NAG Library Chapter Introduction

g05 – Random Number Generators

Contents

1 Scope of the Chapter

This chapter is concerned with the generation of sequences of independent pseudorandom and quasi-random numbers from various distributions, and models.

2 Background to the Problems

2.1 Pseudorandom Numbers

A sequence of pseudorandom numbers is a sequence of numbers generated in some systematic way such that they are independent and statistically indistinguishable from a truly random sequence. A pseudorandom number generator (PRNG) is a mathematical algorithm that, given an initial state, produces a sequence of pseudorandom numbers. A PRNG has several advantages over a true random number generator in that the generated sequence is repeatable, has known mathematical properties and can be implemented without needing any specialist hardware. Many books on statistics and computer science have good introductions to PRNGs, for example Knuth (1981) or Banks (1998).

PRNGs can be split into base generators, and distributional generators. Within the context of this document a base generator is defined as a PRNG that produces a sequence (or stream) of variates (or values) uniformly distributed over the interval $(0,1)$. Depending on the algorithm being considered, this interval may be open, closed or half-closed. A distribution generator is a function that takes variates generated from a base generator and transforms them into variates from a specified distribution, for example a uniform, Gaussian (Normal) or gamma distribution.

The period (or cycle length) of a base generator is defined as the maximum number of values that can be generated before the sequence starts to repeat. The initial state of the base generator is often called the seed.

There are six base generators currently available in the NAG C Library, these are; a basic linear congruential generator (LCG) (referred to as the NAG basic generator) (see Knuth (1981)), two sets of Wichmann–Hill generators (see Maclaren (1989) and Wichmann and Hill (2006)), the Mersenne Twister (see Matsumoto and Nishimura (1998)), the ACORN generator (see Wikramaratna (1989)) and L'Ecuyer generator (see L'Ecuyer and Simard (2002)).

2.1.1 NAG Basic Generator

The NAG basic generator is a linear congruential generator (LCG) and, like all linear congruential generators, has the form:

$$\begin{aligned} x_i &= a_1 x_{i-1} \bmod m_1, \\ u_i &= \tfrac{x_i}{m_1}, \end{aligned}$$

where the u_i, for $i = 1, 2, \ldots$, form the required sequence.

The NAG basic generator uses $a_1 = 13^{13}$ and $m_1 = 2^{59}$, which gives a period of approximately 2^{57}.

This generator has been part of the NAG C Library since Mark 1 and as such has been widely used. It suffers from no known problems, other than those due to the lattice structure inherent in all linear congruential generators, and, even though the period is relatively short compared to many of the newer generators, it is sufficiently large for many practical problems.

The performance of the NAG basic generator has been analysed by the Spectral Test, see Section 3.3.4 of Knuth (1981), yielding the following results in the notation of Knuth (1981).

n	ν_n	Upper bound for ν_n
2	3.44×10^8	4.08×10^8
3	4.29×10^5	5.88×10^5
4	1.72×10^4	2.32×10^4
5	1.92×10^3	3.33×10^3
6	593	939
7	198	380
8	108	197
9	67	120

The right-hand column gives an upper bound for the values of ν_n attainable by any multiplicative congruential generator working modulo 2^{59}.

An informal interpretation of the quantities ν_n is that consecutive n-tuples are statistically uncorrelated to an accuracy of $1/\nu_n$. This is a theoretical result; in practice the degree of randomness is usually much greater than the above figures might support. More details are given in Knuth (1981), and in the references cited therein.

Note that the achievable accuracy drops rapidly as the number of dimensions increases. This is a property of all multiplicative congruential generators and is the reason why very long periods are needed even for samples of only a few random numbers.

2.1.2 Wichmann–Hill I Generator

This series of Wichmann–Hill base generators (see Maclaren (1989)) use a combination of four linear congruential generators and has the form:

$$\begin{aligned} w_i &= a_1 w_{i-1} \bmod m_1 \\ x_i &= a_2 x_{i-1} \bmod m_2 \\ y_i &= a_3 y_{i-1} \bmod m_3 \\ z_i &= a_4 z_{i-1} \bmod m_4 \\ u_i &= \left(\frac{w_i}{m_1} + \frac{x_i}{m_2} + \frac{y_i}{m_3} + \frac{z_i}{m_4}\right) \bmod 1, \end{aligned} \tag{1}$$

where the u_i, for $i = 1, 2, \ldots$, form the required sequence. The NAG C Library implementation includes 273 sets of parameters, a_j, m_j, for $j = 1, 2, 3, 4$, to choose from.

The constants a_i are in the range 112 to 127 and the constants m_j are prime numbers in the range 16718909 to 16776971, which are close to $2^{24} = 16777216$. These constants have been chosen so that each of the resulting 273 generators are essentially independent, all calculations can be carried out in 32-bit integer arithmetic and the generators give good results with the spectral test, see Knuth (1981) and Maclaren (1989). The period of each of these generators would be at least 2^{92} if it were not for common factors between $(m_1 - 1)$, $(m_2 - 1)$, $(m_3 - 1)$ and $(m_4 - 1)$. However, each generator should still have a period of at least 2^{80}. Further discussion of the properties of these generators is given in Maclaren (1989).

2.1.3 Wichmann–Hill II Generator

This Wichmann–Hill base generator (see Wichmann and Hill (2006)) is of the same form as that described in Section 2.1.2, i.e., a combination of four linear congruential generators. In this case $a_1 = 11600$, $m_1 = 2147483579$, $a_2 = 47003$, $m_2 = 2147483543$, $a_3 = 23000$, $m_3 = 2147483423$, $a_4 = 33000$, $m_4 = 2147483123$.

Unlike in the original Wichmann–Hill generator, these values are too large to carry out the calculations detailed in (1) using 32-bit integer arithmetic, however, if

$$w_i = 11600\text{endgroupw}_{\text{i}-1} \bmod 2147483579$$

then setting

$$W_i = 11600(w_{i-1} \bmod 185127) - 10379(w_{i-1}/185127)$$

gives

$$w_i = \begin{cases} W_i & \text{if } W_i \geq 0 \\ 2147483579 + W_i & \text{otherwise} \end{cases}$$

and W_i can be calculated in 32-bit integer arithmetic. Similar expressions exist for x_i, y_i and z_i. The period of this generator is approximately 2^{121}.

Further details of implementing this algorithm and its properties are given in Wichmann and Hill (2006). This paper also gives some useful guidelines on testing PRNGs.

2.1.4 Mersenne Twister Generator

The Mersenne Twister (see Matsumoto and Nishimura (1998)) is a twisted generalized feedback shift register generator. The algorithm underlying the Mersenne Twister is as follows:

(i) Set some arbitrary initial values $x_1, x_2, \ldots, x_r$, each consisting of w bits.

(ii) Letting

$$A = \begin{pmatrix} 0 & I_{w-1} \\ a_w & a_{w-1} \cdots a_1 \end{pmatrix},$$

where I_{w-1} is the $(w-1) \times (w-1)$ identity matrix and each of the $a_i, i = 1$ to w take a value of either 0 or 1 (i.e., they can be represented as bits). Define

$$x_{i+r} = \left(x_{i+s} \oplus \left(x_i^{(\omega:(l+1))} | x_{i+1}^{(l:1)}\right) A\right),$$

where $x_i^{(\omega:(l+1))} | x_{i+1}^{(l:1)}$ indicates the concatenation of the most significant (upper) $w - l$ bits of x_i and the least significant (lower) l bits of x_{i+1}.

(iii) Perform the following operations sequentially:

$$\begin{array}{lcl} z & = & x_{i+r} \oplus (x_{i+r} \gg t_1) \\ z & = & z \oplus ((z \ll t_2) \text{ AND } m_1) \\ z & = & z \oplus ((z \ll t_3) \text{ AND } m_2) \\ z & = & z \oplus (z \gg t_4) \\ u_{i+r} & = & z/(2^w - 1), \end{array}$$

where t_1, t_2, t_3 and t_4 are integers and m_1 and m_2 are bit-masks and '$\gg t$' and '$\ll t$' represent a t bit shift right and left respectively, $\oplus$ is bit-wise exclusively or (xor) operation and 'AND' is a bit-wise and operation.

The u_{i+r}, for $i = 1, 2, \ldots$, form the required sequence. The supplied implementation of the Mersenne Twister uses the following values for the algorithmic constants:

$$\begin{array}{lcl} w & = & 32 \\ a & = & \text{0x9908b0df} \\ l & = & 31 \\ r & = & 624 \\ s & = & 397 \\ t_1 & = & 11 \\ t_2 & = & 7 \\ t_3 & = & 15 \\ t_4 & = & 18 \\ m_1 & = & \text{0x9d2c5680} \\ m_2 & = & \text{0xefc60000} \end{array}$$

where the notation 0x*DD*... indicates the bit pattern of the integer whose hexadecimal representation is *DD*....

This algorithm has a period length of approximately $2^{19,937} - 1$ and has been shown to be uniformly distributed in 623 dimensions (see Matsumoto and Nishimura (1998)).

2.1.5 ACORN Generator

The ACORN generator is a special case of a multiple recursive generator (see Wikramaratna (1989) and Wikramaratna (2007)). The algorithm underlying ACORN is as follows:

(i) Choose an integer value $k \geq 1$.

