribution Random Number Generators describe a number of tests for the VSL generators of various probability distributions. See http://www.intel.com/software/products/mkl for latest test results.

Conventions

The following mathematical notation is used throughout the document:

$\oplus$	Bitwise exclusive OR.
&	Bitwise AND.
|	Bitwise OR.

Introduction

This document does not purport to cover the fundamentals of mathematical statistics and probability theory, nor those of the theory of numbers and statistical simulation. Books and articles listed in the Bibliography section mostly cover these issues. What you will find below is a brief overview of issues pertaining to random number generation, interpretation of the results and

the related notion of quality random number generation. To some extent, it is an attempt to justify 'the fall' of many people engaged in solving problems of randomness simulation, that is, the fall John von Neumann meant, when he wrote: "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin". (Still more and more researchers in a variety of scientific fields are getting themselves involved into this kind of simulation depravity, as simulation is becoming more and more valuable in various scientific disciplines). Computer simulation has become a new and de-facto 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 has come to be more dependable exclusively on our own understanding of phenomena we try to model. This is arguably the key factor in ensuring the great success that computer simulation has achieved of recent.

Randomness and Scientific Experiment

A precise definition of what the word 'random' means can hardly be given, even considering the fact that everyday life provides a variety of examples of 'randomness'. Randomness is closely related to unpredictability of observation results and impossibility to predict them with sufficient accuracy. The nature of randomness is based on lack of exhaustive information about the phenomenon under observation. As soon as we learn the origin of that phenomenon, we 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. We may characterize randomness as a type of relation stipulated by conditions that are inessential, superfluous, and extraneous to this particular phenomenon. Thus, knowledge is incomplete by definition as it is impossible to allow for all sorts of immaterial relations.

Since our knowledge is incomplete (and it is something that can hardly be helped), 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 our instruments, but these small changes may cause significant alterations in the final results. 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 hypothetically "true value".

Nevertheless, with all likelihood of 'irregularities' and 'deviations', observational or experimental results still reveal a certain typical regularity, named statistical stability. Various forms of statistical stability are formulated as specific rules that mathematical statistics calls *laws of large numbers*. In fact, it is this stability that the mathematical theory underlying mathematical model of random phenomena is based upon. This theory is well known as the theory of probability.

Random Numbers

A set of distinctive features characterizes experimental observations. Many of such features are of purely quantitative nature (results of measurements, calculations, and the like) but some of them are mainly qualitative (for example, color of the object, occurrence or non-occurrence, and so on). In the latter case results may also be presented as quantitative if some appropriate conventions have been developed and applied (this may prove to be a rather tricky task to accomplish, though). Thus, even when the result is a particular quality feature it can be expressed by a certain number, which, if the result is a random phenomenon, is called a random number.

Numerical methods consider random numbers not only as data from experimental observations. After emergence of computers an imitation of a huge amount of random numbers is of great interest in various computational areas as well [Knuth81].

For historical reasons methods that utilize random numbers to perform a simulation of phenomena are called Monte Carlo methods. Monte Carlo became a tool to perform the most complex simulations in natural and social sciences, financial analysis, physics of turbulence, rarefied gas and fluid simulations, physics of high energies, chemical kinetics and combustion, radiation transport problems, and photorealistic rendering.

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

Due to 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 can not be considered truly random as they only imitate randomness and are, in fact, pseudo-random. Ideally, pseudo-random numbers imitate ‘truly’ random ones so well that without knowing the method of pseudo-random number generation and judging only by the output sequence, it is impossible to 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 a number of practical applications do not require strict unpredictability. However, there are certain applications for which most now existing pseudorandom generators are useless and at times simply dangerous. Among them, for example, are applications dealing with geometrical behavior of large random vectors. Most of presently existing 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. For such methods (a numerical integration and optimization, for example) the most important is to fill some space with numbers as close to a given distribution as possible to the prejudice of independence. Such sequences do not look random at all. For historical reasons they are called quasi-random (or low discrepancy) sequences, respective generators are called quasi-random number generators, and Monte Carlo methods dealing with quasi-random numbers are called Quasi-Monte Carlo methods.

Hereinafter, the term ‘random number generator’, or RNG, refers to both pseudo- and quasi-random number generators, unless we want to emphasize the fact that a generator produces precisely a pseudo- or quasi-random sequence.

Figures of Merit for Random Number Generators

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

When considering a great variety of probability distributions, special emphasis should be laid upon a uniform distribution over a certain set U of large cardinality. Firstly, such a distribution is most convenient for analysis. And secondly, a random number generator of uniform distribution can

always serve as a basis for an RNG of any other distribution type. That is why we use the term *basic generators* in reference to pseudorandom number generators of uniform distribution.

So the observational output sequence of a basic generator should ideally possess the same properties as a sequence of independent variates evenly distributed over a set *U*, that is, it should be able to pass various statistical tests for uniformity and independence. A pseudorandom number generator, however, is unable to pass all sorts of statistical tests, as it is an a priori fact that the output sequence of such generator is anything but random. In other words, a fairly powerful statistical test can always be created for any individual basic RNG, which the said generator will definitely fail. The situation may not look so desperate, if we consider the time required to detect 'non-randomness' in the generator. It makes sense to consider only those statistical tests that work within a 'reasonable' period of time. What exactly time period is 'reasonable'? No direct answer is possible here, as it depends on the sphere of generator application. For example, 'reasonable' time in cryptography may be measured in years of testing conducted on a powerful cluster, while it may be significantly shorter for most of other applications.

Note: *As of present, VSL contains general-purpose random number generators that are not intended for cryptography applications!*

Cryptographic RNGs are too slow for other fields; most of applications there benefit from simpler (and faster) generators: linear congruential, multiple recursive, feedback-shift-register, add-with-carry, etc.

To summarize, it should be noted that checking the quality of basic RNGs requires a 'reasonable' set, or battery, of statistical tests. Ideally, such tests depend for their choice on types of problems the generator is intended to solve. A suitable test battery for general-purpose RNGs 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. Still a specific application type may require a more complete generator testing.

While duly recognizing the importance and usefulness of empirical testing, we should emphasize the significance of theoretical methods for estimating the quality of basic generators. Theoretical research serves as the basis for better understanding of generator's properties: its period length, lattice structure, discrepancy, equidistribution, etc. Theoretic evaluation is the first stage in rejecting admittedly bad generators. Empirical tests should be applied only to make sure the remaining generators are of acceptable quality. What makes the empirical testing just as important is the fact that most of results obtained with the help of theoretical testing refer to a basic generator used over the entire period, while in practice only a small fraction of the period is (and should be!) engaged. Good behavior of *k*-dimensional random number vectors over the entire period provides us with greater confidence (yet not with a proof) that similarly good statistical behavior will be observed over a smaller portion of the period [L'Ecu94].

Period of a basic generator is a most important feature that characterizes its quality. For example, one of the VSL BRNGs — multiplicative congruential generator MCG31m1 — has a period length of about 2^{31}, while its efficiency amounts to about four processor cycles per one real number, using Intel® Itanium® 2 processor. Therefore, with the processor frequency of 1GHz, the entire period will be covered within slightly more than 2 seconds. Taking into consideration 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) we may assert that such period length is unacceptable. Such generators, however, still may be useful in certain Monte Carlo applications (mostly due to the speed and small volume of memory engaged to keep the generator state as well as efficient methods available for generation of random subsequences), when a relatively little quantity of random numbers should be used. For example, while estimating

a global solution to an integral equation through Monte Carlo method, the same random numbers should be used for different parameters [Mikh2000]. Somehow or other, modern computational capacities require BRNGs of at least 2^{60} period length. All the other VSL BRNGs meet these requirements.

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

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

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

For the same reasons quasi-random number generators filling some hypercube as evenly as possible are called in VSL as Basic Random Number Generators as well. Quasi-random sequences filling space according to a non-uniform distribution can be generated by transforming a sequence produced by a basic quasi-random number generator. It is obvious that 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.

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

First and foremost, it should be noted that a general distribution generator greatly depends on the quality of the underlying BRNG. Several basic approaches may be singled out to test general distribution generators.

Random number distributions can be described with a number of measures: probability moments, central and absolute moments, quantiles, mode, scattering, skewness, and excess (kurtosis) coefficients, etc. 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. Somehow or other, when limit probability distribution is known, it is possible to build a statistical test to check whether a particular sample characteristic agrees with a corresponding measure of the distribution.

Of greatest practical value for simulation purposes are sample mean and variance that are main properties of the distribution bias and scattering. All the VSL 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 among them are *chi*-square Pearson goodness-of-fit test (for discrete and continuous distributions) and Kolmogorov-Smirnov goodness-of-fit test (for

continuous distributions). Every VSL 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 in fact real Monte Carlo applications. Their choice is quite optional and should be made in accordance with the generator's field of application, the only requirement being an opportunity to verify the results obtained against the theoretical value.
A good example of such test application, which is used in checking the VSL generators for quality, is the self-avoiding random walk [Ziff98].

VSL Structure

The VSL library of the current Intel MKL version contains a set of generators to create general probability distributions, most commonly used in simulations, such as uniform, normal (Gaussian), exponential, Poisson, etc. Non-uniform distributions are generated using various transformation techniques applied to the output of a basic (either pseudo-random or quasi-random) RNG.

To generate random numbers of a given probability distribution, you have an option of choosing one of the available VSL basic generators or of registering your own basic random number generator. To enhance their performance, all the VSL BRNGs are highly optimized for various architectures of Intel processors. Besides, VSL provides a number of different techniques for transforming uniformly distributed random numbers into a sequence of required distribution.

All the random number generators that are implemented in VSL are of vector type. Unlike scalar type generators, for example, a standard `rand()` function, when the function output is a successive random number, vector generators produce a vector of *n* successive random numbers of a given distribution with given parameters.

VSL is a thread-safe library convenient for parallel computing with a great variety of configurations of parallel systems. A *random stream* is a basic notion in VSL. Mechanism of streams provides simultaneous generation of several random number sequences produced by one or more basic generators, 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.

Why Vector Type Generators?

Due to architectural features of modern computers vector type library subroutines often perform much more efficiently than scalar type routines. In other words, the overhead expenses are often comparable with the total time required for computations. Certainly, there are subroutines where overhead expenses are negligible in comparison with the total time required for computation. However, this is not usually the case with highly optimized RNGs. To reduce overhead expenses, all VSL random number generator subroutines are of vector type. User is free to call a vector random number generator subroutine to generate just one random number, however, such use is hardly efficient.

On the one hand, vector type random number generators sometimes require more careful programming. A reward in this case is a substantial speedup in overall application performance.

On the other hand, VSL provides a number of services to make vector programming as natural as possible. See Independent Streams. Leapfrogging and Block-Splitting section and Abstract Basic Random Number Generators. Abstarct Streams section for further discussion.

Disregarding possible programming issues, the vector type interface is quite natural for Monte Carlo methods because Monte Carlo requires a lot of random numbers rather than just one.

