

NAG Library Chapter Introduction

g05 – Random Number Generators

Contents

1	Scope of the Chapter	2
2	Background to the Problems	2
2.1	Pseudorandom Numbers	2
2.1.1	NAG Basic Generator	2
2.1.2	Wichmann–Hill I Generator	3
2.1.3	Wichmann–Hill II Generator	3
2.1.4	Mersenne Twister Generator	3
2.1.5	ACORN Generator	4
2.2	Quasi-random Numbers	5
2.3	Scrambled Quasi-random Numbers	5
2.4	Non-uniform Random Numbers	6
2.5	Copulas	6
2.6	Other Random Structures	6
2.7	Multiple Streams of Pseudorandom Numbers	7
2.7.1	Multiple Streams via Different Initial Values (Seeds)	7
2.7.2	Multiple Streams via Different Generators	7
2.7.3	Multiple Streams via Skip-ahead	7
2.7.4	Multiple Streams via Leap-frog	8
2.7.5	Skip-ahead and Leap-frog: An Example	8
3	Recommendations on Choice and Use of Available Functions	9
3.1	Pseudorandom Numbers	9
3.1.1	Initialization	9
3.1.2	Repeated initialization	9
3.1.3	Choice of Base Generator	9
3.1.4	Choice of Method for Generating Multiple Streams	10
3.1.5	Copulas	10
3.2	Quasi-random Numbers	10
3.3	Programming Advice	10
4	Index	11
5	Functions Withdrawn or Scheduled for Withdrawal	12
6	References	13

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 the generation of pseudorandom time series from specified time series 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 routine 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 five base generators currently available in the NAG C Library, these are; a basic linear congruential generator (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)) and the ACORN generator (see Wikramaratna (1989)).

2.1.1 NAG Basic Generator

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

$$x_i = a_1 x_{i-1} \bmod m_1,$$

$$u_i = \frac{x_i}{m_1},$$

where the u_i , for $i = 1, 2, \dots$, 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 Library since Mark 6 and as such has been widely used. It suffers from no known problems, other than those due to the lattice structure inherent in all LCGs, 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 (LCGs) and has the form:

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

where the u_i , for $i = 1, 2, \dots$, 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 LCGs. 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

$$w_i = 11600 w_{i-1} \bmod 2147483579$$

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, \dots, 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

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

where $x_i^{(w:(l+1))} | x_{i+1}^{(l+1)}$ 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:

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

where t_1, t_2, t_3 and t_4 are integers and m_1 and m_2 are bit-masks and ‘ $\ggg t$ ’ and ‘ $\lll t$ ’ 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 $i = 1, 2, \dots$, form the required sequence. The supplied implementation of the Mersenne Twister uses the following values for the algorithmic constants:

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

where the notation $0xDD\dots$ indicates the bit pattern of the integer whose hexadecimal representation is $DD\dots$

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, $Y_0^{(1)}, Y_0^{(2)}, \dots, Y_0^{(k)}$, such that $0 \leq Y_0^{(m)} < M$, for all $m = 1, 2, \dots, k$.

(iv) Perform the following sequentially:

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

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

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

The u_i , for $i = 1, 2, \dots$, 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.

2.2 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, \dots, 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 $x^1, x^2, \dots, x^N \in I^S$ 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 $x = (x_1, x_2, \dots, x_S) \in I^S$, let G_x be the rectangular s -dimensional region

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

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

$$D_N^*(x^1, x^2, \dots, x^N) = \sup_{x \in I^S} |S_N(G_x) - Nx_1, x_2, \dots, x_S|.$$

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

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

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 (2003) 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).

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.

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.

2.4 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, \dots, f_m , then f has a unique copula representation, c , such that

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

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

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

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 Other Random Structures

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

2.7 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 routines available in this chapter:

- (i) using different initial values (seeds);
- (ii) using different generators;
- (iii) skip ahead (also called block-splitting);
- (iv) leap-frogging.

2.7.1 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 2.7.3, 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.

Using different initial values is the only way of generating multiple streams using this implementation of the Mersenne Twister as the base generator. Although the extremely large period of this generator reduces the chance of the different sequences overlapping, especially if the number of sequences required is small, compared to using a generator that has a smaller period, it is not recommended that the Mersenne Twister be used if multiple streams are required.

2.7.2 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 and sequence 3 the ACORN 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, 277 independent streams can be generated with each using a different generator (273 Wichmann–Hill I generators, and 4 additional base generators).