(ii) Choose an integer value M, and an integer seed $Y_0^{(0)}$, such that $0 < Y_0^{(0)} < M$ and $Y_0^{(0)}$ and M are relatively prime.

(iii) Choose an arbitrary set of k initial integer values, $Y_0^{(1)}, Y_0^{(2)}, \ldots, Y_0^{(k)}$, such that $0 \leq Y_0^{(m)} < M$, for all $m = 1, 2, \ldots, k$.

(iv) Perform the following sequentially:

$$Y_i^{(m)} = \left(Y_i^{(m-1)} + Y_{i-1}^{(m)} \right) \bmod M$$

for $m = 1, 2, \ldots, k$.

(v) Set $u_i = Y_i^{(k)}/M$.

The u_i, for $i = 1, 2, \ldots$, then form a pseudorandom sequence, with $u_i \in [0, 1)$, for all i.

Although you can choose any value for k, M, $Y_0^{(0)}$ and the $Y_0^{(m)}$, within the constraints mentioned in (i) to (iii) above, it is recommended that $k \geq 10$, M is chosen to be a large power of two with $M \geq 2^{60}$ and $Y_0^{(0)}$ is chosen to be odd.

The period of the ACORN generator, with the modulus M equal to a power of two, and an odd value for $Y_0^{(0)}$ has been shown to be an integer multiple of M (see Wikramaratna (1992)). Therefore, increasing M will give a series with a longer period.

2.1.6 L'Ecuyer MRG32k3a Combined Recursive Generator

The base generator L'Ecuyer MRG32k3a (see L'Ecuyer and Simard (2002)) combines two multiple recursive generators:

$$\begin{aligned} x_i &= (a_{11}x_{i-1} + a_{12}x_{i-2} + a_{13}x_{i-3}) \bmod m_1 \\ y_i &= (a_{21}y_{i-1} + a_{22}y_{i-2} + a_{23}y_{i-3}) \bmod m_2 \\ z_i &= (x_i - y_i) \bmod m_1 \\ u_i &= (z_i + 1)/d \end{aligned}$$

where $a_{11} = 0$, $a_{12} = 1403580$, $a_{13} = -810728$, $m_1 = 2^{32} - 209$, $a_{21} = 527612$, $a_{22} = 0$, $a_{23} = -1370589$, $m_2 = 2^{32} - 22853$, and $u_i, i = 1, 2, \ldots$ form the required sequence. If $d = m_1$ then $u_i \in (0, 1]$ else if $d = m_1 + 1$ then $u_i \in (0, 1)$. Combining the two multiple recursive generators (MRG) results in sequences with better statistical properties in high dimensions and longer periods compared with those generated from a single MRG. The combined generator described above has a period length of approximately 2^{191}.

2.2 Quasi-random Numbers

Low discrepancy (quasi-random) sequences are used in numerical integration, simulation and optimization. Like pseudorandom numbers they are uniformly distributed but they are not statistically independent, rather they are designed to give more even distribution in multidimensional space (uniformity). Therefore they are often more efficient than pseudorandom numbers in multidimensional Monte–Carlo methods.

The quasi-random number generators implemented in this chapter generate a set of points $x^1, x^2, \ldots, x^N$ with high uniformity in the S-dimensional unit cube $I^S = [0, 1]^S$. One measure of the uniformity is the discrepancy which is defined as follows:

> Given a set of points $x^1, x^2, \ldots, x^N \in I^S$ and a subset $G \subset I^S$, define the counting function $S_N(G)$ as the number of points $x^i \in G$. For each $x = (x_1, x_2, \ldots, x_S) \in I^S$, let G_x be the rectangular S-dimensional region

$$G_x = [0, x_1) \times [0, x_2) \times \cdots \times [0, x_S)$$

with volume $x_1, x_2, \ldots, x_S$. Then the discrepancy of the points $x^1, x^2, \ldots, x^N$ is

$$D_N^*\left(x^1, x^2, \ldots, x^N\right) = \sup_{x \in I^S} \left| S_N(G_x) - N \sum_{k=1}^{S} x_k \right|.$$

The discrepancy of the first N terms of such a sequence has the form

$$D_N^*\left(x^1, x^2, \ldots, x^N\right) \leq C_S (\log N)^S + O\Big((\log N)^{S-1}\Big) \quad \text{for all} \quad N \geq 2.$$

The principal aim in the construction of low-discrepancy sequences is to find sequences of points in I^S with a bound of this form where the constant C_S is as small as possible.

Three types of low-discrepancy sequences are supplied in this library, these are due to Sobol, Faure and Niederreiter. Two sets of Sobol sequences are supplied, the first is based on work of Joe and Kuo (2008) and the second on the work of Bratley and Fox (1988). More information on quasi-random number generation and the Sobol, Faure and Niederreiter sequences in particular can be found in Bratley and Fox (1988) and Fox (1986).

The efficiency of a simulation exercise may often be increased by the use of variance reduction methods (see Morgan (1984)). It is also worth considering whether a simulation is the best approach to solving the problem. For example, low-dimensional integrals are usually more efficiently calculated by functions in Chapter d01 rather than by Monte–Carlo integration.

2.3 Scrambled Quasi-random Numbers

Scrambled quasi-random sequences are an extension of standard quasi-random sequences that attempt to eliminate the bias inherent in a quasi-random sequence whilst retaining the low-discrepancy properties. The use of a scrambled sequence allows error estimation of Monte–Carlo results by performing a number of iterates and computing the variance of the results.

This implementation of scrambled quasi-random sequences is based on TOMS algorithm 823 and details can be found in the accompanying paper, Hong and Hickernell (2003). Three methods of scrambling are supplied; the first a restricted form of Owen's scrambling (Owen (1995)), the second based on the method of Faure and Tezuka (2000) and the last method combines the first two.

Scrambled versions of both Sobol sequences and the Niederreiter sequence can be obtained.

2.4 Non-uniform Random Numbers

Random numbers from other distributions may be obtained from the uniform random numbers by the use of transformations and rejection techniques, and for discrete distributions, by table based methods.

(a) Transformation Methods

For a continuous random variable, if the cumulative distribution function (CDF) is $F(x)$ then for a uniform $(0, 1)$ random variate u, $y = F^{-1}(u)$ will have CDF $F(x)$. This method is only efficient in a few simple cases such as the exponential distribution with mean μ, in which case $F^{-1}(u) = -\mu \log(u)$. Other transformations are based on the joint distribution of several random variables. In the bivariate case, if v and w are random variates there may be a function g such that $y = g(v, w)$ has the required distribution; for example, the Student's t-distribution with n degrees of freedom in which v has a Normal distribution, w has a gamma distribution and $g(v, w) = v\sqrt{n/w}$.

(b) Rejection Methods

Rejection techniques are based on the ability to easily generate random numbers from a distribution (called the envelope) similar to the distribution required. The value from the envelope distribution is then accepted as a random number from the required distribution with a certain probability; otherwise, it is rejected and a new number is generated from the envelope distribution.

(c) Table Search Methods

For discrete distributions, if the cumulative probabilities, $P_i = \text{Prob}(x \leq i)$, are stored in a table then, given u from a uniform $(0,1)$ distribution, the table is searched for i such that $P_{i-1} < u \leq P_i$. The returned value i will have the required distribution. The table searching can be made faster by means of an index, see Ripley (1987). The effort required to set up the table and its index may be considerable, but the methods are very efficient when many values are needed from the same distribution.

2.5 Copulas

A copula is a function that links the univariate marginal distributions with their multivariate distribution. Sklar's theorem (see Sklar (1973)) states that if f is an m-dimensional distribution function with continuous margins $f_1, f_2, \ldots, f_m$, then f has a unique copula representation, c, such that

$$f(x_1, x_2, \ldots, x_m) = c(f_1(x_1), f_2(x_2), \ldots, f_m(x_m))$$

The copula, c, is a multivariate uniform distribution whose dependence structure is defined by the dependence structure of the multivariate distribution f, with

$$c(u_1, u_2, \ldots, u_m) = f\big(f_1^{-1}(u_1), f_2^{-1}(u_2), \ldots, f_m^{-1}(u_m)\big)$$

where $u_i \in [0,1]$. This relationship can be used to simulate variates from distributions defined by the dependence structure of one distribution and each of the marginal distributions given by another. For additional information see Nelsen (1998) or Boye (Unpublished manuscript) and the references therein.

2.6 Brownian Bridge

2.6.1 Brownian Bridge Process

Fix two times $t_0 < T$ and let $W = (W_t)_{0 \leq t \leq T-t_0}$ be a standard d-dimensional Wiener process on the interval $[0, T-t_0]$. Recall that the terms Wiener process and Brownian motion are often used interchangeably.

A *standard* d-dimensional Brownian bridge $B = (B_t)_{t_0 \leq t \leq T}$ on $[t_0, T]$ is defined (see Revuz and Yor (1999)) as

$$B_t = W_{t-t_0} - \frac{t-t_0}{T-t_0} W_{T-t_0}.$$

The process is continuous, starts at zero at time t_0 and ends at zero at time T. It is Gaussian, has zero mean and has a covariance structure given by

$$\mathbb{E}\big(B_s B_t^{\mathrm{T}}\big) = \frac{(s-t_0)(T-t)}{T-t_0} I_d$$

for any $s \leq t$ in $[t_0, T]$ where I_d is the d-dimensional identity matrix. The Brownian bridge is often called a non-free or ‘pinned’ Wiener process since it is forced to be 0 at time T, but is otherwise very similar to a standard Wiener process.