Basic Generators

As indicated above, the basic generators may serve to obtain random numbers of various statistical distributions. Non-uniform distribution generators strongly depend on the quality of the underlying basic generators. Besides, as we have already mentioned, at present there is no such basic generator that would be fully adequate for any application. Many of the current generators are useless and simply dangerous for a certain category of tasks. In a number of applications quality requirements for RNGs prevail over other requirements, such as speed, memory use, etc. In some other tasks quality requirements are not that stringent and speed criterion or efficiency in generating random number subsequences are of higher importance. Some applications use random numbers as real ones, while others treat random numbers as a bit stream. It should be noted that, even if a basic generator has trouble providing true randomness for lower bits, it is not necessarily inadequate for applications using variates as real numbers.

All of the above arguments testify to the fact that a library of general-purpose RNGs should provide a set of several different basic generators, both pseudo- and quasi-random. Besides, such a library should provide an option of including new basic generators, which you may find preferable. VSL provides a variety of basic pseudo- and quasi-random number generators yet allowing the user to register user-defined basic generators and also utilize random numbers generated externally, for example, from physical source of random numbers [Jun99]. See Abstract Basic Random Number Generators. Abstarct Streams section for details.

One of the important issues for computational experimentation is verification of the results. Typically, a researcher is unable to verify the output since the solution is simply unknown. Without going into details of verification for sophisticated simulation systems, we would state that any verification process involves testing of each structural element of the system. A random number generator, being one of such structural elements, may bring about inadequate results. Therefore, to obtain more reliable results of the experiment, many authors recommend that several different basic generators should be used in a series of computational experiments. This is yet another argument favoring inclusion of several BRNGs of different types in a library.

VSL provides the following basic pseudorandom number generators:

- **MCG31m1**. A 31-bit multiplicative congruential generator.

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

- **R250**. A generalized feedback shift register generator.

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

- **MRG32k3a**. A combined multiple recursive generator with two components of order 3.

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

- **MCG59**. A 59-bit multiplicative congruential generator.

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

- **WH**. A set of 273 Wichmann-Hill combined multiplicative congruential generators. ($j = 1, 2, \ldots, 273$)

$$x_n = a_{1,j} x_{n-1} (\mathrm{mod}\, m_{1,j})$$
$$y_n = a_{2,j} y_{n-1} (\mathrm{mod}\, m_{2,j})$$
$$z_n = a_{3,j} z_{n-1} (\mathrm{mod}\, m_{3,j})$$
$$w_n = a_{4,j} w_{n-1} (\mathrm{mod}\, m_{4,j})$$
$$u_n = \left(x_n / m_{1,j} + y_n / m_{2,j} + z_n / m_{3,j} + w_n / m_{4,j} \right) \mathrm{mod} 1$$

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**. Mersenne Twister pseudorandom number generator.

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

$$y_n = x_n,$$

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

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

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

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

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

`k = 624, m = 397, w = 11, s = 7, t = 15, l = 18, b = 0x9D2C5680,`
`c = 0xEFC60000, p = 0x80000000,`

`q = 0x7FFFFFFF,`

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

$$A = \begin{vmatrix} 0 & 1 & 0 & & \\ 0 & 0 & \ldots & 0 & \\ & & & \ldots & \\ & & 0 & 0 & 1 \\ a_{31} & a_{30} & \ldots & \ldots & a_0 \end{vmatrix},$$

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

- **MT2203**. A set of 1024 Mersenne-Twister pseudorandom number generators (j = 1,...,1024).

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

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

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

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

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

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

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

k = 69, m = 34, w = 12, s = 7, t = 15, l = 18, p = 0xFFFFFFE0,

q = 0x1F,

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

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

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

In addition, two basic quasi-random number generators are available in VSL.

- **SOBOL** (with Antonov-Saleev [Ant79] modification). A 32-bit Gray code-based generator producing low-discrepancy sequences for dimensions $1 \le s \le 40$.

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

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

Note: *The value c is the rightmost zero bit in n-1;* $\mathbf{x}_n$ *is an s-dimensional vector of 32-bit values. The s-dimensional vectors (calculated during random stream initialization)* $\mathbf{v}_i, i = \overline{1,32}$ *are called direction numbers. The vector* $\mathbf{u}_n$ *is the generator output normalized to the unit hypercube* $(0,1)^s$.

- **NIEDERREITER** (with Antonov-Saleev [Ant79] modification). A 32-bit Gray code-based generator producing low-discrepancy sequences for dimensions $1 \le s \le 318$.

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

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

- **ABSTRACT.** Abstract source of random numbers. See Abstract Basic Random Number Generators. Abstract Streams section for details.

Below we discuss each basic generator in more detail and provide references for further reading.

MCG31m1

32-bit linear congruential generators, which also include MCG31m1 [L'Ecuyer99], are still used as default RNGs in various systems mostly due to 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 basic random number generators. Nevertheless, MCG31m1 possesses good statistical properties and may be used to advantage in generating random numbers of various distribution types for relatively small samplings.

R250

R250 is a generalized feedback shift register generator. Feedback shift register generators possess 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. It is common in the field of physics. However, the generator fails a number of tests, a 2D self-avoiding random walk [Ziff98] being an example.

MRG32k3a

A combined generator MRG32k3a [L'Ecu99] meets the requirements for modern RNGs: good multidimensional uniformity, fairly large period, etc. Besides, being optimized for various Intel® architectures, this generator rivals the other VSL BRNGs in speed.

MCG59

A multiplicative congruential generator MCG59 is one of the two basic generators implemented in NAG Numerical Libraries [NAG] (see www.nag.co.uk). Since the module of this generator is not prime, its period length is not 2^{59}, but just 2^{57}, if the seed is an odd number. A drawback of such generators is well-known (for example, see [Knuth81], [L'Ecu94]): the lower bits of the output sequence are not random, therefore breaking numbers down into their bit patterns and using individual bits may cause trouble. Besides, block-splitting of the sequence over the entire period into 2^d similar blocks results in full coincidence of such blocks in d lower bits (see, for instance, [Knuth81], [L'Ecu94]).

WH

WH is a set of 273 different basic generators. It is the second basic generator in NAG libraries. The constants $a_{i,j}$ are in the range 112 to 127 and the constants $m_{i,j}$ are prime numbers in the range 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 Wichmann–Hill generator would be at least 2^{92}, if it were 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}. Further discussion of the properties of these generators is given in [MacLaren89], which shows that the generated pseudo-random sequences are essentially independent of one another according to the spectral test.

MT19937

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}-1 and 623-dimensional equidistribution up to 32-bit accuracy) make this generator applicable for simulations in various fields of science and engineering. Initialization procedure is essentially the same as described in [MT2002].

MT2203

The set of 1024 MT2203 pseudorandom number generators is an addition to MT19937 generator intended for application 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}-1 and possesses 68-dimensional equidistribution up to 32-bit accuracy. Initialization procedure is essentially the same as described in [MT2002].

SOBOL

Bratley and Fox [Brat88] provide an implementation of the Sobol quasi-random number generator. VSL implementation allows generating Sobol's low-discrepancy sequences of length up to 2^{32}. The dimension of quasi-random vectors can vary from 1 to 40 inclusive.

NIEDERREITER

According to the results of Bratley, Fox, and Niederreiter [Brat92] Niederreiter's sequences have the best known theoretical asymptotic properties. VSL implementation allows generating Niederreiter's low-discrepancy sequences of length up to 2^{32}. The dimension of quasi-random vectors can vary from 1 to 318 inclusive.

VSL provides an option of registering one or more new basic generators that you see as preferable or more reliable. Use them in the same way as the BRNGs available with VSL. The registration procedure makes it easy to include a variety of user-designed generators.

ABSTRACT

Abstract basic generators are designed to allow VSL distribution generators to be used with underlying uniform random numbers that are already generated. There are several cases when this feature might be useful:

- random numbers of the uniform distribution are generated externally [Mars95] (for example, in physical device [Jun99]);

- 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. Due to specificity of abstract basic generators, `vslNewStream` and `vslNewStreamEx` functions cannot be used to create abstract streams. Special `vsliNewAbstractStream`, `vslsNewAbstractStream`, and `vsldNewAbstractStream` functions are provided to initialize integer, single precision, and double precision abstract streams respectively.

Each of the VSL basic generators consists of 4 subroutines:

- **Stream Initialization Subroutine.** See the section Random Streams and RNGs in Parallel Computation for details.
- **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.
- **Single Precision Floating-Point Random Number Vector Generation Subroutine.** The subroutine generates a real arithmetic vector of uniform distribution over the interval [a, b].
- **Double Precision Floating-Point Random Number Vector Generation Subroutine.** The subroutine generates a real arithmetic vector of uniform distribution over the interval [a, b].

Random Streams and RNGs in Parallel Computation

Initializing Basic Generator

To obtain a random number sequence from a given basic generator, you should assign initial, or seed values. The assigning procedure is called the generator initialization (the C language function analogous with the initialization function is `srand(seed))` in `stdlib.h`). Different types of basic generators 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. In contrast to the pseudorandom number generators, quasi-random generators require the dimension parameter on input. Thus, each BRNG, including those registered by the user, requires an individual initialization function. However, requiring individual initialization functions within the library interface would limit the versatility of the routines.

The basic concept of VSL is to provide an interface with universal mechanism for generator initialization, while encapsulating details of the initialization process from the user. (Nevertheless, the initialization process is clearly documented in VSL Notes for each library basic generator). In line with this concept, VSL offers two subroutines to initialize any basic generator (see the functions of random stream creation and initialization in Random Streams section). These initialization functions can also be used to initialize user-supplied functions. One of the subroutines initializes a given basic generator using one 32-bit initial value, which is called the seed by tradition. If the generator requires more than one 32-bit seed, VSL 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 basic generator by passing an array of *n* 32-bit initial values. If the number of the initial values *n* is insufficient to initialize a given basic generator, the missing initial values are initialized by default values. On the contrary, if the number of the initial values *n* is excessive, the redundant values are ignored. For details on initialization procedure see Basic Random Generator Properties and Testing Results.

When calling initialization functions you may ignore acceptability of the passed initial values for a given basic generator. If the passed seeds are unacceptable, the initialization procedure replaces them with those acceptable 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 basic generator to VSL, you should implement an appropriate initialization function, which supports the above mechanism of initial values passing, and, if required, apply the leapfrog and block-splitting techniques.

Creating and Initializing Random Streams

VSL assumes that at any moment during the program operation you may simultaneously use several random number subsequences generated by one or more basic generators. Consider the following scenarios:

- The simulation system has several independent structural blocks of random number generation (for example, one block generates random numbers of normal distribution, another generates uniformly distributed numbers, etc.) 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.
- It is necessary 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].
- 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 basic generator and assigning specific initial values we predetermine the random number sequence produced by this particular stream. In VSL 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 basic generator and its current state as well as some additional data necessary for the leapfrog and/or skip-ahead method. VSL 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 one 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 the pointer to *stream*, that is, the stream state descriptor.