2.7.3 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,

$$\frac{x_1, x_2, \dots, x_n}{\text{block 1}}, \frac{x_{n+1}, x_{n+2}, \dots, x_{2n}}{\text{block 2}}, \frac{x_{2n+1}, x_{2n+2}, \dots, x_{3n}}{\text{block 3}}, \text{ etc.}$$

where x_1, x_2, \dots 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. Due to their form this can be done efficiently for linear congruential generators and multiple congruential generators. A skip-ahead algorithm is provided for the NAG Basic generator, both the Wichmann–Hill I and Wichmann–Hill II generators and the ACORN 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).

2.7.4 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:

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

where x_1, x_2, \dots 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 k th variate from the base generator. Due 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 and both the Wichmann–Hill I and Wichmann–Hill II generators.

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.

2.7.5 Skip-ahead and Leap-frog: 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 (LCG) is given. A linear congruential generator has the form $x_{i+1} = a_1 x_i \bmod m_1$. The recursive nature of a LCG 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 LCG 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 emphasised by the lattice structure of LCGs. Similarly, 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.

3 Recommendations on Choice and Use of Available Functions

3.1 Pseudorandom Numbers

Prior to 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 routine (`nag_rand_basic` (`g05sac`)) should be called.

3.1.1 Initialization

Prior to generating any variates the base generator must be initialized. Two utility routines 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 2.7.1, 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` = 3). 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.

If multiple sequences are required, then the Wichmann–Hill II generator is a good choice, especially if a period of 2^{90} is sufficient, in which case the method described in Section 2.7.1 can be used. This generator has also been shown to perform well on various test suites (see Wichmann and Hill (2006)).

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 2.7.1 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 2.7.4, using leap-frogging can, in some instances, change the statistical properties of the sequences being generated.

3.1.5 Copulas

After calling `nag_rand_copula_students_t` (g05rcc) or `nag_rand_copula_normal` (g05rdc) the g01f functions in Chapter g01 can be used to convert the uniform marginal distributors into a different form as required.

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.

Due to the random nature of the scrambling, prior to calling the initialization routine `nag_quasi_init_scrambled` (g05ync) one of the pseudorandom initialization routines, `nag_rand_init_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 routines 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.

Sequences from other distributions can be obtained by calling the ‘deviate’ routines supplied in Chapter g01 on the results from `nag_quasi_rand_uniform` (g05ymc). However, care should be taken when doing this as some of these ‘deviate’ routines are only accurate up to a limited number of significant figures which may effect the statistical properties of the resulting sequence of variates.

3.3 Programming Advice

Take care when programming calls to those functions in this chapter which are functions. The reason is that different calls with the same arguments are intended to give different results.

For example, if you wish to assign to `z` the difference between two successive random numbers generated by `nag_rngs_basic` (g05kac), beware of writing

```
z = g05kac(igen,iseed) - g05kac(igen,iseed)
```

It is quite legitimate for a C compiler to compile zero, one or two calls to `nag_rngs_basic` (`g05kac`); if two calls, they may be in either order (if zero or one calls are compiled, `z` would be set to zero). A safe method to program this would be

```
x = g05kac(igen,iseed);
y = g05kac(igen,iseed);
z = x-y
```

4 Index

Generating samples, matrices and tables,

random correlation matrix	<code>nag_rand_corr_matrix</code> (<code>g05pyc</code>)
random orthogonal matrix	<code>nag_rand_orthog_matrix</code> (<code>g05pxc</code>)
random permutation of an integer vector	<code>nag_rand_permute</code> (<code>g05ncc</code>)
random sample from an integer vector	<code>nag_rand_sample</code> (<code>g05ndc</code>)
random table	<code>nag_rand_2_way_table</code> (<code>g05pzc</code>)

Generation of time series,

asymmetric GARCH Type II	<code>nag_rand_agarchII</code> (<code>g05pec</code>)
asymmetric GJR GARCH	<code>nag_rand_garchGJR</code> (<code>g05pfc</code>)
EGARCH	<code>nag_rand_egarch</code> (<code>g05pgc</code>)
exponential smoothing	<code>nag_rand_exp_smooth</code> (<code>g05pmc</code>)
type I AGARCH	<code>nag_rand_agarchI</code> (<code>g05pdc</code>)
univariate ARMA	<code>nag_rand_arma</code> (<code>g05phc</code>)
vector ARMA	<code>nag_rand_varma</code> (<code>g05pjc</code>)