We can generalize this construction as follows. Fix points $x, w \in \mathbb{R}^d$, let Σ be a $d \times d$ covariance matrix and choose any $d \times d$ matrix C such that $CC^{\mathrm{T}} = \Sigma$. The *generalized* d-dimensional Brownian bridge $X = (X_t)_{t_0 \leq t \leq T}$ is defined by setting

$$X_t = \frac{(t-t_0)w + (T-t)x}{T-t_0} + CB_t = \frac{(t-t_0)w + (T-t)x}{T-t_0} + CW_{t-t_0} - \frac{(t-t_0)}{T-t_0} CW_{T-t_0}$$

for all $t \in [t_0, T]$. The process X is continuous, starts at x at time t_0 and ends at w at time T. It has mean $((t-t_0)w + (T-t)x)/(T-t_0)$ and covariance structure

$$\mathbb{E}(X_s - \mathbb{E}X_s)(X_t - \mathbb{E}X_t)^{\mathrm{T}} = \mathbb{E}\big(CB_s B_t^{\mathrm{T}} C^{\mathrm{T}}\big) = \frac{(s-t_0)(T-t)}{T-t_0} \Sigma$$

for all $s \leq t$ in $[t_0, T]$. This is a non-free Wiener process since it is forced to be equal to w at time T. However if we set $w = x + CW_{T-t_0}$, then X simplifies to

$$X_t = x + CW_{t-t_0}$$

for all $t \in [t_0, T]$ which is nothing other than a d-dimensional Wiener process with covariance given by Σ.

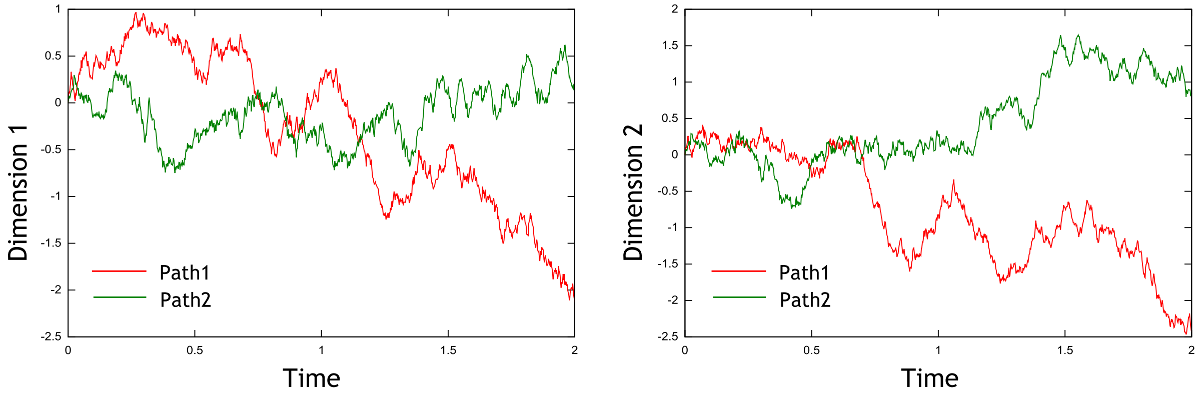

Figure 1
Two sample paths for a two-dimensional free Wiener process

Figure 1 shows two sample paths for a two-dimensional free Wiener process $X = \left(X_t^1, X_t^2\right)_{0 \leq t \leq 2}$. The correlation coefficient between the one-dimensional processes X^1 and X^2 at any time is $\rho = 0.80$. Note that the red and green paths in each figure are uncorrelated, however it is fairly evident that the two red paths are correlated, and that the two green paths are correlated (when one path increases so does the other, and vice versa).

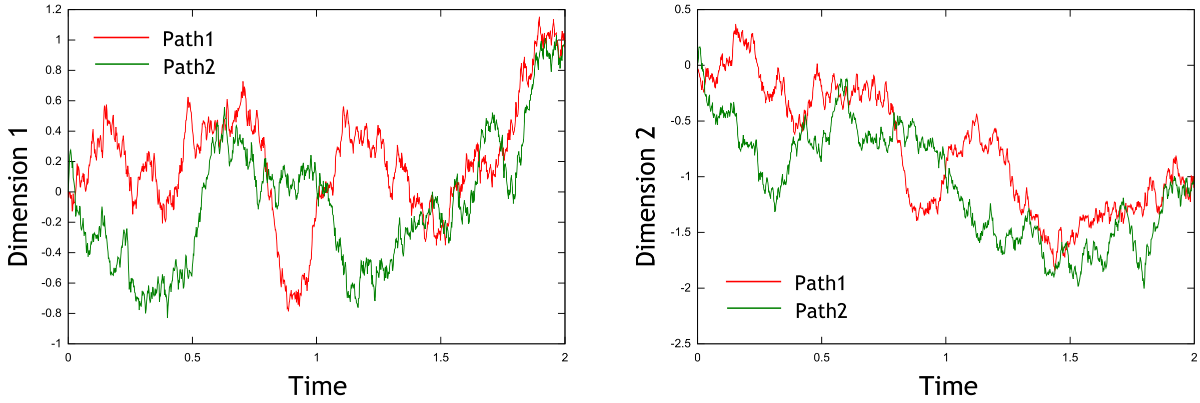

Figure 2
Two sample paths for a two-dimensional non-free Wiener process. The process starts at $(0, 0)$ and ends at $(1, -1)$

Figure 2 shows two sample paths for a two-dimensional non-free Wiener process. The process starts at $(0, 0)$ and ends at $(1, -1)$. The correlation coefficient between the one-dimensional processes is again $\rho = 0.80$. The red and green paths in each figure are uncorrelated, while the two red paths tend to increase and decrease together, as do the two green paths. Both Figure 1 and Figure 2 were constructed using nag_rand_bb (g05xbc).

2.6.2 Brownian Bridge Algorithm

The ideas above can also be used to construct sample paths of a free or non-free Wiener process (recall that a non-free Wiener process is the Brownian bridge process outlined above). Fix two times $t_0 < T$ and let $(t_i)_{1 \leq i \leq N}$ be any set of time points satisfying $t_0 < t_1 < t_2 < \cdots < t_N < T$. Let $(X_{t_i})_{1 \leq i \leq N}$ denote a d-dimensional (free or non-free) Wiener sample path at these times. These values can be generated by the so-called *Brownian bridge algorithm* (see Glasserman (2004)) which works as follows. From any two known points X_{t_i} at time t_i and X_{t_k} at time t_k with $t_i < t_k$, a new point X_{t_j} can be interpolated at any time $t_j \in (t_i, t_k)$ by setting

$$X_{t_j} = \frac{X_{t_i}(t_k - t_j) + X_{t_k}(t_j - t_i)}{t_k - t_i} + CZ\sqrt{\frac{(t_k - t_j)(t_j - t_i)}{(t_k - t_i)}} \tag{2}$$

where Z is a d-dimensional standard Normal random variable and C is any $d \times d$ matrix such that CC^{T} is the desired covariance structure for the (free or non-free) Wiener process X. Clearly this algorithm is iterative in nature. All that is needed to complete the specification is to fix the start point X_{t_0} and end point X_T, and to specify how successive interpolation times t_j are chosen. For X to behave like a usual (free) Wiener process we should set X_{t_0} equal to some value $x \in \mathbb{R}^d$ and then set $X_T = x + C\sqrt{T - t_0}Z$ where Z is any d-dimensional standard Normal random variable. However when it comes to deciding how the successive interpolation times t_j should be chosen, there is virtually no restriction. Any method of choosing which $t_j \in (t_i, t_k)$ to interpolate next is equally valid, provided t_i is the nearest known point to the left of t_j and t_k is the nearest known point to the right of t_j. In other words, the interpolation interval (t_i, t_k) must not contain any other known points, otherwise the covariance structure of the process will be incorrect.

The order in which the successive interpolation times t_j are chosen is called the *bridge construction order*. Since all construction orders will produce a correct process, the question arises whether one construction order should be preferred over another. When the Z values are drawn from a pseudorandom generator, the answer is typically no. However the bridge algorithm is frequently used with quasi-random numbers, and in this case the bridge construction order can be important.

2.6.3 Bridge Construction Order and Quasi-random Sequences

Consider the one-dimensional case of a free Wiener process where $d = C = 1$. The Brownian bridge is frequently combined with low-discrepancy (quasi-random) sequences to perform quasi-Monte–Carlo integration. Quasi-random points $Z^1, Z^2, Z^3, \ldots$ are generated from the standard Normal distribution, where each quasi-random point $Z^i = \left(Z^i_1, Z^i_2, \cdots, Z^i_D\right)$ consists of D one-dimensional values. The process X starts at $X_{t_0} = x$ which is known. There remain $N + 1$ time points at which the bridge is to be computed, namely $(X_{t_i})_{1 \leq i \leq N}$ and X_T (recall we are considering a free Wiener process). In this case D is set equal to $N + 1$, so that $N + 1$ dimensional quasi-random points are generated. A single quasi-random point is used to construct one Wiener sample path.