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 basic generator.

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 basic generator, 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 the 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, the descriptor can no longer be used.

Creating Random Stream Copy and Copying Stream State

VSL provides an option of producing an exact copy of a generated stream by calling 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* will be exactly the same as that of *srcstream*, and both the streams will generate random numbers using the same basic generator.

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 basic generator. The function vslCopyStreamState copies the information about the current stream state from *srcstream* into *deststream*. Other stream-related information remains unchanged.

Saving and Restoring Random Streams

Typically, to get one more correct decimal digit in Monte Carlo, you need to increase the sample by a 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 so as be able to continue computation using that result in case the application is terminated intentionally or abnormally.

In the case of basic generators, 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, VSL 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 *fname* parameter. In `vslSaveStreamF` function a valid random stream to be written is specified by 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. Non-negative value indicates an error.

Independent Streams. Leapfrogging and Block-Splitting

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.

The independence of streams is provided through a number of methods. We discuss three of them, all supported by VSL, in more detail.

- For each of the streams 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 output random number sequences. Mersenne Twister generator is a good example here. It has 1024 parameter sets, which ensure that the resulting subsequences are independent (see [Matsum2000] for details). Another example is WH generator capable of creating up to 273 random number streams. The produced sequences are independent according to the spectral test (see [Knuth81] for the spectral test details).
- 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.
- 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 $x_1, x_{k+1}, x_{2k+1}, x_{3k+1}, \ldots$, the second stream would generate the random numbers $x_2, x_{k+2}, x_{2k+2}, x_{3k+2}, \ldots$, and, finally, the k^{th} stream would generate the random numbers $x_k, x_{2k}, x_{3k}, \ldots$ This method is known as leapfrogging. Note, however, that multidimensional uniformity properties of each subsequence deteriorate seriously as k grows. The method may be recommended if k is fairly small.

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

VSL allows you to use any of the above methods, leapfrog and skip-ahead (block-split) methods deserving special attention.

VSL implements block-splitting through the function `vslSkipAheadStream`:

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

The function changes current state of the stream *stream* so that with the further call of the generator the output subsequence would begin 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, the following sequence of operations should be implemented:

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 );
}
```

VSL implements the leapfrog method trough the function `vslLeapfrogStream`:

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

The function changes the stream *stream* so that the further call of the generator would generate 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 that two latter splitting methods make programming with vector random number generators more natural and easy not only in parallel applications but in a sequential programs as well.

Not all VSL BRNGs support both the methods of generating independent subsequences. Leapfrog (or Skip-Ahead) 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 which BRNG supports what methods:

BRNG	Leapfrog	Skip-Ahead
MCG31m1	supported	supported
R250	not supported	not supported
MRG32k3a	not supported	supported
MCG59	supported	supported
WH	supported	supported
MT19937	not supported	not supported
MT2203	not supported	not supported
SOBOL	supported to pick out individual components of quasi-random vectors	supported
NIEDERREITER	supported to pick out individual components of quasi-random vectors	supported
?ABSTRACT	not supported	not supported

To initialize `nstreams` independent streams for MT2203 set of generators, the following code sequence may be written:

```
…
#define nstreams 1024
…
VSLStreamStatePtr stream[nstreams];
int k;
for ( k=0; k< nstreams; k++ )
{
  vslNewStream( &stream[k], VSL_BRNG_MT2203+k, seed );
}
…
```

Abstract Basic Random Number Generators. Abstract Streams

If you have a preferable basic random number generator, which is not included in VSL, but still want to use VSL distribution generators, you can register such a generator using `vslRegisterBrng` function. In this case your own basic generator should meet VSL BRNG interface requirements. It is not a problem in most cases, but you can also use VSL abstract basic random number generators as a wrapper.

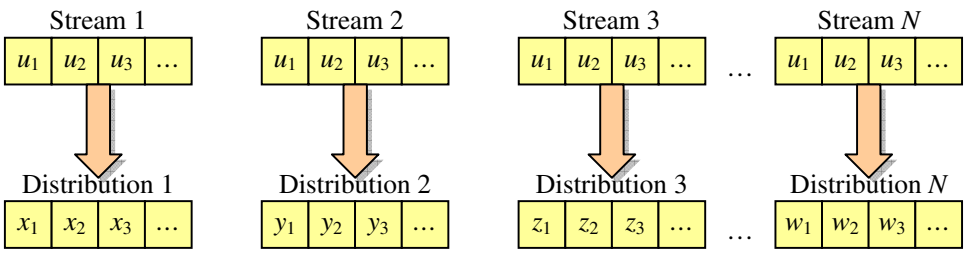
Abstract basic generators are useful when you need to store random numbers in a buffer first and then use these numbers in distribution transformations. The reasons might be:

- Random numbers are read from a file into a buffer.
- Random numbers are taken from a physical device that stores numbers into a buffer.
- You study the system's behavior under different distribution generator parameters using the same BRNG sequence for each parameter set.

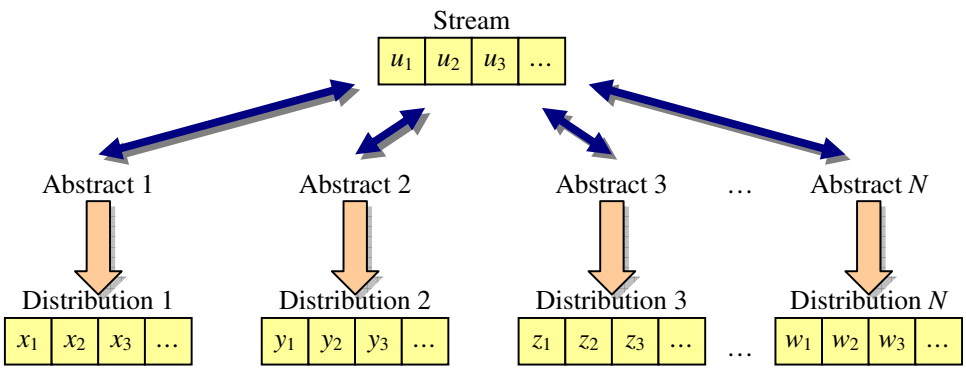
- Your algorithm is essentially sequential but you still want to use a vector random number generator.

While the first two items do not require further discussion, we will focus on the latter two items.

System is being studied under different parameters using the same BRNG sequence. One of the options is to create as many identical streams as many parameters you want to study. Each of these streams is used with a particular distribution generator parameter set. The drawback is that a basic random number generator generates underlying uniform sequence as many times as many distribution parameters are studied. See the following diagram for illustration:



Abstract basic random number generators and respective abstract random streams help you overcome such a drawback. Underlying uniform sequence is generated only once and 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:



Algorithm is essentially sequential. How to utilize vector random number generators? One of typical situations in Monte Carlo methods is when 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 */
```

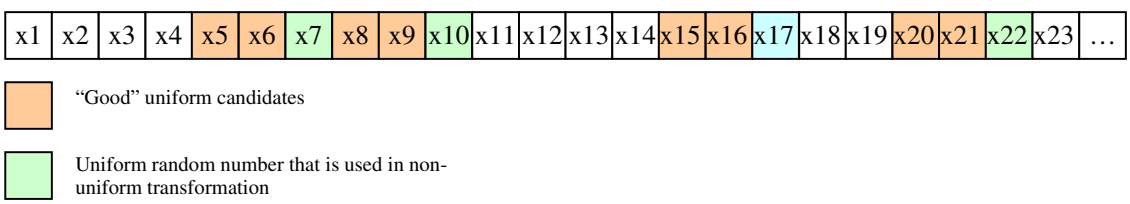
```
    w := Nonuniform() // get successive uniform random number from BRNG
                      // and transform it to non-uniform random number

    /* 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 the modern processor architectures. In particular, this means that you should try to generate and process random numbers as vectors rather than as scalars:

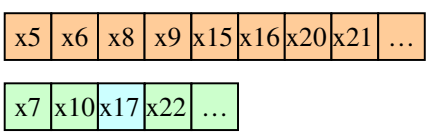
1. Generate vector *U* of pairs (*u*, *v*)
2. Applying "good candidate" criterion $f(u,v)>a$ form new vector *V*, that consists of "good" candidates only.
3. Get vector *W* of non-uniform random numbers *w*.
4. Get vector *R* of results $g(u,v,w)$.

Note that steps 1- 4 do not preserve the original order of 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 non-uniform. So underlying uniform random numbers are utilized as follows:



To keep the original order of underlying uniform random number utilization, yet applying 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 VSL 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

VSL provides three types of abstract basic random number generators intended for:

- integer-valued buffers
- single precision floating-point buffers
- double precision floating-point buffers

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 new abstract stream *stream* and associates it with 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. An obligatory 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*. Note that the buffer is considered as cyclic and index *idx* varies from 0 to *n*-1. Minimal number of buffer entries to be updated is *nmin*. 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 an algorithm specifics allows this.

If you utilize multiple abstract streams, creation of multiple callback functions is not required. Instead, you may have one callback function and distinguish particular abstract stream and particular buffer using *stream* and *buf* parameters respectively.

The callback function should return the quantity of numbers that has 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 basic generator reports an error. It is allowable however to 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 still may be useful if there are no *nmin* numbers by the moment of 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, however, while using abstract streams. For instance, checking the distribution generator status to determine whether the callback function successfully updated buffer or not is a good practice in working with abstract streams. Another important note is that 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 by the moment of abstract stream initialization with `vsl[i,s,d]NewAbstractStream` function call.

Type of the abstract stream to be created should be also chosen carefully. This type depends on a particular distribution generator routine. For instance, all single precision continuous distribution generator routines utilize abstract streams associated with single precision buffers, while double precision distribution generators utilize abstract streams associated with double precision buffers. Most of discrete distribution generators utilize 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 $\lambda = 3$ using an abstract stream. Random numbers are assumed to be uniform integers from 0 to 2^{31}-1 and are stored in the *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_METHOD_IPOISSON_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;
}
```

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.

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 the 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

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

is of $F(x)$ distribution. 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 closed form is often hard to find, while numerical solution to the equation

$F(x) - u = 0$

to calculate x is, as a rule, excessively time consuming.

For discrete distributions the CDF is a step function, the inverse transformation method still being applicable. For simplicity, let us assume that the distribution has probability mass points $k = 0, 1, 2, \ldots$ 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 closed form so that

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

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 $k = \texttt{floor}(G(u))$ 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 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.

Inverse transformation method can be applied to the s-dimensional quasi-random vectors. The resulting quasi-random sequence has the required s-dimensional non-uniform distribution.

Acceptance/Rejection

The cumulative distribution function, let alone the inverse one, is very often 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. We will consider a case of continuous probability distribution, although this technique is just as effective for discrete distributions.

Suppose, we need to generate random numbers x with distribution density $f(x)$. Apart from the variate X, let us 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 $u \le f(y)/cg(y)$, accept y as a random number x with the distribution $F(x)$; else go back to Step 1.