Pseudorandom numbers,

array of variates from multivariate distributions,

Dirichlet distribution	<code>nag_rand_dirichlet</code> (<code>g05sec</code>)
multinomial distribution	<code>nag_rand_gen_multinomial</code> (<code>g05tgc</code>)
Normal distribution	<code>nag_rand_matrix_multi_students_t</code> (<code>g05ryc</code>)
Student's <i>t</i> distribution	<code>nag_rand_matrix_multi_normal</code> (<code>g05rzc</code>)

copulas

Clayton/Cook–Johnson copula	<code>nag_rand_bivariate_copula_clayton</code> (<code>g05rec</code>)
Frank copula	<code>nag_rand_bivariate_copula_frank</code> (<code>g05rhc</code>)
Gaussian copula	<code>nag_rand_copula_normal</code> (<code>g05rdc</code>)
Gumbel–Hougaard copula	<code>nag_rand_copula_gumbel</code> (<code>g05rkc</code>)
Plackett copula	<code>nag_rand_bivariate_copula_plackett</code> (<code>g05rgc</code>)
Student's <i>t</i> copula	<code>nag_rand_copula_students_t</code> (<code>g05rcc</code>)

initialize generator,

multiple streams,

leap-frog	<code>nag_rand_skip_ahead</code> (<code>g05kjc</code>)
skip-ahead	<code>nag_rand_leap_frog</code> (<code>g05khc</code>)
nonrepeatable sequence	<code>nag_rand_init_nonrepeatable</code> (<code>g05kgc</code>)
repeatable sequence	<code>nag_rand_init_repeatable</code> (<code>g05kfc</code>)

vector of variates from discrete univariate distributions,

binomial distribution	<code>nag_rand_binomial</code> (<code>g05tac</code>)
geometric distribution	<code>nag_rand_geom</code> (<code>g05tcc</code>)
hypergeometric distribution	<code>nag_rand_hypergeometric</code> (<code>g05tec</code>)
logarithmic distribution	<code>nag_rand_logarithmic</code> (<code>g05tfc</code>)
logical value <code>Nag_TRUE</code> or <code>Nag_FALSE</code>	<code>nag_rand_logical</code> (<code>g05tbc</code>)
negative binomial distribution	<code>nag_rand_neg_bin</code> (<code>g05thc</code>)
Poisson distribution	<code>nag_rand_poisson</code> (<code>g05tjc</code>)
uniform distribution	<code>nag_rand_discrete_uniform</code> (<code>g05tlc</code>)
user-supplied distribution	<code>nag_rand_gen_discrete</code> (<code>g05tdc</code>)

variate array from discrete distributions with array of parameters,

Poisson distribution with varying mean	<code>nag_rand_compound_poisson</code> (<code>g05tkc</code>)
--	--

vectors of variates from continuous univariate distributions,

beta distribution	<code>nag_rand_beta</code> (<code>g05sbc</code>)
-------------------------	--

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)

5 Functions Withdrawn or Scheduled for Withdrawal

Withdrawn Function	Mark of Withdrawal	Replacement Function(s)
nag_random_continuous_uniform (g05cac)	11	nag_rand_basic (g05sac)
nag_random_init_repeatable (g05cbc)	11	nag_rand_init_repeatable (g05kfc)
nag_random_init_nonrepeatable (g05ccc)	11	nag_rand_init_nonrepeatable (g05kgc)
nag_save_random_state (g05cfc)	11	No replacement routine required
nag_restore_random_state (g05cgc)	11	No replacement routine required
nag_random_continuous_uniform_ab (g05dac)	11	nag_rand_uniform (g05sqc)
nag_random_exp (g05dbc)	11	nag_rand_exp (g05sfc)
nag_random_normal (g05ddc)	11	nag_rand_normal (g05skc)
nag_random_discrete_uniform (g05dyc)	11	nag_rand_discrete_uniform (g05tlc)
nag_ref_vec_multi_normal (g05eac)	11	nag_rand_matrix_multi_normal (g05zrc)
nag_ref_vec_poisson (g05ecc)	11	nag_rand_poisson (g05tjc)
nag_ref_vec_binomial (g05edc)	11	nag_rand_binomial (g05tac)
nag_ran_permut_vec (g05ehc)	11	nag_rand_permute (g05ncc)
nag_ran_sample_vec (g05ejc)	11	nag_rand_sample (g05ndc)
nag_ref_vec_discrete_pdf_cdf (g05exc)	11	nag_rand_gen_discrete (g05tdc)
nag_return_discrete (g05eyc)	11	nag_rand_gen_discrete (g05tdc)
nag_return_multi_normal (g05ezc)	11	nag_rand_matrix_multi_normal (g05zrc)
nag_random_beta (g05fec)	11	nag_rand_beta (g05sbc)
nag_random_gamma (g05ffc)	11	nag_rand_gamma (g05sjc)
nag_arma_time_series (g05hac)	11	nag_rand_arma (g05phc)
nag_generate_agarchI (g05hkc)	11	nag_rand_agarchI (g05pdc)
nag_generate_agarchII (g05hlc)	11	nag_rand_agarchII (g05pec)
nag_generate_garchGJR (g05hmc)	11	nag_rand_garchGJR (g05pfc)
nag_rngs_basic (g05kac)	11	nag_rand_basic (g05sac)
nag_rngs_init_repeatable (g05kbc)	11	nag_rand_init_repeatable (g05kfc)
nag_rngs_init_nonrepeatable (g05kcc)	11	nag_rand_init_nonrepeatable (g05kgc)
nag_rngs_logical (g05kec)	11	nag_rand_logical (g05tbc)
nag_rngs_normal (g05lac)	11	nag_rand_normal (g05skc)