The question is how to use the dimension values of each $N + 1$ dimensional quasi-random point. Often the 'lower' dimension values (Z^i_1, Z^i_2, etc.) display better uniformity properties than the 'higher' dimension values (Z^i_{N+1}, Z^i_N, etc.) so that the 'lower' dimension values should be used to construct the most *important* sections of the sample path. For example, consider a model which is particularly sensitive to the behaviour of the underlying process at time 3. When constructing the sample paths, one would therefore ensure that time 3 was one of the interpolation points of the bridge, and that a 'lower' dimension value was used in (2) to construct the corresponding bridge point X_3. Indeed, one would most likely also ensure that time X_3 was one of the *first* bridge points that was constructed: 'lower' dimension values would be used to construct both the left and right bridge points used in (2) to interpolate X_3, so that the distribution of X_3 benefits as much as possible from the uniformity properties of the quasi-random sequence. For further discussions in this regard we refer to Glasserman (2004). These remarks extend readily to the case of a non-free Wiener process.

2.6.4 Brownian Bridge and Stochastic Differential Equations

The Brownian bridge algorithm, especially when combined with quasi-random variates, is frequently used to obtain numerical solutions to stochastic differential equations (SDEs) driven by (free or non-free) Wiener processes. The quasi-random variates produce a family of Wiener sample paths which cover the space of all Wiener sample paths fairly evenly. This is analogous to the way in which a two-dimensional quasi-random sequence covers the unit square $[0, 1]^2$ evenly. When solving SDEs one is typically interested in the *increments* of the driving Wiener process between two time points, rather than the value of the process at a particular time point. Section 3.3 contains details on which functions can be used to obtain such Wiener increments.

2.7 Random Fields

A random field is a stochastic process, taking values in a Euclidean space, and defined over a parameter space of dimensionality at least one. They are often used to simulate some physical space-dependent parameter, such as the permeability of rock, which cannot be measured at every point in the space. The simulated values can then be used to model other dependent quantities, for example, underground flow of water, often through the use of partial differential equations (PDEs).

A d-dimensional random field $Z(\mathbf{x})$ is a function which is random at every point $(\mathbf{x} \in D)$ for some domain $D \subset \mathbb{R}^d$, so $Z(\mathbf{x})$ is a random variable for each $\mathbf{x}$. The random field has a mean function $\mu(\mathbf{x}) = \mathbb{E}[Z(\mathbf{x})]$ and a symmetric positive semidefinite covariance function $C(\mathbf{x}, \mathbf{y}) = \mathbb{E}[(Z(\mathbf{x}) - \mu(\mathbf{x}))(Z(\mathbf{y}) - \mu(\mathbf{y}))]$.

A random field, $Z(\mathbf{x})$, is a Gaussian random field if, for any choice of $n \in \mathbb{N}$ and $\mathbf{x}_1, \dots, \mathbf{x}_n \in \mathbb{R}^d$, the random vector $[Z(\mathbf{x}_1), \dots, Z(\mathbf{x}_n)]^{\mathrm{T}}$ follows a multivariate Gaussian distribution.

A Gaussian random field $Z(\mathbf{x})$ is stationary if $\mu(\mathbf{x})$ is constant for all $\mathbf{x} \in \mathbb{R}$ and $C(\mathbf{x}, \mathbf{y}) = C(\mathbf{x} + \mathbf{a}, \mathbf{y} + \mathbf{a})$ for all $\mathbf{x}, \mathbf{y}, \mathbf{a} \in \mathbb{R}^d$ and hence we can express the covariance function $C(\mathbf{x}, \mathbf{y})$ as a function γ of one variable: $C(\mathbf{x}, \mathbf{y}) = \gamma(\mathbf{x} - \mathbf{y})$. γ is known as a variogram (or more correctly, a semivariogram) and includes the multiplicative factor σ^2 representing the variance such that $\gamma(0) = \sigma^2$. There are a number of commonly used variograms, including:

Symmetric stable variogram

$$\gamma(x) = \sigma^2 \exp\bigl(-(x')^\nu\bigr).$$

Cauchy variogram

$$\gamma(x) = \sigma^2 \Bigl(1 + (x')^2\Bigr)^{-\nu}.$$

Differential variogram with compact support

$$\gamma(x) = \begin{cases} \sigma^2 \Bigl(1 + 8x' + 25(x')^2 + 32(x')^3\Bigr)(1 - x')^8, & x' < 1, \\ 0, & x' \geq 1. \end{cases}$$

Exponential variogram

$$\gamma(x) = \sigma^2 \exp(-x').$$

Gaussian variogram

$$\gamma(x) = \sigma^2 \exp\Bigl(-(x')^2\Bigr).$$

Nugget variogram

$$\gamma(x) = \begin{cases} \sigma^2, & x = 0, \\ 0, & x \neq 0. \end{cases}$$

Spherical variogram

$$\gamma(x) = \begin{cases} \sigma^2 \Bigl(1 - 1.5x' + 0.5(x')^3\Bigr), & x' < 1, \\ 0, & x' \geq 1. \end{cases}$$

Bessel variogram

$$\gamma(x) = \sigma^2 \frac{2^\nu \Gamma(\nu + 1) J_\nu(x')}{(x')^\nu},$$

Hole effect variogram

$$\gamma(x) = \sigma^2 \frac{\sin(x')}{x'}.$$

Whittle–Matérn variogram

$$\gamma(x)=\sigma^2\frac{2^{1-\nu}(x')^\nu K_\nu(x')}{\Gamma(\nu)}.$$

Continuously parameterised variogram with compact support

$$\gamma(x)=\begin{cases}\sigma^2\frac{2^{1-\nu}(x')^\nu K_\nu(x')}{\Gamma(\nu)}\left(1+8x''+25(x'')^2+32(x'')^3\right)(1-x'')^8, & x''<1,\\ 0, & x''\geq 1.\end{cases}$$

Generalized hyperbolic distribution variogram

$$\gamma(x)=\sigma^2\frac{\left(\delta^2+(x')^2\right)^{\frac{\lambda}{2}}}{\delta^\lambda K_\lambda(\kappa\delta)}K_\lambda\left(\kappa\left(\delta^2+(x')^2\right)^{\frac{1}{2}}\right).$$

Cosine variogram

$$\gamma(x)=\sigma^2\cos(x').$$

Where x' is a scaled norm of x.

2.8 Sampling

The term sampling can have a number of different meanings. Here we are using it to mean randomly selecting one or more observations or records from a particular dataset. Sampling can be performed in one of two ways:

With replacement:
: where each observation in the original dataset can appear multiple times in the sample. The sample can therefore be larger than the original dataset.

Without replacement:
: where each observation in the original dataset can appear at most once in the sample. The sample is therefore no larger than the original dataset.

Each of these sampling methods can be further divided into two categories:

With equal weights:
: where each observation in the original dataset has the same probability of appearing in the sample as every other observation.

With unequal weights:
: where the probability of an observation from the original dataset appearing in the sample is proportional to the weight assigned to that observation.

The need to sample from a dataset appears in many areas. For example, it forms the basis for: bootstrapping (sampling with replacement, usually using equal weights); cross-validation (sampling without replacement, using equal weights); importance sampling (sampling with replacement, using unequal weights); randomization of experimental units in designed experiments or reducing the size of large databases (sampling with replacement with either equal or unequal weights).

Rather than drawing a sample from the whole dataset it is sometimes desirable to take samples from different strata or subpopulations within that dataset, referred to as stratified sampling. Within each stratum one or more of the above sampling methods may be adopted.

2.9 Sampling Based Validation

Let (Y_o, X_o) denote a dataset of observed values from a known population, where Y_o is a matrix of one or more dependent or response variables and X_o a matrix of one more more independent variables or covariates. Let M denote a model described in terms β a vector of one or more unknown parameters. The purpose of model M is to describe the behaviour of the dependent variables in terms of the independent variables. In order to do this the parameter estimates must first be estimated and then how well the models fits, that is, how well it describes the dependent variables, assessed.

An example of such a model would be a simple linear regression as described in Section 2.3 in the g02 Chapter Introduction. The simple linear regression has two parameters, an intercept, β_0 and slope, β_1 and the observed dataset consists of the dependent variable y and the single independent variable x. The parameter estimates are usually obtained via least squares.

Given a set of parameter estimates and a matrix of independent variables one way of assessing how well a model fits is to use the model to predict the values of the dependent variable and compare these predictions to the observed values. Ideally two datasets will be involved, a **training dataset**, (Y_t, X_t), used to estimate the model parameters and a **validation dataset**, (Y_v, X_v), used for the prediction and comparison. These two datasets should be drawn independently from the same population. However, in practice, this is often not possible either because a second dataset can not be drawn from the same population or because the value of the dependent variables are unknowable (for example the dataset in question is a time series and the event of interest has not yet happened). Rather than use the same dataset as both the training and validation dataset, which leads to overfitting and hence an over estimation of how well the model fits, a sampling based validation method can be used.