The efficiency of this method greatly depends on 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, great care should be taken when using quasi-random basic generators with acceptance/rejection methods.*

Mixture of Distributions

Sometimes it may be useful to split the initial distribution into several simpler distributions:

$$F(x) = p_1 F_1(x) + p_2 F_2(x) + ... + p_k F_k(x), \quad \sum_{i=1}^{k} p_i = 1,$$

so that random numbers for each of the distributions $F_i(x)$ are easy to generate. Then the 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 most common in the acceptance/rejection method, when for the whole range of acceptable x values a density $g(x)$, which would approximate the 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, great care should be taken when using quasi-random basic generators with mixture methods.*

Special Properties

The most efficient algorithms, though based on the general methods described in the previous sections, should, nevertheless, make use of special properties of distributions, if possible. For example, the inverse transformation method is inapplicable for normal distribution directly. However, use of polar coordinates for a pair of independent normal variates makes it possible to

develop an efficient method of random number generation based on 2D inverse transformation, which is 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 (VSL name is the Box-Muller2 method), when *s* is odd, seems to be problematic 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 (VSL name is the Box-Muller method). In this case to generate *s*-dimensional normally distributed quasi-random numbers, use 2*s*-dimensional quasi-random numbers produced by a basic quasi-random generator.

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

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

For $p > 0.5$, it is convenient to make use of the fact that

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

To summarize, we note that a uniform distribution can be converted to a general distribution by a number of methods. Also, two different transformation techniques implemented for one and the same uniform distribution produce two different sequences of a general distribution, though possessing the same statistical properties.

Let us 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 $F(x) \cdot F(x)$. Thus, on the one hand, the random number x_1 with 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 $x_1 = \mathrm{sqrt}(u_1)$ 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, ... will give two absolutely different sequences x_i.

Transformation into non-uniform distribution sequences may be accomplished in a variety of ways with no fastest or most accurate method existing, as a rule. The inverse transformation method may be preferable over the acceptance/rejection method for some applications and architectures, while reverse preference is common for others. Taking this into account, the VSL interface provides different options of random number generation for one and 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 Functions and Discrete Distribution Functions sections.

Example of VSL Use

A typical algorithm for VSL generators is as follows:

1. Create and initialize stream/streams. Functions `vslNewStream`, `vslNewStreamEx`, `vslCopyStream`, `vslCopyStreamState`, `vslLeapfrogStream`, `vslSkipAheadStream`.
2. Call one or more RNGs.
3. Process the output.
4. Delete the stream/streams. Function `vslDeleteStream`.

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

The following example demonstrates generation of two random streams. The first of them is the output of the basic generator MCG31m1 and the second one is the output of the basic generator R250. The seeds are equal to 1 for each of the streams. The first stream is used to generate 1,000 normally distributed random numbers in blocks of 100 random numbers with parameters $a = 5$ and $sigma = 2$. The second stream is used to produce 1,000 exponentially distributed random numbers in blocks of 100 random numbers with parameters $a = -3$ and $beta = 2$. 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 */
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_METHOD_SEXPONENTIAL_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 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_METHOD_SGAUSSIAN_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_METHOD_SEXPONENTIAL_ICDF`.