nag_rngs_students_t (g05lbc)	11	nag_rand_students_t (g05snc)
nag_rngs_chi_sq (g05lcc)	11	nag_rand_chi_sq (g05sdc)
nag_rngs_f (g05ldc)	11	nag_rand_f (g05shc)
nag_rngs_beta (g05lec)	11	nag_rand_beta (g05sbc)
nag_rngs_gamma (g05lfc)	11	nag_rand_gamma (g05sjc)
nag_rngs_uniform (g05lgc)	11	nag_rand_uniform (g05sqc)
nag_rngs_triangular (g05lhc)	11	nag_rand_triangular (g05spc)
nag_rngs_exp (g05lje)	11	nag_rand_exp (g05sfc)
nag_rngs_lognormal (g05lkc)	11	nag_rand_lognormal (g05smc)
nag_rngs_cauchy (g05llc)	11	nag_rand_cauchy (g05scc)
nag_rngs_weibull (g05lmc)	11	nag_rand_weibull (g05ssc)
nag_rngs_logistic (g05lnc)	11	nag_rand_logistic (g05slc)
nag_rngs_von_mises (g05lpc)	11	nag_rand_von_mises (g05src)
nag_rngs_exp_mix (g05lqc)	11	nag_rand_exp_mix (g05sgc)
nag_rngs_matrix_multi_students_t (g05lxc)	11	nag_rand_matrix_multi_students_t (g05ryc)
nag_rgsn_matrix_multi_normal (g05lyc)	11	nag_rand_matrix_multi_normal (g05rzc)
nag_rngs_multi_normal (g05lzc)	11	nag_rand_matrix_multi_normal (g05rzc)
nag_rngs_discrete_uniform (g05mac)	11	nag_rand_discrete_uniform (g05tlc)
nag_rngs_geom (g05mbc)	11	nag_rand_geom (g05tcc)
nag_rngs_neg_bin (g05mcc)	11	nag_rand_neg_bin (g05thc)
nag_rngs_logarithmic (g05mdc)	11	nag_rand_logarithmic (g05tfc)
nag_rngs_compnd_poisson (g05mec)	11	nag_rand_compnd_poisson (g05tkc)
nag_rngs_binomial (g05mjc)	11	nag_rand_binomial (g05tac)
nag_rngs_poisson (g05mkc)	11	nag_rand_poisson (g05tjc)
nag_rngs_hypergeometric (g05mlc)	11	nag_rand_hypergeometric (g05tec)
nag_rngs_gen_multinomial (g05mrc)	11	nag_rand_gen_multinomial (g05tgc)
nag_rngs_gen_discrete (g05mzc)	11	nag_rand_gen_discrete (g05tdc)
nag_rngs_permute (g05nac)	11	nag_rand_permute (g05ncc)
nag_rngs_sample (g05nbc)	11	nag_rand_sample (g05ndc)
nag_rngs_arma_time_series (g05pac)	11	nag_rand_arma (g05phc)
nag_rngs_varma_time_series (g05pcc)	11	nag_rand_varma (g05pjc)
nag_rngs_orthog_matrix (g05qac)	11	nag_rand_orthog_matrix (g05pxc)
nag_rngs_corr_matrix (g05qbc)	11	nag_rand_corr_matrix (g05pyc)
nag_rngs_2_way_table (g05qdc)	11	nag_rand_2_way_table (g05pzc)
nag_rngs_copula_normal (g05rac)	11	nag_rand_copula_normal (g05rdc)
nag_rngs_copula_students_t (g05rbc)	11	nag_rand_copula_students_t (g05rcc)
nag_quasi_random_uniform (g05yac)	11	nag_quasi_init (g05ylc) and nag_quasi_rand_uniform (g05ymc)
nag_quasi_random_normal (g05ybc)	11	nag_quasi_rand_normal (g05yjc) and nag_quasi_init (g05ylc)

6 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

Brately P and Fox B L (1988) Algorithm 659: Implementing Sobol's Quasirandom Sequence Generator *ACM Trans. Math. Software* **14** (1) 88–100

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

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 (2003) Remark on Algorithm 659: Implementing Sobol's quasirandom sequence generator *ACM Trans. Math. Software* **29** 49–57
- 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)
- 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*
-