In ***K*-fold cross-validation** the original dataset is randomly divided into K equally sized folds (or groups). The model fitting and assessment process is performed using a validation dataset consisting of those observations in the kth group and a training dataset consisting of all observations not in the kth group. This is repeated K times, with $k = 1, 2, \ldots, K$, and the results combined. **Repeated random sub-sampling validation** is similar, but rather than systematically dividing the original dataset into a training and validation dataset, whether an observation resides in a given dataset is chosen randomly each time the model fitting and assessment process is repeated.

2.10 Other Random Structures

In addition to random numbers from various distributions, random compound structures can be generated. These include random time series and random matrices.

2.11 Multiple Streams of Pseudorandom Numbers

It is often advantageous to be able to generate variates from multiple, independent, streams (or sequences) of random variates. For example when running a simulation in parallel on several processors. There are four ways of generating multiple streams using the functions available in this chapter:

(i) using different initial values (seeds);

(ii) using different generators;

(iii) skip ahead (also called block-splitting);

(iv) leap-frogging.

2.11.1 Multiple Streams via Different Initial Values (Seeds)

A different sequence of variates can be generated from the same base generator by initializing the generator using a different set of seeds. The statistical properties of the base generators are only guaranteed within, not between sequences. For example, two sequences generated from two different starting points may overlap if these initial values are not far enough apart. The potential for overlapping sequences is reduced if the period of the generator being used is large. In general, of the four methods for creating multiple streams described here, this is the least satisfactory.

The one exception to this is the Wichmann–Hill II generator. The Wichmann and Hill (2006) paper describes a method of generating blocks of variates, with lengths up to 2^{90}, by fixing the first three seed values of the generator (w_0, x_0 and y_0), and setting z_0 to a different value for each stream required. This is similar to the skip-ahead method described in Section 2.11.3, in that the full sequence of the Wichmann–Hill II generator is split into a number of different blocks, in this case with a fixed length of 2^{90}. But without the computationally intensive initialization usually required for the skip-ahead method.

2.11.2 Multiple Streams via Different Generators

Independent sequences of variates can be generated using a different base generator for each sequence. For example, sequence 1 can be generated using the NAG basic generator, sequence 2 using Mersenne Twister, sequence 3 the ACORN generator and sequence 4 using L'Ecuyer generator. The Wichmann–Hill I generator implemented in this chapter is, in fact, a series of 273 independent generators. The particular sub-generator to use is selected using the **subid** variable. Therefore, in total, 278 independent streams can be generated with each using a different generator (273 Wichmann–Hill I generators, and 5 additional base generators).

2.11.3 Multiple Streams via Skip-ahead

Independent sequences of variates can be generated from a single base generator through the use of block-splitting, or skipping-ahead. This method consists of splitting the sequence into k non-overlapping blocks, each of length n, where n is no smaller than the maximum number of variates required from any of the sequences. For example,

$$\underbrace{x_1, x_2, \ldots, x_n}_{\text{block 1}}, \underbrace{x_{n+1}, x_{n+2}, \ldots, x_{2n}}_{\text{block 2}}, \underbrace{x_{2n+1}, x_{2n+2}, \ldots, x_{3n}}_{\text{block 3}}, \text{etc.}$$

where $x_1, x_2, \ldots$ is the sequence produced by the generator of interest. Each of the k blocks provide an independent sequence.

The skip-ahead algorithm therefore requires the sequence to be advanced a large number of places, as to generate values from say, block b, you must skip over the $(b-1)n$ values in the first $b-1$ blocks. Owing to their form this can be done efficiently for linear congruential generators and multiple congruential generators. A skip-ahead algorithm is also provided for the Mersenne Twister generator.

Although skip-ahead requires some additional computation at the initialization stage (to 'fast forward' the sequence) no additional computation is required at the generation stage.

This method of producing multiple streams can also be used for the Sobol and Niederreiter quasi-random number generator via the argument **iskip** in nag_quasi_init (g05ylc).

2.11.4 Multiple Streams via Leap-frog

Independent sequences of variates can also be generated from a single base generator through the use of leap-frogging. This method involves splitting the sequence from a single generator into k disjoint subsequences. For example:

$$\begin{array}{rl} \text{Subsequence 1 :} & x_1, x_{k+1}, x_{2k+1}, \ldots \\ \text{Subsequence 2 :} & x_2, x_{k+2}, x_{2k+2}, \ldots \\ \vdots & \\ \text{Subsequence } k\text{:} & x_k, x_{2k}, x_{3k}, \ldots, \end{array}$$

where $x_1, x_2, \ldots$ is the sequence produced by the generator of interest. Each of the k subsequences then provides an independent stream of variates.

The leap-frog algorithm therefore requires the generation of every kth variate from the base generator. Owing to their form this can be done efficiently for linear congruential generators and multiple congruential generators. A leap-frog algorithm is provided for the NAG Basic generator, both the Wichmann–Hill I and Wichmann–Hill II generators and L'Ecuyer generator.

It is known that, dependent on the number of streams required, leap-frogging can lead to sequences with poor statistical properties, especially when applied to linear congruential generators. In addition, leap-frogging can increase the time required to generate each variate. Therefore leap-frogging should be avoided unless absolutely necessary.

2.11.5 Skip-ahead and Leap-frog for a Linear Congruential Generator (LCG): An Example

As an illustrative example, a brief description of the algebra behind the implementation of the leap-frog and skip-ahead algorithms for a linear congruential generator is given. A linear congruential generator has the form $x_{i+1} = a_1 x_i \bmod m_1$. The recursive nature of a linear congruential generator means that

$$\begin{aligned} x_{i+v} &= a_1 x_{i+v-1} \bmod m_1 \\ &= a_1 (a_1 x_{i+v-2} \bmod m_1) \bmod m_1 \\ &= a_1^2 x_{i+v-2} \bmod m_1 \\ &= a_1^v x_i \bmod m_1. \end{aligned}$$

The sequence can therefore be quickly advanced v places by multiplying the current state (x_i) by $a_1^v \bmod m_1$, hence skipping the sequence ahead. Leap-frogging can be implemented by using a_1^k, where k is the number of streams required, in place of a_1 in the standard linear congruential generator recursive formula, in order to advance k places, rather than one, at each iteration.

In a linear congruential generator the multiplier a_1 is constructed so that the generator has good statistical properties in, for example, the spectral test. When using leap-frogging to construct multiple streams this multiplier is replaced with a_1^k, and there is no guarantee that this new multiplier will have suitable properties especially as the value of k depends on the number of streams required and so is likely to change depending on the application. This problem can be emphasized by the lattice structure of linear congruential generators. Similiarly, the value of a_1 is often chosen such that the computation $a_1 x_i \bmod m_1$ can be performed efficiently. When a_1 is replaced by a_1^k, this is often no longer the case.

Note that, due to rounding, when using a distributional generator, a sequence generated using leap-frogging and a sequence constructed by taking every k value from a set of variates generated without leap-frogging may differ slightly. These differences should only affect the least significant digit.

2.11.6 Skip-ahead and Leap-frog for the Mersenne Twister: An Example

Skipping ahead with the Mersenne Twister generator is based on the definition of a $k \times k$ (where $k = 19937$) transition matrix, A, over the finite field $\mathbb{F}_2$ (with elements 0 and 1). Multiplying A by the current state x_n, represented as a vector of bits, produces the next state vector x_{n+1}:

$$x_{n+1} = A x_n.$$

Thus, skipping ahead v places in a sequence is equivalent to multiplying by A^v:

$$x_{n+v} = A^v x_n.$$

Since calculating A^v by a standard square and multiply algorithm is $O\big(k^3 \log(v)\big)$ and requires over 47MB of memory (see Haramoto *et al.* (2008)), an indirect calculation is performed which relies on a property of the characteristic polynomial $p(z)$ of A, namely that $p(A) = 0$. We then define

$$g(z) = z^v \bmod p(z) = a_{k-1} z^{k-1} + \ldots + a_1 z + a_0,$$

and observe that

$$g(z) = z^v + q(z)p(z)$$

for a polynomial $q(z)$. Since $p(A) = 0$, we have that $g(A) = A^v$ and

$$A^v x_n = \big(a_{k-1} A^{k-1} + \ldots + a_1 A + a_0 I\big) x_n.$$

This polynomial evaluation can be performed using Horner's method:

$$A^v x_n = A(\ldots A(A(A a_{k-1} x_n + a_{k-2} x_n) + a_{k-3} x_n) + \cdots + a_1 x_n) + a_0 x_n,$$

which reduces the problem to advancing the generator $k - 1$ places from state x_n and adding (where addition is as defined over $\mathbb{F}_2$) the intermediate states for which a_i is nonzero.

There are therefore two stages to skipping the Mersenne Twister ahead v places:

(i) Calculate the coefficients of the polynomial $g(z) = z^v \bmod p(z)$;

(ii) advance the sequence $k - 1$ places from the starting state and add the intermediate states that correspond to nonzero coefficients in the polynomial calculated in the first step.

The resulting state is that for position v in the sequence.