The following example generates 100 3-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 */
vslNewStream( &stream, VSL_BRNG_SOBOL, 3 );
/* Generating */
vsRngUniform( VSL_METHOD_SUNIFORM_STD,
              stream, 100*3, (float*)r, 2.0f, 3.0f );
/* Deleting the streams */
vslDeleteStream( &stream );
```

Testing of Basic Random Number Generators

Three implementations are available for every basic generator:

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

You can use the basic generator 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 basic generator in general and its bit precision in particular. For detailed information on implementations of each basic generator see Basic Random Generator Properties and Testing Results.

All VSL basic generators are tested by a number of specially designed empirical tests. These tests are applied either for floating-point sequences or for integer-valued sequences.

The set of tests for basic generators can be divided into three categories:

- tests to analyze the randomness of bits/groups of bits
- tests to analyze the randomness of real random numbers normalized to the interval (0, 1)

- tests to analyze conformance to the template.

First Category

You can only use the first category tests to evaluate the basic generator integer implementation. The function `viRngUniformBits` corresponds to the integer implementation on the interface level. The testing in this category of tests is made with regard to characteristics of each basic generator and its bit precision in particular. You can subsequently use the results of the tests to decide if you can apply this particular basic generator to obtain random bits or groups of bits. A failed test does not mean that the generator is bad but rather that the interpretation of the integer output as the stream of random bits may result in an inadequate simulation outcome. Also, this category 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.

Second Category

The second category contains different tests for basic generator 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 basic generator output normalized to the (0, 1) interval. The functions `vsRngUniform` and `vdRngUniform`, for single and double precision respectively, correspond to real implementations on the interface level.

Third Category

The third category contains tests to check how a basic generator 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 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]. However, situations are possible, where the random choice of the method of generation is not a result of personal preference but rather the curse of a bug.

Interpreting Test Results

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

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, we may consider the sequence $u_1, u_2, \ldots, u_n$ 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 the sequence makes the statistical conclusion about the sequence randomness more substantiated with several options to arrive at such a conclusion.

One-Level (Threshold) Testing

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

We have conducted threshold testing for the VSL generators with 10 iterations (K=10), the interval (p_l, p_h) equal to (0.05, 0.95), and r = 5. The chance to reject a 'good' sequence in this case is 0.16349374% ≅ 0.2%.

Two-Level Testing

When we test K subsequences $u_1, u_2, \ldots, u_n; u_{n+1}, u_{n+2}, \ldots, u_{2n}; \ldots; u_{(K-1)n+1}, u_{(K-1)n+2}, \ldots, u_{Kn}$ of the original sequence, we compute p-values $p_1, p_2, \ldots, p_K$. Since the resulting p-values for the sequence $u_1, u_2, \ldots, u_{Kn}$ of truly random numbers are supposed to be uniformly distributed over the interval (0, 1), we may subject those p-values to any uniformity test, thus obtaining p-value q_1 of the second level. After going through this procedure L times we obtain L p-values of the second level $q_1, q_2, \ldots, q_L$ that we subject to threshold testing.

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

BRNG Tests Description

Most of empirical tests that are used for testing the VSL BRNGs are well documented (for example, see [Mars95], [Ziff98]). Nevertheless, we find it useful to describe them and the testing procedure in greater detail here since tests may vary as to their applicability and implementation for a particular basic generator. We also provide figures of merit that are used to decide on passing vs. failure in one- or two level testing. For ideas underlying such criteria see Interpreting Test Results section.

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 $a = 0$, $\beta = 40\pi$ parameters. Thus, the distribution of the $p = 1 – \exp(–(d_{min})^3/30)$ value should be close to the uniform distribution. The p-value is the result of the first level test.

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 basic generator*.

Birthday Spacing Test

Test Purpose

The test uses simulation to evaluate the randomness of groups of 24 sequential bits of the integer output of basic generator. The test analyzes all possible groups of the kind, that is, for example, from 0 to 23 bit, from 1 to 24 bit, etc.

First Level Test

The first level test selects at random $m = 2^{10}$ “birthdays” from a “year” of $n = 2^{24}$ 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 $\lambda = 16$ parameter. The first level test determines 200 values of K_j ($j = 1, 2, …, 200$). 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 basic generator.

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.

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

- Selects the b_s, b_{s+1}, …, b_{s+23} bits from the next WS-bit integer of the integer output of `viRngUniformBits`.
- 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.
- 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): $0 \le s \le \text{NB} - 24$.

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 $0 \le s \le \text{NB} - 24$. The test computes the $FAIL_s$ percentage for the failed second level tests. The final result is the minimal percentage of the failed tests $FAIL = \min(FAIL_0, FAIL_1, \ldots, FAIL_{\text{NB}-24})$ for $0 \le s \le \text{NB} - 24$. 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.

- 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.
- 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.
- 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.
- 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

Bitstream Test

Test Purpose

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

Note: *The bit precision of a basic generator 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, … from the integer output of the basic generator and then forms 20-bit overlapping words $w_0 = b_0 b_1 \ldots b_{19}$, $w_1 = b_1 b_2 \ldots b_{20}$, … from the sequence. From the total number of 20^{21} 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 σ = 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).

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.

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

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 the 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, the number of matrices with the rank of 30, the number of matrices with the rank of 29, and the number of matrices with the rank less than 29. For the truly random sequence, the probability of composing a 31 rank matrix is 0.289, a 30 rank matrix is 0.578, a 29 rank matrix is 0.128, and a less than 29 rank matrix is 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.

Note: *The acceptable values of* $0 \le s \le NB - 31$ *are specific for each basic generator. 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.

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 $p_j, j = 1, 2, \ldots, 10$ .The test applies the Kolmogorov-Smirnov goodness-of-fit test with Anderson-Darling statistics to the obtained set of p_j $j = 1, 2, \ldots, 10$. 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 $0 \le s \le NB - 31$. The test computes the *FAIL* percentage of the failed second level tests. The final result is the minimal percentage of the failed tests $FAIL = \min(FAIL_0, FAIL_1, \ldots, FAIL_{NB-31})$ for $0 \le s \le NB - 31$. 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.

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, the number of matrices with the rank of 31, the number of matrices with the rank of 30, and the number of matrices with the rank less than 30. For the truly random sequence the probability of composing a 30 rank matrix is 0.289, a 31 rank matrix is 0.578, a 30 rank matrix is 0.128, and a less than 30 rank matrix is 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.

Note: *The acceptable values of $0 \le s \le NB–32$ are specific for each basic generator. The test is not applicable for basic generators MCG31 and 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.

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 $0 \le s \le NB – 32$. The test computes the *FAIL* percentage of the failed second level tests. The final result is the minimal percentage of the failed tests $FAIL = \min(FAIL_0, FAIL_1, ..., FAIL_{NB–32})$ for $0 \le s \le NB – 32$. 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

The Rank of 32x32 Binary Matrices Test cannot be applied to the WH generator as each element of this generator is only 24-bit.

The Rank of 32x32 Binary Matrices Test cannot be applied to the MCG31generator as each element of this generator is only 31-bit.

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, the number of matrices with the rank of 5, and the number of matrices with the rank less than 5. For the truly random sequence the probability of composing a 6 rank matrix is 0.773, a 5 rank matrix is 0.217, and a less than 5 rank matrix is 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.

Note: *The acceptable values of $0 \le s \le NB - 8$ are specific for each basic generator. The test checks each of the 4 elements of the integer output for the WH basic generator.*

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.

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 $p_j, j = 1, 2, \ldots, 10$. The test applies the Kolmogorov-Smirnov goodness-of-fit test with Anderson-Darling statistics to the obtained set of $p_j, j = 1, 2, \ldots, 10$. 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 $0 \le s \le NB–8$. The test computes the *FAIL* percentage of the failed second level tests. The final result is the minimal percentage of the failed tests $FAIL = \min(FAIL_0, FAIL_1, \ldots, FAIL_{NB-8})$ for $0 \le s \le NB–8$. 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.

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 the basic generator. The test regards 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_2, q_3, q_4 respectively. 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 2,560,000 of the obtained words. According to these frequencies, the test makes the *chi*-square statistics V_1 and V_2 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 $V_2 - V_1$ statistic. The $V_2 - V_1$ statistic is asymptotically normal with a mean a = 2500 and standard deviation σ = 70.71. The result of the first level test is the p-value.

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.

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 generator uses all the four elements to form a bit sequence.

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 the basic generator. 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 $0 \le s \le$ 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 V_1 and V_2 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 $V_2 - V_1$ statistic. The $V_2 - V_1$ statistic is asymptotically normal with a mean $a = 2500$ and standard distribution $\sigma = 70.71$. The result of the first level test is the *p*-value.

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.

Second Level Test

The second level test performs the first level test ten times for the fixed $0 \le s \le$ NB–8. The test applies the Kolmogorov-Smirnov goodness-of-fit test with Anderson-Darling statistics to the obtained p-values of $p_j, j = 1, 2, \ldots, 10$. 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 of $0 \le s \le$ NB–8. The test computes the *FAIL* percentage of the failed second level tests. The final result is the minimal for $0 \le s \le$ NB–8 percentage of the failed tests $FAIL = \min(FAIL_0, FAIL_1, \ldots, FAIL_{NB-8})$. 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 generator.

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 $\sigma = a*251/495$.

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, $j = 1, 2, \ldots, 10$. 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. Similarly, the test applies the Kolmogorov-Smirnov goodness-of-fit test with Anderson-Darling statistics to the obtained p-values of q_j, $j = 1, 2, \ldots, 10$. 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

Parking Lot Test

Test Purpose

The test evaluates the randomness of two-dimensional random points uniformly distributed in the square with a side of length 100. The stable response is 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 K value of points successfully parked after 12,000 attempts haves close to normal distribution with mean a = 3,523 and standard deviation σ = 21.9. Consequently, $(K–a)/\sigma$ should have a close to standard normal distribution with the $\Phi(x)$ cumulative distribution function. The result of the test is the p-value $p = \Phi((K - a)/\sigma)$.

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 *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

2D 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 cornerwise. 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 ½ 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

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 $x_1, x_2, \ldots x_k, x_{k+1}, \ldots$ 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 $x_{k+1}, \ldots$ 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. In this connection the quality of the obtained sequence is highly probable to be much worse than the quality of the template sequence. That is why all the basic generators of the VSL undergo thorough tests for template sequences conformity.

The test also checks the basic generators with the random output numbers $u_1, u_2, \ldots, u_k, u_{k+1}, \ldots$, 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. Other from the integer output where every member should coincide bitwisely with the template member, it is not necessary for the real output members. The lower bits of mantissa of the real output do not influence randomness, these are the upper bits that determine the quality of the output sequence. For example, the coincidence of the upper binary digits of mantissa is sufficient enough for most applications. (See the chapter Spectral Test in [Knuth81]).

This test is also used to validate VSL 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 mantissa coincide.

Tested Generators

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

Basic Random Generator Properties and Testing Results

This section contains the empirical testing results for the VSL basic generators described in the BRNG Test Description section and other information on the properties of basic generators and the rules of the output vector interpretation.

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^{32}. 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 $u_0, u_1, \ldots$

Integer Implementation

The output vector of 32-bit integers $x_0, x_1, \ldots$

Stream Initialization by the Function `vslNewStream`

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

- Assume x_0 = `seed mod 0x7FFFFFFF`
- If $x_0 = 0$, assume $x_0 = 1$.

Stream Initialization by the Function `vslNewStreamEx`

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

- If $n = 0$, assume x_0 = `1`
- Otherwise assume x_0 = `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)	N/A	OK (10% errors)
Birthday Spacing Test	N/A	N/A	N/A	OK (0% errors)
Bitstream Test	N/A	N/A	N/A	OK (10% errors)
Rank of 31x31 Binary Matrices Test	N/A	N/A	N/A	OK (10% errors)
Rank of 32x32 Binary Matrices Test	N/A	N/A	N/A	N/A
Rank of 6x8 Binary Matrices Test	N/A	N/A	N/A	OK (0% errors)
Counts-the-1's Test (stream of bits)	N/A	N/A	N/A	OK (20% errors)
Counts-the-1's Test (stream of specific bytes)	N/A	N/A	N/A	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)	N/A	OK (10% errors)
2D Self-Avoiding Random Walk Test	OK (20% errors)	OK (20% errors)	N/A	OK (20% errors)

Note:

- N/A means that the test is not applicable to this function.

- The tabulated data is obtained using the one-level (threshold) testing technique. The *OK* result indicates *FAIL* < 50%, that is, when *FAIL*s occur in less than 5 runs out of 10. The run is failed when *p*-value falls outside the interval [0.05, 0.95].
- The stream tested is generated by calling the function `vslNewStream` with `seed=7,777,777`.

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 first were intended for cryptographic and communication applications. The physicists widely use R250 generator, as it is simple and fast in implementation. However, it fails 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 $u_0, u_1, \ldots$

Integer Implementation

The output vector of 32-bit integers $x_0, x_1, \ldots$

Stream Initialization by the 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 $x_{-250}, x_{-249}, \ldots, x_{-1}$, initialized in the following way:

- If `seed` = 0, assume `seed` = 1. Assume x_{-250} = `seed`.
- Initialize $x_{-249}, \ldots, x_0$ according to recurrent correlation $x_{n+1} = 69069 x_n \pmod{2^{32}}$.
- Interpret the values x_{7k-247}, $k = 0,1,\ldots,31$ 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 the Function `vslNewStreamEx`

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

- 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)	N/A	OK (0% errors)
Birthday Spacing Test	N/A	N/A	N/A	OK (0% errors)
Bitstream Test	N/A	N/A	N/A	OK (25% errors)
Rank of 31x31 Binary Matrices Test	N/A	N/A	N/A	OK (10% errors)
Rank of 32x32 Binary Matrices Test	N/A	N/A	N/A	OK (0% errors)
Rank of 6x8 Binary Matrices Test	N/A	N/A	N/A	OK (0% errors)
Counts-the-1's Test (stream of bits)	N/A	N/A	N/A	OK (30% errors)
Counts-the-1's Test (stream of specific bytes)	N/A	N/A	N/A	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)	N/A	OK (0% errors)
2D Self-Avoiding Random Walk Test	FAIL (70% errors)	FAIL (80% errors)	N/A	FAIL (80% errors)

Note:

- N/A means that the test is not applicable to this function.
- The tabulated data is obtained using the one-level (threshold) testing technique. The *OK* result indicates *FAIL* < 50%, that is, when *FAIL*s occur in less than 5 runs out of 10. The run is failed when *p*-value falls outside the interval [0.05, 0.95].
- The stream tested is generated by calling the function `vslNewStream` with `seed=7,777,777`.

MRG32k3a

This is a 32-bit combined multiple recursive generator with 2 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, long period, etc. Optimization for various Intel® architectures makes it competitive with the other VSL basic generators 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 the 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:

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

Stream Initialization of the Function `vslNewStreamEx`

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

- If *n* = 0, assume $x_{-3} = x_{-2} = x_{-1} = y_{-3} = y_{-2} = y_{-1} = 1$.
- If *n* = 1, assume x_{-3} = `params[0] mod` m_1, $x_{-2} = x_{-1} = y_{-3} = y_{-2} = y_{-1} = 1$.
- If *n* = 2, assume x_{-3} = `params[0] mod` m_1, x_{-2} = `params[1] mod` m_1, $x_{-1} = y_{-3} = y_{-2} = y_{-1} = 1$.
- 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 $x_{-3} = 1$.
- 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 $x_{-3} = 1$.

- If $n = 5$, assume x_{-3} = `params[0] mod m1`, x_{-2} = `params[1] mod m1`, x_{-1} = `params[2] mod m1`, y_{-3} = `params[3] mod m2`, y_{-2} = `params[4] mod m2`, $y_{-1} = 1$. If the values prove to be $x_{-3} = x_{-2} = x_{-1} = 0$, assume $x_{-3} = 1$.
- If $n \geq 6$, assume x_{-3} = `params[0] mod m1`, x_{-2} = `params[1] mod m1`, x_{-1} = `params[2] mod m1`, y_{-3} = `params[3] mod m2`, y_{-2} = `params[4] mod m2`, y_{-1} = `params[5] mod m2`. 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