The cost of calculating the polynomial is $O\left(k^2\log(v)\right)$ and the cost of applying it to state is constant. Skip ahead functionality is typically used in order to generate n independent pseudorandom number streams (e.g., for separate threads of computation). There are two options for generating the n states:

(i) On the master thread calculate the polynomial for a skip ahead distance of v and apply this polynomial to state n times, after each iteration j saving the current state for later usage by thread j.

(ii) Have each thread j independently and in parallel with other threads calculate the polynomial for a distance of $(j+1)v$ and apply to the original state.

Since $\lim_{v\to\infty} \log(v) = \log nv$, then for large v the cost of generating the polynomial for a skip ahead distance of nv (i.e., the calculation performed by thread $n-1$ in option (ii) above) is approximately the same as generating that for a distance of v (i.e., the calculation performed by thread 0). However, only one application to state need be made per thread, and if n is sufficiently large the cost of applying the polynomial to state becomes the dominant cost in option (i), in which case it is desirable to use option (ii). Tests have shown that as a guideline it becomes worthwhile to switch from option (i) to option (ii) for approximately $n > 30$.

Leap frog calculations with the Mersenne Twister are performed by computing the sequence fully up to the required size and discarding the redundant numbers for a given stream.

3 Recommendations on Choice and Use of Available Functions

3.1 Pseudorandom Numbers

Before generating any pseudorandom variates the base generator being used must be initialized. Once initialized, a distributional generator can be called to obtain the variates required. No interfaces have been supplied for direct access to the base generators. If a sequence of random variates from a uniform distribution on the open interval $(0,1)$, is required, then the uniform distribution function (nag_rand_basic (g05sac)) should be called.

3.1.1 Initialization

Before generating any variates the base generator must be initialized. Two utility functions are provided for this, nag_rand_init_repeatable (g05kfc) and nag_rand_init_nonrepeatable (g05kgc), both of which allow any of the base generators to be chosen.

nag_rand_init_repeatable (g05kfc) selects and initializes a base generator to a repeatable (when executed serially) state: two calls of nag_rand_init_repeatable (g05kfc) with the same argument-values will result in the same subsequent sequences of random numbers (when both generated serially).

nag_rand_init_nonrepeatable (g05kgc) selects and initializes a base generator to a non-repeatable state in such a way that different calls of nag_rand_init_nonrepeatable (g05kgc), either in the same run or different runs of the program, will almost certainly result in different subsequent sequences of random numbers.

No utilities for saving, retrieving or copying the current state of a generator have been provided. All of the information on the current state of a generator (or stream, if multiple streams are being used) is stored in the integer array **state** and as such this array can be treated as any other integer array, allowing for easy copying, restoring, etc.

3.1.2 Repeated initialization

As mentioned in Section 2.11.1, it is important to note that the statistical properties of pseudorandom numbers are only guaranteed within sequences and not between sequences produced by the same generator. Repeated initialization will thus render the numbers obtained less rather than more independent. In a simple case there should be only one call to nag_rand_init_repeatable (g05kfc) or nag_rand_init_nonrepeatable (g05kgc) and this call should be before any call to an actual generation function.

3.1.3 Choice of Base Generator

If a single sequence is required then it is recommended that the Mersenne Twister is used as the base generator (**genid** = Nag_MersenneTwister). This generator is fast, has an extremely long period and has been shown to perform well on various test suites, see Matsumoto and Nishimura (1998), L'Ecuyer and Simard (2002) and Wichmann and Hill (2006) for example.

When choosing a base generator, the period of the chosen generator should be borne in mind. A good rule of thumb is never to use more numbers than the square root of the period in any one experiment as the statistical properties are impaired. For closely related reasons, breaking numbers down into their bit patterns and using individual bits may also cause trouble.

3.1.4 Choice of Method for Generating Multiple Streams

If the Wichmann–Hill II base generator is being used, and a period of 2^{90} is sufficient, then the method described in Section 2.11.1 can be used. If a different generator is used, or a longer period length is required then generating multiple streams by altering the initial values should be avoided.

Using a different generator works well if less than 277 streams are required.

Of the remaining two methods, both skip-ahead and leap-frogging use the sequence from a single generator, both guarantee that the different sequences will not overlap and both can be scaled to an arbitrary number of streams. Leap-frogging requires no *a-priori* knowledge about the number of variates being generated, whereas skip-ahead requires you to know (approximately) the maximum number of variates required from each stream. Skip-ahead requires no *a-priori* information on the number of streams required. In contrast leap-frogging requires you to know the maximum number of streams required, prior to generating the first value. Of these two, if possible, skip-ahead should be used in preference to leap-frogging. Both methods required additional computation compared with generating a single sequence, but for skip-ahead this computation occurs only at initialization. For leap-frogging additional computation is required both at initialization and during the generation of the variates. In addition, as mentioned in Section 2.11.4, using leap-frogging can, in some instances, change the statistical properties of the sequences being generated.

Leap-frogging is performed by calling nag_rand_leap_frog (g05khc) after the initialization function (nag_rand_init_repeatable (g05kfc) or nag_rand_init_nonrepeatable (g05kgc)). For skip-ahead, either nag_rand_skip_ahead (g05kjc) or nag_rand_skip_ahead_power2 (g05kkc) can be called. Of these, nag_rand_skip_ahead_power2 (g05kkc) restricts the amount being skipped to a power of 2, but allows for a large ‘skip’ to be performed.

3.1.5 Copulas

After calling one of the copula functions the inverse cumulative distribution function (CDF) can be applied to convert the uniform marginal distribution into the required form. Scalar and vector functions for evaluating the CDF, for a range of distributions, are supplied in Chapter g01. It should be noted that these functions are often described as computing the ‘deviates’ of the distribution.

When using the inverse CDF functions from Chapter g01 it should be noted that some are limited in the number of significant figures they return. This may affect the statistical properties of the resulting sequence of variates. Section 7 of the individual function documentation will give a discussion of the accuracy of the particular algorithm being used and any available alternatives.

3.2 Quasi-random Numbers

Prior to generating any quasi-random variates the generator being used must be initialized via nag_quasi_init (g05ylc) or nag_quasi_init_scrambled (g05ync). Of these, nag_quasi_init (g05ylc) can be used to initialize a standard Sobol, Faure or Niederreiter sequence and nag_quasi_init_scrambled (g05ync) can be used to initialize a scrambled Sobol or Niederreiter sequence.

Owing to the random nature of the scrambling, before calling the initialization function nag_quasi_init_scrambled (g05ync) one of the pseudorandom initialization functions, nag_rand_ini t_repeatable (g05kfc) or nag_rand_init_nonrepeatable (g05kgc), must be called.

Once a quasi-random generator has been initialized, using either nag_quasi_init (g05ylc) or nag_quasi_init_scrambled (g05ync), one of three generation functions can be called to generate uniformly distributed sequences (nag_quasi_rand_uniform (g05ymc)), Normally distributed sequences (nag_quasi_rand_normal (g05yjc)) or sequences with a log-normal distribution (nag_quasi_rand_lognormal (g05ykc)). For example, for a repeatable sequence of scrambled quasi-random variates from the Normal distribution, nag_rand_init_repeatable (g05kfc) must be called first (to initialize a pseudorandom generator), followed by nag_quasi_init_scrambled (g05ync) (to initialize a scrambled quasi-random generator) and then nag_quasi_rand_normal (g05yjc) can be called to generate the sequence from the required distribution.

See the last paragraph of Section 3.1.5 on how sequences from other distributions can be obtained using the inverse CDF.

3.3 Brownian Bridge

nag_rand_bb (g05xbc) may be used to generate sample paths from a (free or non-free) Wiener process using the Brownian bridge algorithm. Prior to calling nag_rand_bb (g05xbc), the generator must be initialized by a call to nag_rand_bb_init (g05xac). nag_rand_bb_init (g05xac) requires you to specify a *bridge construction order*. The function nag_rand_bb_make_bridge_order (g05xec) can be used to convert a set of input times into one of several common bridge construction orders, which can then be used in the initialization call to nag_rand_bb_init (g05xac).

nag_rand_bb_inc (g05xdc) may be used to generate the *scaled increments* of the sample paths of a (free or non-free) Wiener process. Prior to calling nag_rand_bb_inc (g05xdc), the generator must be initialized by a call to nag_rand_bb_inc_init (g05xcc). Note that nag_rand_bb_inc (g05xdc) generates these scaled increments directly; it is not necessary to call nag_rand_bb (g05xbc) before calling nag_rand_bb_inc (g05xdc). As before, nag_rand_bb_make_bridge_order (g05xec) can be used to convert a set of input times into a bridge construction order which can be passed to nag_rand_bb_inc_init (g05xcc).

3.4 Random Fields

Functions for simulating from either a one-dimensional or a two-dimensional stationary Gaussian random field are provided. These functions use the *circulant embedding method* of Dietrich and Newsam (1997) to efficiently generate from the required field. In both cases a setup function is called, which defines the domain and variogram to use, followed by the generation function. A number of preset variograms are supplied or a user-defined function can be used.

One-dimensional random field:

nag_rand_field_1d_predef_setup (g05znc) setup function, using a preset variogram.

nag_rand_field_1d_user_setup (g05zmc) setup function, using a user-defined variogram.

nag_rand_field_1d_generate (g05zpc) generation function.