Test Name	vsRngUniform	vdRngUniform	viRngUniform	viRngUniformBits
3D Spheres Test	OK (10% errors)	OK (10% errors)	N/A	OK (10% errors)
Birthday Spacing Test	N/A	N/A	N/A	OK (0% errors)
Bitstream Test	N/A	N/A	N/A	OK (20% errors)
Rank of 31x31 Binary Matrices Test	N/A	N/A	N/A	OK (20% errors)
Rank of 32x32 Binary Matrices Test	N/A	N/A	N/A	OK (10% errors)
Rank of 6x8 Binary Matrices Test	N/A	N/A	N/A	OK (0% errors)
Counts-the-1's Test (stream of bits)	N/A	N/A	N/A	OK (20% errors)
Counts-the-1's Test (stream of specific bytes)	N/A	N/A	N/A	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)	N/A	OK (10% errors)
2D Self-Avoiding Random Walk Test	OK (20% errors)	OK (20% errors)	N/A	OK (20% errors)

Note:

- N/A means that the test is not applicable to this function.

- The tabulated data is obtained using the one-level (threshold) testing technique. The *OK* result indicates *FAIL* < 50%, that is, when *FAIL*s occur in less than 5 runs out of 10. The run is failed when *p*-value falls outside the interval [0.05, 0.95].
- The stream tested is generated by calling the function `vslNewStream` with `seed=7,777,777`.

MCG59

This is a 59-bit multiplicative congruential generator:

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

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, for example, [Cram46], [Ent98]): 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 trouble**.** Besides, block-splitting an entire period sequence into 2^d 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 $u_0, u_1, \ldots$

Integer Implementation

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

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 2*n* 32-bit numbers.

Stream Initialization by the Function `vslNewStream`

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

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

Stream Initialization of the Function `vslNewStreamEx`

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

- If $n = 0$, assume $x_0 = 1$.
- If $n = 1$, assume `seed = params[0]`, follow the instructions described in the above section on stream initialization by the function `vslNewStream`.
- 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)	N/A	OK (10% errors)
Birthday Spacing Test	N/A	N/A	N/A	OK (0% errors)[1]
Bitstream Test	N/A	N/A	N/A	OK (45% errors)
Rank of 31x31 Binary Matrices Test	N/A	N/A	N/A	OK (0% errors)[2]
Rank of 32x32 Binary Matrices Test	N/A	N/A	N/A	OK (0% errors)[3]
Rank of 6x8 Binary Matrices Test	N/A	N/A	N/A	OK (0% errors)[4]
Counts-the-1's Test (stream of bits)	N/A	N/A	N/A	FAIL (100% errors)
Counts-the-1's Test (stream of specific bytes)	N/A	N/A	N/A	OK (0% errors)[5]
Craps Test	OK (10% errors)	OK (10% errors)	OK (10% errors)	OK (10% errors)
Parking Lot Test	OK (20% errors)	OK (20% errors)	N/A	OK (20% errors)
2D Self-Avoiding Random Walk Test	OK (20% errors)	OK (10% errors)	N/A	OK (10% errors)

Note:

- N/A means that the test is not applicable to this function.
- The tabulated data is obtained using the one-level (threshold) testing technique. The *OK* result indicates *FAIL* < 50%, that is, when *FAIL*s occur in less than 5 runs out of 10. The run is failed when *p*-value falls outside the interval [0.05, 0.95].

[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.

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

WH

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

$$x_n = a_{1,j}x_{n-1} (\mathrm{mod}\, m_{1,j})$$
$$y_n = a_{2,j}y_{n-1} (\mathrm{mod}\, m_{2,j})$$
$$z_n = a_{3,j}z_{n-1} (\mathrm{mod}\, m_{3,j})$$
$$w_n = a_{4,j}w_{n-1} (\mathrm{mod}\, m_{4,j})$$
$$u_n = \left(x_n/m_{1,j} + y_n/m_{2,j} + z_n/m_{3,j} + w_n/m_{4,j}\right) \mathrm{mod} 1$$

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 constant 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 $u_0, u_1, \ldots$

Integer Implementation

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

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 for storage of 4n 32-bit numbers.

Stream Initialization by the Function `vslNewStream`

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

- Assume x_0 = `seed mod` m_1. If $x_0 = 0$, assume $x_0 = 1$.
- Assume $y_0 = 1$, $z_0 = 1$, $w_0 = 1$.

WH generator is a set of 273 basic generators. 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 the Function `vslNewStreamEx`

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

- If $n = 0$, assume $x_0 = 1, y_0 = 1, z_0 = 1, w_0 = 1$.
- If $n = 1$, assume $x_0 =$ `params[0] mod` m_1, $y_0 = 1, z_0 = 1, w_0 = 1$. If $x_0 = 0$, assume $x_0 = 1$.
- If $n = 2$, assume $x_0 =$ `params[0] mod` m_1, $y_0 =$ `params[1] mod` m_2, $z_0 = 1, w_0 = 1$. If $x_0 = 0$, assume $x_0 = 1$. If $y_0 = 0$, assume $y_0 = 1$.
- If $n = 3$, assume $x_0 =$ `params[0] mod` m_1, $y_0 =$ `params[1] mod` m_2, $z_0 =$ `params[2] mod` m_3, $w_0 = 1$. If $x_0 = 0$, assume $x_0 = 1$. If $y_0 = 0$, assume $y_0 = 1$. If $z_0 = 0$, assume $z_0 = 1$.
- If $n \geq 4$, assume $x_0 =$ `params[0] mod` m_1, $y_0 =$ `params[1] mod` m_2, $z_0 =$ `params[2] mod` m_3, $w_0 =$ `params[3] mod` m_4. If $x_0 = 0$, 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)	N/A	OK (0% errors)
Birthday Spacing Test	N/A	N/A	N/A	FAIL (60% errors)
Bitstream Test	N/A	N/A	N/A	OK (10% errors)
Rank of 31x31 Binary Matrices Test	N/A	N/A	N/A	N/A
Rank of 32x32 Binary Matrices Test	N/A	N/A	N/A	N/A
Rank of 6x8 Binary Matrices Test	N/A	N/A	N/A	OK (0% errors)[1]
Counts-the-1's Test (stream of bits)	N/A	N/A	N/A	OK (10% errors)
Counts-the-1's Test (stream of specific bytes)	N/A	N/A	N/A	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)	N/A	OK (10% errors)
2D Self-Avoiding Random Walk Test	OK (10% errors)	OK (0% errors)	N/A	OK (20% errors)

[1] The component *y* of the generator fails the test for bit group 1-8.

Note:

- N/A means that the test is not applicable to this function.
- The tabulated data is obtained using the one-level (threshold) testing technique. The *OK* result indicates *FAIL* < 50%, that is, when *FAIL*s occur in less than 5 runs out of 10. The run is failed when *p*-value falls outside the interval [0.05, 0.95].
- The stream tested is generated by calling the function `vslNewStream` with `seed=7,777,777`.

MT19937

This is a Mersenne Twister pseudorandom number generator:

$$x_n = x_{n-(624-397)} \oplus ((x_{n-624} \;\&\; 0\text{x}80000000\;) \,|\, (x_{n-624+1} \;\&\; 0\text{x}7\text{FFFFFFF}\;))A\,,$$

$$y_n = x_n\,,$$

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

$$y_n = y_n \oplus ((y_n << 7) \,\&\, 0\text{x}9\text{D}2\text{C}5680)\,,$$

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

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

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

Matrix A (32x32) has the following format:

$$\mathtt{A} = \begin{vmatrix} 0 & 1 & 0 & & \\ 0 & 0 & \ldots & 0 & \\ & & & \ldots & \\ & & 0 & 0 & 1 \\ a_{31} & a_{30} & \ldots & \ldots & a_0 \end{vmatrix},$$

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

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 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 $u_0, u_1, \ldots$

Integer Implementation

The output vector of 32-bit integers $y_0, y_1, \ldots$

Stream Initialization by the 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 $x_0, \ldots, x_{623}$, is initialized by the procedure described in [MT2002] and based on the `seed` value.

Stream Initialization of the Function `vslNewStreamEx`

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

- If $n \geq 1$, perform initialization as described in [MT2002] using array `params[]` on input.
- 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

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

Empirical Testing Results Summary

Test Name	vsRngUniform	vdRngUniform	viRngUniform	viRngUniformBits
3D Spheres Test	OK (0% errors)	OK (0% errors)	N/A	OK (0% errors)
Birthday Spacing Test	N/A	N/A	N/A	OK (10% errors)
Bitstream Test	N/A	N/A	N/A	OK (10% errors)
Rank of 31x31 Binary Matrices Test	N/A	N/A	N/A	OK (10% errors)
Rank of 32x32 Binary Matrices Test	N/A	N/A	N/A	OK (0% errors)
Rank of 6x8 Binary Matrices Test	N/A	N/A	N/A	OK (0% errors)
Counts-the-1's Test (stream of bits)	N/A	N/A	N/A	OK (20% errors)
Counts-the-1's Test (stream of specific bytes)	N/A	N/A	N/A	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)	N/A	OK (0% errors)
2D Self-Avoiding Random Walk Test	OK (0% errors)	OK (10% errors)	N/A	OK (10% errors)

Note:

- N/A means that the test is not applicable to this function.

- The tabulated data is obtained using the one-level (threshold) testing technique. The *OK* result indicates *FAIL* < 50%, that is, when *FAIL*s occur in less than 5 runs out of 10. The run is failed when *p*-value falls outside the interval [0.05, 0.95].
- The stream tested is generated by calling the function `vslNewStream` with `seed=7,777,777`.

MT2203

This is a set of 1024 Mersenne Twister pseudorandom number generators (j = 1, …, 1024):

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

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

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

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

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

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

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

Matrix A_j (32x32) has the following format:

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

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

The set of 1024 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}-1 is big enough to meet the requirements of modern Monte Carlo problems. MT2203 produces up to 1024 independent random number sequences. The parameters have been carefully chosen according to the method described in [Matsum2000].

Real Implementation (single and double precision)

The output vector is the sequence of the floating-point values $u_0, u_1, \ldots$

Integer Implementation

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

Stream Initialization by the 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 1024 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 10^{th} (of 1024) MT2203 generator:

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

Stream Initialization of the Function `vslNewStreamEx`

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

- If $n \geq 1$, perform initialization as described in [MT2002] using array `params[]` on input.
- 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)	N/A	OK (20% errors)
Birthday Spacing Test	N/A	N/A	N/A	OK (0% errors)
Bitstream Test	N/A	N/A	N/A	OK (15% errors)
Rank of 31x31 Binary Matrices Test	N/A	N/A	N/A	OK (10% errors)
Rank of 32x32 Binary Matrices Test	N/A	N/A	N/A	OK (0% errors)
Rank of 6x8 Binary Matrices Test	N/A	N/A	N/A	OK (0% errors)
Counts-the-1's Test (stream of bits)	N/A	N/A	N/A	OK (0% errors)
Counts-the-1's Test (stream of specific bytes)	N/A	N/A	N/A	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)	N/A	OK (0% errors)
2D Self-Avoiding Random Walk Test	OK (10% errors)	OK (0% errors)	N/A	OK (0% errors)

Note:

- N/A means that the test is not applicable to this function.
- The tabulated data is obtained using the one-level (threshold) testing technique. The *OK* result indicates *FAIL* < 50%, that is, when *FAIL*s occur in less than 5 runs out of 10. The run is failed when *p*-value falls outside the interval [0.05, 0.95].
- The stream tested is generated by calling the function `vslNewStream` with `seed=7,777,777`.

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;* $\mathbf{x}_n$ *is s-dimensional vector of 32-bit values. The s-dimensional vectors (calculated during random stream initialization)* $\mathbf{v}_i, i = \overline{1,32}$ *are called direction numbers. The vector* $\mathbf{u}_n$ *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. VSL implementation allows generating Sobol's low-discrepancy sequences of length up to 2^{32}. The 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 $\mathbf{u}_1$, $u_{s+11}, u_{s+2}, \ldots, u_{2s}$ correspond to the $\mathbf{u}_2$, 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 $\mathbf{x}_1$, $x_{s+11}, x_{s+2}, \ldots, x_{2s}$ correspond to the $\mathbf{x}_2$, and so on.

Stream Initialization by the Function `vslNewStream`

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

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

Stream Initialization by the Function `vslNewStreamEx`

SOBOL generates the stream and initializes it specifying the array *n* of 32-bit integers `params[]` to set the dimension `dimen` of a quasi-random vector:

- If *n* = 0, assume `dimen = 1`
- Otherwise assume `dimen = params[0]`
    - If `dimen` < 1 or `dimen` > 40, assume `dimen` = 1.

Subsequences Selection Methods

vslSkipAheadStream	supported
vslLeapfrogStream	supported

Note:

- 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*×s.
- 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

$1 \le s \le 40$.

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 rightmost zero bit in n-1;* $\mathbf{x}_n$ *is s-dimensional vector of 32-bit values. The s-dimensional vectors (calculated during random stream initialization)* $\mathbf{v}_i, i = \overline{1,32}$ *are called direction numbers. The vector* $\mathbf{u}_n$ *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. VSL implementation allows generating Niederreiter low-discrepancy sequences of length up to 2^{32}. The 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 $u_1, u_2, \ldots$, where elements $u_1, u_2, \ldots, u_s$ correspond to the $\mathbf{u}_1$, $u_{s+11}, u_{s+2}, \ldots, u_{2s}$ correspond to the $\mathbf{u}_2$, 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 $\mathbf{x}_1$, $x_{s+11}, x_{s+2}, \ldots, x_{2s}$ correspond to the $\mathbf{x}_2$, and so on.

Stream Initialization by the Function `vslNewStream`

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

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

Stream Initialization by the Function `vslNewStreamEx`

NIEDERREITER generates the stream and initializes it specifying the array n of 32-bit integers `params[]` to set the dimension `dimen` of a quasi-random vector:

- If n = 0, assume `dimen` = 1
- Otherwise assume `dimen = params[0]`
    - If `dimen` < 1 or `dimen` > 318, assume `dimen` = 1.

Subsequences Selection Methods

vslSkipAheadStream	supported
vslLeapfrogStream	supported

Note:

- 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.
- 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

$\rho = 2^{32} \approx 4.2\times10^{9}$.

Dimensions

$1 \le s \le 318$.

Testing of Distribution Random Number Generators

VSL generators are tested with a testing suite comprising 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.

VSL 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.

VSL offers two different implementations for each of continuous distributions:

- single-precision real arithmetic
- double-precision real arithmetic.

Single-precision generator implementation is, as a rule, faster than that for double-precision implementation. Moreover, single-precision implementation is quite sufficient for most applications. VSL offers only one implementation 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.

Interpreting Test Results

Test results for general distribution generators are interpreted almost in the same way as for basic generators. For reliable results, either one-level (threshold) or two-level testing is performed.

Description of Distribution Generator Tests

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 β all the output members x_i should lie within the range $a \le x_i < \infty$. A value $x_i < a$ is impossible, that is, the fact that the variate X of exponential distribution with parameters a and β 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 constitutes the case of an error.

Such a test is necessary because statistical tests (for example, distribution moments test or *chi*-square test) are unable to 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.

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 response.

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 asymptomatically 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 10 times, each run producing a pair of p-values p_j^M and p_j^D, $j = 1, 2, \ldots, 10$. The Kolmogorov-Smirnov goodness-of-fit test with Anderson-Darling's statistics is applied to the obtained p-values p_j^M, $j = 1, 2, \ldots, 10$. 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, $j = 1, 2, \ldots, 10$, 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

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

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. Since 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, p-value, which is 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 10 times, each run producing a p-value p_j, $j = 1, 2, \ldots, 10$. The Kolmogorov-Smirnov goodness-of-fit test with Anderson-Darling's statistics is applied to the obtained p-values p_j, $j = 1, 2, \ldots, 10$. If the resulting p-value $p^M < 0.05$ or $p^M > 0.95$, the test is considered failed.

Interpreting Final Results

The final result of the test is the percentage $FAIL$ of failed second level tests. The second level test is run 10 times. The value of $FAIL < 50\%$ is considered acceptable.

Performance

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

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

VSL random number generators are optimized for Intel® Pentium® 4 processor and Intel® Itanium® 2 processor. See specific tables at http://www.intel.com/software/products/mkl for generator performance for each individual processor. For earlier Intel processors VSL generators are fully functional, yet not specifically optimized.

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

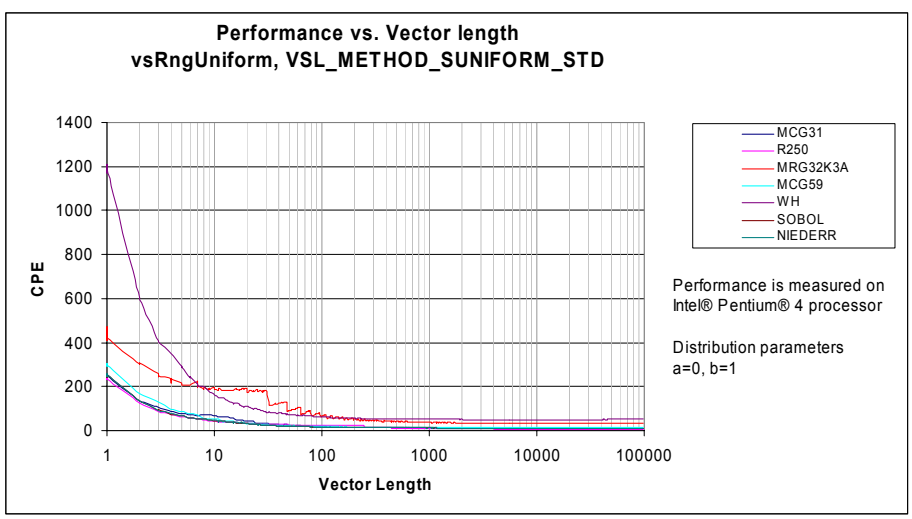
For example, if the generator performance is equal to 10 CPE and the processor rate is 1 GHz, then the generator will produce 10^8 random numbers per second.

The VSL 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 VSL 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.

The tables of RNG performance provide speed data obtained using the most indicative vector length of 1000 elements. For other vector lengths the performance of any generator behaves approximately in the same way as shown in the following graph.



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.

Continuous Distribution Functions

Uniform (VSL_METHOD_SUNIFORM_STD/ VSL_METHOD_DUNIFORM_STD)

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 `Uniform` function calls real implementation (of single precision for `vsRngUniform` and of double precision for `vdRngUniform`) of this basic generator.

See http://www.intel.com/software/products/mkl for test results summary.

Gaussian (VSL_METHOD_SGAUSSIAN_BOXMULLER / VSL_METHOD_DGAUSSIAN_BOXMULLER)

Random number generator of normal (Gaussian) distribution with the parameters a and σ. You may obtain any successive random number x of the standard normal distribution according to the formula (for details, see [Box58])

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

where u_1, u_2 are a pair of successive random numbers uniformly distributed over the interval (0, 1).

The normal distribution with the parameters a and σ is transformed to the random number y by scaling and the shift $y = \sigma x + a$.

See http://www.intel.com/software/products/mkl for test results summary.

Gaussian (VSL_METHOD_SGAUSSIAN_BOXMULLER2 / VSL_METHOD_DGAUSSIAN_BOXMULLER2)

Random number generator of normal (Gaussian) distribution with the parameters a and σ. You may produce a successive pair of the random numbers x_1, x_2 of the standard normal distribution according to the formula (for details, see [Box58])

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

where u_1, u_2 are a pair of successive random numbers uniformly distributed over the interval (0, 1).

The normal distribution with the parameters a and σ is transformed to the random number y by scaling and the shift $y = \sigma x + a$.

In VSL you can safely call this method even when the random numbers are generated in blocks with the size aliquant to 2. Consider the following example.

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

Option 1. Single call of the method VSL_METHOD_DGAUSSIAN_BOXMULLER2 with the vector length equal to 2:

```
…
double x[2];
…
vdRngGaussian(VSL_METHOD_DGAUSSIAN_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_METHOD_DGAUSSIAN_BOXMULLER2, stream, 1, &x[0], 0.0, 1.0);
vdRngGaussian(VSL_METHOD_DGAUSSIAN_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

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

as it might be supposed.

See http://www.intel.com/software/products/mkl for test results summary.

GaussianMV (VSL_METHOD_SGAUSSIANMV_BOXMULLER / VSL_METHOD_DGAUSSIANMV_BOXMULLER)

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

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

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

Random numbers from standard normal distribution are generated by the method

`VSL_METHOD_SGAUSSIAN_BOXMULLER/VSL_METHOD_DGAUSSIAN_BOXMULLER`.

See http://www.intel.com/software/products/mkl for test results summary and performance graphs.

GaussianMV (VSL_METHOD_SGAUSSIANMV_BOXMULLER2 / VSL_METHOD_DGAUSSIANMV_BOXMULLER2)

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

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

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

Random numbers from standard normal distribution are generated by the method

`VSL_METHOD_SGAUSSIAN_BOXMULLER2/VSL_METHOD_DGAUSSIAN_BOXMULLER2`.

See http://www.intel.com/software/products/mkl for test results summary and performance graphs.

Exponential (VSL_METHOD_SEXPONENTIAL_ICDF/ VSL_METHOD_DEXPONENTIAL_ICDF)

Random number generator of the exponential distribution with the parameters a and β. You may 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 http://www.intel.com/software/products/mkl for test results summary.

Laplace (VSL_METHOD_SLAPLACE_ICDF/ VSL_METHOD_DLAPLACE_ICDF)

Random number generator of the Laplace distribution with the parameters a and β. You may 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 http://www.intel.com/software/products/mkl for test results summary.

Weibull (VSL_METHOD_SWEIBULL_ICDF/ VSL_METHOD_DWEIBULL_ICDF)

Random number generator of the Weibull distribution with the parameters α, a and β. You may generate any successive random number x of the Weibull distribution by the inverse transformation method from the formula

$$x = \beta(-\ln(u))^{1/\alpha} + a,$$

where u is a successive random number of a uniform distribution over the interval (0, 1).

See http://www.intel.com/software/products/mkl for test results summary.

Cauchy (VSL_METHOD_SCAUCHY_ICDF/ VSL_METHOD_DCAUCHY_ICDF)

Random number generator of the Cauchy distribution with the parameters a and β. You may 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 $(-\pi/2, \pi/2)$.

See http://www.intel.com/software/products/mkl for test results summary.

Rayleigh (VSL_METHOD_SRAYLEIGH_ICDF/ VSL_METHOD_DRAYLEIGH_ICDF)

Random number generator of the Rayleigh distribution with the parameters a and β. You may 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 http://www.intel.com/software/products/mkl for test results summary.

Lognormal (VSL_METHOD_SLOGNORMAL_ICDF/ VSL_METHOD_DLOGNORMAL_ICDF)

Random number generator of the lognormal distribution with the parameters a, σ, b and β. You may 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 the parameters a and σ.

The random numbers of the normal distribution are generated using the method `VSL_METHOD_SGAUSSIAN_BOXMULLER2 / VSL_METHOD_DGAUSSIAN_BOXMULLER2`.

See http://www.intel.com/software/products/mkl for test results summary.

Gumbel (VSL_METHOD_SGUMBEL_ICDF/ VSL_METHOD_DGUMBEL_ICDF)

Random number generator of the Gumbel distribution with the parameters a and β. You may 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 $\beta = 1$.

The random numbers of the exponential distribution are generated using the method `VSL_METHOD_SEXPONENTIAL_ICDF/ VSL_METHOD_DEXPONENTIAL_ICDF`.

See http://www.intel.com/software/products/mkl for test results summary.

Gamma (VSL_METHOD_SGAMMA_GNORM/ VSL_METHOD_DGAMMA_GNORM)

Random number generator of the gamma distribution with the parameters shape α, offset a, and scalefactor β. You may generate any successive random number γ_α of the standard gamma distribution (a=0, β=1) as follows:

- 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.
- 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].

Note that when $\alpha = 1$ gamma distribution is reduced to exponential distribution with parameters a, β. The random numbers of the exponential distribution are generated using the method VSL_METHOD_SEXPONENTIAL_ICDF/VSL_METHOD_DEXPONENTIAL_ICDF. The gamma distributed random number y with the parameters α, a, and β is transformed from γ_α using scale and shift $y = a + \beta\gamma_\alpha$.