Two-dimension random field:

nag_rand_field_2d_user_setup (g05zqc) setup function, using a preset variogram.

nag_rand_field_2d_predef_setup (g05zrc) setup function, using a user-defined variogram.

nag_rand_field_2d_generate (g05zsc) generation function.

In addition to generating a random field, it is possible to use the circulant embedding method to generate realizations of fractional Brownian motion, this functionality is provided in nag_rand_field_fracbm_generate (g05ztc).

Before calling nag_rand_field_1d_generate (g05zpc), nag_rand_field_2d_predef_setup (g05zrc) or nag_rand_field_fracbm_generate (g05ztc) one of the initialization functions, nag_rand_init_repeatable (g05kfc) or nag_rand_init_nonrepeatable (g05kgc) must be called.

3.5 Sampling

Each of the four sampling methods described in Section 2.8 can be performed using the following functions:

nag_rand_discrete_uniform (g05tlc) Sampling with replacement, equal weights.

nag_rand_gen_discrete (g05tdc) Sampling with replacement, unequal weights.

nag_rand_sample (g05ndc) Sampling without replacement, equal weights.

nag_rand_sample_unequal (g05nec) Sampling without replacement, unequal weights.

In addition to these functions for directly sampling from a dataset two utility functions that perform an in-place permutation to give datasets suitable for use in validation are provided. nag_rand_kfold_xyw (g05pvc) generates training and validation datasets suitable for K-fold cross-validation and nag_rand_subsamp_xyw (g05pwc) generates training and validation datasets suitable for random sub-sampling validation. To perform stratified sampling the dataset should first be ordered by stratum using a sorting function from Chapter m01 and then one of the above sampling functions can be applied to each stratum.

4 Functionality Index

- Brownian bridge,
  - circulant embedding generator,
    - generate fractional Brownian motion .... nag_rand_field_fracbm_generate (g05ztc)
  - increments generator,
    - generate Wiener increments .... nag_rand_bb_inc (g05xdc)
    - initialize generator .... nag_rand_bb_inc_init (g05xcc)
  - path generator,
    - create bridge construction order .... nag_rand_bb_make_bridge_order (g05xec)
    - generate a free or non-free (pinned) Wiener process for a given set of time steps ..... nag_rand_bb (g05xbc)
    - initialize generator .... nag_rand_bb_init (g05xac)

- Generating samples, matrices and tables,
  - permutation of real matrix, vector, vector triplet
    - $K-$fold cross-validation .... nag_rand_kfold_xyw (g05pvc)
    - random sub-sampling validation .... nag_rand_subsamp_xyw (g05pwc)
  - random correlation matrix .... nag_rand_corr_matrix (g05pyc)
  - random orthogonal matrix .... nag_rand_orthog_matrix (g05pxc)
  - random permutation of an integer vector .... nag_rand_permute (g05ncc)
  - random sample from an integer vector,
    - unequal weights, without replacement .... nag_rand_sample_unequal (g05nec)
    - unweighted, without replacement .... nag_rand_sample (g05ndc)
  - random table .... nag_rand_2_way_table (g05pzc)

- Generation of time series,
  - asymmetric GARCH Type II .... nag_rand_agarchII (g05pec)
  - asymmetric GJR GARCH .... nag_rand_garchGJR (g05pfc)
  - EGARCH .... nag_rand_egarch (g05pgc)
  - exponential smoothing .... nag_rand_exp_smooth (g05pmc)
  - type I AGARCH .... nag_rand_agarchI (g05pdc)
  - univariate ARMA .... nag_rand_arma (g05phc)
  - vector ARMA .... nag_rand_varma (g05pjc)

- Pseudorandom numbers,
  - array of variates from multivariate distributions,
    - Dirichlet distribution .... nag_rand_dirichlet (g05sec)
    - multinomial distribution .... nag_rand_gen_multinomial (g05tgc)
    - Normal distribution .... nag_rand_matrix_multi_normal (g05rzc)
    - Student's t distribution .... nag_rand_matrix_multi_students_t (g05ryc)

- copulas,
  - Clayton/Cook–Johnson copula (bivariate) .... nag_rand_bivariate_copula_clayton (g05rec)
  - Clayton/Cook–Johnson copula (multivariate) .... nag_rand_copula_clayton (g05rhc)
  - Frank copula (bivariate) .... nag_rand_bivariate_copula_frank (g05rfc)
  - Frank copula (multivariate) .... nag_rand_copula_frank (g05rjc)
  - Gaussian copula .... nag_rand_copula_normal (g05rdc)
  - Gumbel–Hougaard copula .... nag_rand_copula_gumbel (g05rkc)
  - Plackett copula .... nag_rand_bivariate_copula_plackett (g05rgc)
  - Student's t copula .... nag_rand_copula_students_t (g05rcc)
- initialize generator,
  - multiple streams,
    - leap-frog .... nag_rand_leap_frog (g05khc)
    - skip-ahead .... nag_rand_skip_ahead (g05kjc)
    - skip-ahead (power of 2) .... nag_rand_skip_ahead_power2 (g05kkc)
  - nonrepeatable sequence .... nag_rand_init_nonrepeatable (g05kgc)
  - repeatable sequence .... nag_rand_init_repeatable (g05kfc)
- vector of variates from discrete univariate distributions,
  - binomial distribution .... nag_rand_binomial (g05tac)
  - geometric distribution .... nag_rand_geom (g05tcc)
  - hypergeometric distribution .... nag_rand_hypergeometric (g05tec)
  - logarithmic distribution .... nag_rand_logarithmic (g05tfc)
  - logical value Nag_TRUE or Nag_FALSE .... nag_rand_logical (g05tbc)
  - negative binomial distribution .... nag_rand_neg_bin (g05thc)
  - Poisson distribution .... nag_rand_poisson (g05tjc)
  - uniform distribution .... nag_rand_discrete_uniform (g05tlc)
  - user-supplied distribution .... nag_rand_gen_discrete (g05tdc)
  - variate array from discrete distributions with array of parameters,
    - Poisson distribution with varying mean .... nag_rand_compd_poisson (g05tkc)
- vectors of variates from continuous univariate distributions,
  - beta distribution .... nag_rand_beta (g05sbc)
  - Cauchy distribution .... nag_rand_cauchy (g05scc)
  - exponential mix distribution .... nag_rand_exp_mix (g05sgc)
  - F-distribution .... nag_rand_f (g05shc)
  - gamma distribution .... nag_rand_gamma (g05sjc)
  - logistic distribution .... nag_rand_logistic (g05slc)
  - log-normal distribution .... nag_rand_lognormal (g05smc)
  - negative exponential distribution .... nag_rand_exp (g05sfc)
  - Normal distribution .... nag_rand_normal (g05skc)
  - real number from the continuous uniform distribution .... nag_rand_basic (g05sac)
  - Student's t-distribution .... nag_rand_students_t (g05snc)
  - triangular distribution .... nag_rand_triangular (g05spc)
  - uniform distribution .... nag_rand_uniform (g05sqc)
  - von Mises distribution .... nag_rand_von_mises (g05src)
  - Weibull distribution .... nag_rand_weibull (g05ssc)
  - χ^2 square distribution .... nag_rand_chi_sq (g05sdc)

Quasi-random numbers,

- array of variates from univariate distributions,
  - log-normal distribution .... nag_quasi_rand_lognormal (g05ykc)
  - Normal distribution .... nag_quasi_rand_normal (g05yjc)
  - uniform distribution .... nag_quasi_rand_uniform (g05ymc)
- initialize generator,
  - scrambled Sobol or Niederreiter .... nag_quasi_init_scrambled (g05ync)
  - Sobol, Niederreiter or Faure .... nag_quasi_init (g05ylc)

Random fields,

- one-dimensional,
  - generation .... nag_rand_field_1d_generate (g05zpc)

initialize generator,
preset variogram ... nag_rand_field_1d_predef_setup (g05znc)
user-defined variogram ... nag_rand_field_1d_user_setup (g05zmc)

two-dimensional,
generation ... nag_rand_field_2d_generate (g05zsc)
initialize generator,
preset variogram ... nag_rand_field_2d_predef_setup (g05zrc)
user-defined variogram ... nag_rand_field_2d_user_setup (g05zqc)

5 Auxiliary Functions Associated with Library Function Arguments

None.

6 Functions Withdrawn or Scheduled for Withdrawal

The following lists all those functions that have been withdrawn since Mark 23 of the Library or are scheduled for withdrawal at one of the next two marks.