See http://www.intel.com/software/products/mkl for test results summary.

Beta (VSL_METHOD_SBETA_CJA/ VSL_METHOD_DBETA_CJA)

Random number generator of the beta distribution with two shape parameters p and q, offset a, and scalefactor β. You may generate any successive random number $\theta(p,q)$ of the standard gamma distribution (a=0, β=1) as follows:

- if $\min(p,q) > 1$, Cheng algorithm is used (for details, see [Cheng78])
- if $\max(p,q) < 1$, composition of two algorithms is applied: if $q + K * P^2 + C \leq 0$, where K = 0.852..., C = - 0.956..., Jöhnk algorithm is used (for details, see [Jöhnk64]); otherwise Atkinson switching algorithm is used (for details, see [Atkin79])
- if $\min(p,q) < 1$ and $\max(p,q) > 1$, the random numbers are generated using the switching algorithm of Atkinson (for details, see [Atkin79])
- if $p = 1$ or $q = 1$, the inverse transformation method is used
- 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_METHOD_SUNIFORM_STD/VSL_METHOD_DUNIFORM_STD method.

The algorithms of Cheng and Atkinson use acceptance/rejection technique. The beta distributed random number y with the parameters p, q, a, and β is transformed from $\theta(p,q)$ as follows: $y = a + \beta\theta(p,q)$.

See http://www.intel.com/software/products/mkl for test results summary.

Discrete Distribution Functions

Uniform (VSL_METHOD_IUNIFORM_STD)

Uniform discrete distribution over the integer interval $[a,b)$. You may 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 http://www.intel.com/software/products/mkl for test results summary.

UniformBits (VSL_METHOD_IUNIFORMBITS_STD)

Random number generator of uniform distribution that produces an integer (non-normalized to the interval (0, 1)) sequence. You may 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 table provides rules for interpreting 32-bit integer output $r[i]$ for each VSL basic generator.

BRNG	Integer Recurrence	Interpretation of 32-bit integer output array *r[i]* after calling `viRngUniformBits`
MCG31m1	$x_i = ax_{i-1} (\mathrm{mod}\, m)$ $u_i = x_i / m$ $a = 1132489760, m = 2^{31} - 1$	$r[i] = x_i$
R250	$x_i = x_{i-103} \oplus x_{i-250}$ $u_i = x_i / 2^{32}$	$r[i] = x_i$
MRG32k3a	$x_i = a_{11}x_{i-1} + a_{12}x_{i-2} + a_{13}x_{i-3} (\mathrm{mod}\, m_1)$ $y_i = a_{21}y_{i-1} + a_{22}y_{i-2} + a_{23}y_{i-3} (\mathrm{mod}\, m_2)$ $z_i = x_i - y_i (\mathrm{mod}\, m_1)$ $u_i = z_i / m_1$ $a_{11} = 0, a_{12} = 1403580, a_{13} = -810728, m_1 = 2^{32} - 209$ $a_{21} = 527612, a_{22} = 0, a_{23} = -1370589, m_2 = 2^{32} - 22853$	$r[i] = z_i$
MCG59	$x_n = ax_{n-1} (\mathrm{mod}\, m)$ $u_n = x_n / m$ $a = 13^{13}, m = 2^{59}$	$r[2i] = Lo(x_i),$ $r[2i+1] = Hi(x_i)$
WH	$x_n = a_{1,j}x_{n-1} (\mathrm{mod}\, m_{1,j})$ $y_n = a_{2,j}y_{n-1} (\mathrm{mod}\, m_{2,j})$ $z_n = a_{3,j}z_{n-1} (\mathrm{mod}\, m_{3,j})$ $w_n = a_{4,j}w_{n-1} (\mathrm{mod}\, m_{4,j})$ $u_n = (x_n/m_{1,j} + y_n/m_{2,j} + z_n/m_{3,j} + w_n/m_{4,j}) \mathrm{mod} 1$	$r[4i] = x_i$ $r[4i+1] = y_i$ $r[4i+2] = z_i$ $r[4i+3] = w_i$

MT19937	$x_n = x_{n-(624-397)} \oplus ((x_{n-624} \,\&\, 0\text{x}80000000) \mid (x_{n-624+1} \,\&\, 0\text{x}7\text{FFFFFFF}))A$, $y_n = x_n$, $y_n = y_n \oplus (y_n >> 11)$, $y_n = y_n \oplus ((y_n << 7) \,\&\, 0\text{x}9\text{D}2\text{C}5680)$, $y_n = y_n \oplus ((y_n << 15) \,\&\, 0\text{xEFC60000})$, $y_n = y_n \oplus (y_n >> 18)$, $u_n = y_n / 2^{32}$, where $A = \begin{vmatrix} 0 & 1 & 0 & & \\ 0 & 0 & \cdots & 0 & \\ & & & \cdots & \\ & & 0 & 0 & 1 \\ a_{31} & a_{30} & \cdots & \cdots & a_0 \end{vmatrix}$, with $a = a_{31} \ldots a_0 = 0\text{x}9908\text{B}0\text{DF}$.	$r[i] = y_i$
MT2203	$x_{i,j} = x_{i-(69-34),j} \oplus ((x_{i-69,j} \,\&\, 0xFFFFFFFE\ 0) \mid (x_{i-69+1,j} \,\&\, 0x1F))A_j$ $y_{i,j} = x_{i,j}$ $y_{i,j} = y_{i,j} \oplus (y_{i,j} >> 12)$ $y_{i,j} = y_{i,j} \oplus ((y_{i,j} << 7) \,\&\, b_j)$ $y_{i,j} = y_{i,j} \oplus ((y_{i,j} << 15) \,\&\, c_j)$ $y_{i,j} = y_{i,j} \oplus (y_{i,j} >> 18)$ $u_i = y_{i,j} / 2^{32}$, where	$r[i] = y_{i,j}$

	$A_j = \begin{vmatrix} 0 & 1 & 0 & & \\ 0 & 0 & \dots & 0 & \\ & & & \dots & \\ & & 0 & 0 & 1 \\ a_{31,j} & a_{30,j} & \dots & \dots & a_{0,j} \end{vmatrix}$, with $a_j = a_{31,j}\dots a_{0,j}$, $j = 1,\dots,1024$.	
SOBOL	$\mathbf{x}_n = \mathbf{x}_{n-1} \oplus \mathbf{v}_c$ $\mathbf{u}_n = \mathbf{x}_n / 2^{32}$, where $\mathbf{x}_n = (x_{s(n-1)+1}, x_{s(n-1)+2}, \dots, x_{sn})$, $\mathbf{u}_n = (u_{s(n-1)+1}, u_{s(n-1)+2}, \dots, u_{sn})$ and *s* is the dimension of quasi-random vector.	$r[i-1] = x_i$
NIEDERR	$\mathbf{x}_n = \mathbf{x}_{n-1} \oplus \mathbf{v}_c$ $\mathbf{u}_n = \mathbf{x}_n / 2^{32}$, where $\mathbf{x}_n = (x_{s(n-1)+1}, x_{s(n-1)+2}, \dots, x_{sn})$, $\mathbf{u}_n = (u_{s(n-1)+1}, u_{s(n-1)+2}, \dots, u_{sn})$ and *s* is the dimension of quasi-random vector.	$r[i-1] = x_i$

Notes:

- *Lo*(*x*) means obtaining lower 32 bits of the 64-bit unsigned integer *x*, that is, $Lo(x) = x \bmod 2^{32}$.
- *Hi*(*x*) means obtaining upper 32 bits of the 64-bit unsigned integer *x*, that is, $Hi(x) = \lfloor x / 2^{32} \rfloor$.

So, when you generate an integer sequence of *n* elements, the output array *r*[*i*] of the function `viRngUniformBits` comprises:

- *n* elements for the basic generators MCG31m1, R250, MRG32k3a, MT19937, MT2203, SOBOL, and NIEDERR
- 2*n* elements for the basic generator MCG59
- 4*n* elements for the basic generator WH.

You may use the integer output, in particular, for fast generation of bit vectors. However, in this case some bits (or groups of them) may happen to 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 http://www.intel.com/software/products/mkl for test results summary.

Bernoulli (VSL_METHOD_IBERNOULLI_ICDF)

Bernoulli distribution with the 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 http://www.intel.com/software/products/mkl for test results summary.

Geometric (VSL_METHOD_IGEOMETRIC_ICDF)

Geometrical distribution with the 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 http://www.intel.com/software/products/mkl for test results summary.

Binomial (VSL_METHOD_IBINOMIAL_BTPE)

Binomial distribution with the parameters $ntrial$ and p. If $ntrial \cdot \min(p, 1-p) \ge 30$, random numbers of the binomial distribution are generated by BTPE method (see [Kach88] for details), otherwise combination of inverse transformation and table lookup methods is used. 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 squeezing technique is applied.

See http://www.intel.com/software/products/mkl for test results summary and performance graphs.

Hypergeometric (VSL_METHOD_IHYPERGEOMETRIC_H2PE)

Hypergeometric distribution with the 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 H2PE method (see [Kach85] for details), otherwise by the inverse transformation method in combination with the table lookup method. 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, squeezing technique is applied.

See http://www.intel.com/software/products/mkl for test results summary and performance graphs.

Poisson (VSL_METHOD_IPOISSON_PTPE)

Poisson distribution with the parameter λ. If $\lambda \geq 27$, random numbers are generated by PTPE method (see [Schmeiser81] for details), otherwise combination of inverse transformation and table lookup methods is used. 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 squeezing technique is applied.

See http://www.intel.com/software/products/mkl for test results summary and performance graphs.

Poisson (VSL_METHOD_IPOISSON_POISNORM)

Poisson distribution with the 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_METHOD_SGAUSSIAN_BOXMULLER2 method is used to generate random numbers of normal distribution.

See http://www.intel.com/software/products/mkl for test results summary and performance graphs.

PoissonV (VSL_METHOD_IPOISSONV_POISNORM)

Poisson distribution with the 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_METHOD_SGAUSSIAN_BOXMULLER2 method is used to generate random numbers of normal distribution.

See http://www.intel.com/software/products/mkl for test results summary and performance graphs.

NegBinomial (VSL_METHOD_INEGBINOMIAL_NBAR)

Negative binomial distribution with the parameters a and p. If $(a-1)(1-p)/p \geq 100$, the random numbers are generated by NBAR method, otherwise by combination of inverse transformation and table lookup methods. 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 http://www.intel.com/software/products/mkl for test results summary and performance graphs.

Bibliography

[Ant79] Antonov, I.A., and Saleev, V.M. An economic method of computing LP_τ-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.

[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.

[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.

[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.

[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/%7Em-mat/MT/MT2002/emt19937ar.html

[NAG] Numerical Algorithms Group, www.nag.co.uk.

[Ripley87] Ripley, B.D. Stochastic Simulation, Wiley, New York, 1987.

[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.