Withdrawn	Mark of	
Function	**Withdrawal**	**Replacement Function(s)**
nag_random_continuous_uniform (g05cac)	24	nag_rand_basic (g05sac)
nag_random_init_repeatable (g05cbc)	24	nag_rand_init_repeatable (g05kfc)
nag_random_init_nonrepeatable (g05ccc)	24	nag_rand_init_nonrepeatable (g05kgc)
nag_save_random_state (g05cfc)	24	No longer required.
nag_restore_random_state (g05cgc)	24	No longer required.
nag_random_continuous_uniform_ab (g05dac)	24	nag_rand_uniform (g05sqc)
nag_random_exp (g05dbc)	24	nag_rand_exp (g05sfc)
nag_random_normal (g05ddc)	24	nag_rand_normal (g05skc)
nag_random_discrete_uniform (g05dyc)	24	nag_rand_discrete_uniform (g05tlc)
nag_ref_vec_multi_normal (g05eac)	24	nag_rand_matrix_multi_normal (g05rzc)
nag_ref_vec_poisson (g05ecc)	24	nag_rand_poisson (g05tjc)
nag_ref_vec_binomial (g05edc)	24	nag_rand_binomial (g05tac)
nag_ran_permut_vec (g05ehc)	24	nag_rand_permute (g05ncc)
nag_ran_sample_vec (g05ejc)	24	nag_rand_sample (g05ndc)
nag_ref_vec_discrete_pdf_cdf (g05exc)	24	nag_rand_gen_discrete (g05tdc)
nag_return_discrete (g05eyc)	24	nag_rand_gen_discrete (g05tdc)
nag_return_multi_normal (g05ezc)	24	nag_rand_matrix_multi_normal (g05rzc)
nag_random_beta (g05fec)	24	nag_rand_beta (g05sbc)
nag_random_gamma (g05ffc)	24	nag_rand_gamma (g05sjc)
nag_arma_time_series (g05hac)	24	nag_rand_arma (g05phc)
nag_generate_agarchI (g05hkc)	24	nag_rand_agarchI (g05pdc)
nag_generate_agarchII (g05hlc)	24	nag_rand_agarchII (g05pec)
nag_generate_garchGJR (g05hmc)	24	nag_rand_garchGJR (g05pfc)
nag_rngs_basic (g05kac)	24	nag_rand_basic (g05sac)
nag_rngs_init_repeatable (g05kbc)	24	nag_rand_init_repeatable (g05kfc)
nag_rngs_init_nonrepeatable (g05kcc)	24	nag_rand_init_nonrepeatable (g05kgc)
nag_rngs_logical (g05kec)	24	nag_rand_logical (g05tbc)
nag_rngs_normal (g05lac)	24	nag_rand_normal (g05skc)
nag_rngs_students_t (g05lbc)	24	nag_rand_students_t (g05snc)
nag_rngs_chi_sq (g05lcc)	24	nag_rand_chi_sq (g05sdc)
nag_rngs_f (g05ldc)	24	nag_rand_f (g05shc)
nag_rngs_beta (g05lec)	24	nag_rand_beta (g05sbc)
nag_rngs_gamma (g05lfc)	24	nag_rand_gamma (g05sjc)
nag_rngs_uniform (g05lgc)	24	nag_rand_uniform (g05sqc)
nag_rngs_triangular (g05lhc)	24	nag_rand_triangular (g05spc)
nag_rngs_exp (g05ljc)	24	nag_rand_exp (g05sfc)

nag_rngs_lognormal (g05lkc)	24	nag_rand_lognormal (g05smc)
nag_rngs_cauchy (g05llc)	24	nag_rand_cauchy (g05scc)
nag_rngs_weibull (g05lmc)	24	nag_rand_weibull (g05ssc)
nag_rngs_logistic (g05lnc)	24	nag_rand_logistic (g05slc)
nag_rngs_von_mises (g05lpc)	24	nag_rand_von_mises (g05src)
nag_rngs_exp_mix (g05lqc)	24	nag_rand_exp_mix (g05sgc)
nag_rngs_matrix_multi_students_t (g05lxc)	24	nag_rand_matrix_multi_students_t (g05ryc)
nag_rgsn_matrix_multi_normal (g05lyc)	24	nag_rand_matrix_multi_normal (g05rzc)
nag_rngs_multi_normal (g05lzc)	24	nag_rand_matrix_multi_normal (g05rzc)
nag_rngs_discrete_uniform (g05mac)	24	nag_rand_discrete_uniform (g05tlc)
nag_rngs_geom (g05mbc)	24	nag_rand_geom (g05tcc)
nag_rngs_neg_bin (g05mcc)	24	nag_rand_neg_bin (g05thc)
nag_rngs_logarithmic (g05mdc)	24	nag_rand_logarithmic (g05tfc)
nag_rngs_compd_poisson (g05mec)	24	nag_rand_compd_poisson (g05tkc)
nag_rngs_binomial (g05mjc)	24	nag_rand_binomial (g05tac)
nag_rngs_poisson (g05mkc)	24	nag_rand_poisson (g05tjc)
nag_rngs_hypergeometric (g05mlc)	24	nag_rand_hypergeometric (g05tec)
nag_rngs_gen_multinomial (g05mrc)	24	nag_rand_gen_multinomial (g05tgc)
nag_rngs_gen_discrete (g05mzc)	24	nag_rand_gen_discrete (g05tdc)
nag_rngs_permute (g05nac)	24	nag_rand_permute (g05ncc)
nag_rngs_sample (g05nbc)	24	nag_rand_sample (g05ndc)
nag_rngs_arma_time_series (g05pac)	24	nag_rand_arma (g05phc)
nag_rngs_varma_time_series (g05pcc)	24	nag_rand_varma (g05pjc)
nag_rngs_orthog_matrix (g05qac)	24	nag_rand_orthog_matrix (g05pxc)
nag_rngs_corr_matrix (g05qbc)	24	nag_rand_corr_matrix (g05pyc)
nag_rngs_2_way_table (g05qdc)	24	nag_rand_2_way_table (g05pzc)
nag_rngs_copula_normal (g05rac)	24	nag_rand_copula_normal (g05rdc)
nag_rngs_copula_students_t (g05rbc)	24	nag_rand_copula_students_t (g05rcc)
nag_quasi_random_uniform (g05yac)	24	nag_quasi_init (g05ylc) and nag_quasi_rand_uniform (g05ymc)
nag_quasi_random_normal (g05ybc)	24	nag_quasi_rand_normal (g05yjc) and nag_quasi_init (g05ylc)

7 References

Banks J (1998) *Handbook on Simulation* Wiley

Boye E (Unpublished manuscript) Copulas for finance: a reading guide and some applications Financial Econometrics Research Centre, City University Business School, London

Bratley P and Fox B L (1988) Algorithm 659: implementing Sobol's quasirandom sequence generator *ACM Trans. Math. Software* **14(1)** 88–100

Dietrich C R and Newsam G N (1997) Fast and exact simulation of stationary Gaussian processes through circulant embedding of the covariance matrix *SIAM J. Sci. Comput.* **18** 1088–1107

Faure H and Tezuka S (2000) Another random scrambling of digital (t,s)-sequences *Monte Carlo and Quasi-Monte Carlo Methods* Springer-Verlag, Berlin, Germany (eds K T Fang, F J Hickernell and H Niederreiter)

Fox B L (1986) Algorithm 647: implementation and relative efficiency of quasirandom sequence generators *ACM Trans. Math. Software* **12(4)** 362–376

Glasserman P (2004) *Monte Carlo Methods in Financial Engineering* Springer

Haramoto H, Matsumoto M, Nishimura T, Panneton F and L'Ecuyer P (2008) Efficient jump ahead for F2-linear random number generators *INFORMS J. on Computing* **20(3)** 385–390

Hong H S and Hickernell F J (2003) Algorithm 823: implementing scrambled digital sequences *ACM Trans. Math. Software* **29:2** 95–109

Joe S and Kuo F Y (2008) Constructing Sobol sequences with better two-dimensional projections *SIAM J. Sci. Comput.* **30** 2635–2654

Knuth D E (1981) *The Art of Computer Programming (Volume 2)* (2nd Edition) Addison–Wesley

L'Ecuyer P and Simard R (2002) *TestU01: a software library in ANSI C for empirical testing of random number generators* Departement d'Informatique et de Recherche Operationnelle, Universite de Montreal http://www.iro.umontreal.ca/~lecuyer

Maclaren N M (1989) The generation of multiple independent sequences of pseudorandom numbers *Appl. Statist.* **38** 351–359

Matsumoto M and Nishimura T (1998) Mersenne twister: a 623-dimensionally equidistributed uniform pseudorandom number generator *ACM Transactions on Modelling and Computer Simulations*

Morgan B J T (1984) *Elements of Simulation* Chapman and Hall

Nelsen R B (1998) *An Introduction to Copulas. Lecture Notes in Statistics 139* Springer

Owen A B (1995) Randomly permuted (t,m,s)-nets and (t,s)-sequences *Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing, Lecture Notes in Statistics* **106** Springer-Verlag, New York, NY 299–317 (eds H Niederreiter and P J-S Shiue)

Revuz D and Yor M (1999) *Continuous Martingales and Brownian Motion* Springer

Ripley B D (1987) *Stochastic Simulation* Wiley

Sklar A (1973) Random variables: joint distribution functions and copulas *Kybernetika* **9** 499–460

Wichmann B A and Hill I D (2006) Generating good pseudo-random numbers *Computational Statistics and Data Analysis* **51** 1614–1622

Wikramaratna R S (1989) ACORN - a new method for generating sequences of uniformly distributed pseudo-random numbers *Journal of Computational Physics* **83** 16–31

Wikramaratna R S (1992) Theoretical background for the ACORN random number generator *Report AEA-APS-0244* AEA Technology, Winfrith, Dorest, UK

Wikramaratna R S (2007) The additive congruential random number generator a special case of a multiple recursive generator *Journal of Computational and Applied Mathematics*