Chapter G05
Random Number Generators

Contents

1 Scope of the Chapter

This chapter is concerned with the generation of sequences of independent pseudo-random numbers.

2 Background to the Problems

A sequence of pseudo-random numbers is a sequence of numbers generated in some systematic way such that its statistical properties are as close as possible to those of a sequence of true random numbers: for example, the correlation between consecutive numbers should be negligible. The most common method used is the **multiplicative congruential** algorithm (see Knuth [2]). The NAG Parallel Library uses a variant of the multiplicative congruential algorithm known as the Wichmann–Hill algorithm (see [3]), defined as:

$$\begin{aligned} n_{1,i} &= (a_1 \times n_{1,i-1}) \bmod m_1 \\ n_{2,i} &= (a_2 \times n_{2,i-1}) \bmod m_2 \\ n_{3,i} &= (a_3 \times n_{3,i-1}) \bmod m_3 \\ n_{4,i} &= (a_4 \times n_{4,i-1}) \bmod m_4 \end{aligned} \tag{1}$$

$$U_i = \left(\frac{n_{1,i}}{m_1} + \frac{n_{2,i}}{m_2} + \frac{n_{3,i}}{m_3} + \frac{n_{4,i}}{m_4} \right) \bmod 1.0$$

This generates pseudo-random numbers U_i, uniformly distributed in the interval (0,1), from four initial values, $n_{0k}, k = 1, 2, 3, 4$, known as seeds.

On a parallel machine, there may be a requirement to generate statistically independent sequences of random numbers on each processor. The NAG Parallel Library generators provide 273 distinct sets of a_k and $m_k, k = 1, 2, 3, 4$, to allow a choice of 273 statistically independent generators. Each generator has a typical **cycle length** (i.e., the number of random numbers before the sequence starts repeating itself) of about 2^{80}. A good rule of thumb is never to use more numbers than the square root of the cycle length in any one experiment with the same generator, as the statistical properties are impaired. For closely related reasons, breaking numbers down into their bit patterns and using individual bits may cause trouble.

The use of the same seeds with a given generator will lead to the same sequence of numbers. One method of obtaining the seeds is to use the real-time clock; this will give a non-repeatable sequence. It is important to note that the statistical properties of the random numbers produced by a particular generator are only guaranteed within sequences and not between sequences. Repeated initialization for a given generator will thus render the numbers obtained less rather than more independent.

Random numbers from other distributions may be obtained from the uniform random numbers by the use of transformations and rejection techniques. In this release of the NAG Parallel Library other distributions are not provided but brief descriptions of the methods are given here.

Two distributions that are often used are the exponential and Normal distributions. A simple way to generate a variate from the exponential distribution with mean μ is the transformation $-\mu \log(1 - U_i)$. For the Normal distribution the polar Box–Müller method uses a pair of random numbers (U_1, U_2) to give a pair of Normal variates (X_1, X_2). The method involves a rejection stage; so not all pairs of random numbers are used. The following pseudo-code illustrates the transformations for two given random numbers U1 and U2:

```
Y1 = 2.0 * U1 - 1.0
Y2 = 2.0 * U2 - 1.0
S = Y1 * Y1 + Y2 * Y2
IF(S.LE.1.0) THEN
    B = SQRT(-2.0*LOG(S)/S)
    X1 = B * Y1
    X2 = B * Y2
ENDIF
```

The efficiency of a simulation exercise may often be increased by the use of variance reduction methods (see Morgan [4]). 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 routines in Chapter D01 rather than by Monte Carlo integration.

2.1 References

[1] Dagpunar J (1988) *Principles of Random Variate Generation* Oxford University Press

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

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

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

[5] Ripley B D (1987) *Stochastic Simulation* Wiley

3 Recommendations on Choice and Use of Available Routines

Note: Refer to the Users' Note for your implementation to check that a routine is available.

3.1 Design of the Chapter

There is a single routine for selecting one of the 273 generators provided and initialising the seeds, and a single function for returning the random numbers. If a specific generator and seeds are not selected, the generator assumes default values for the generator and seeds (which will therefore return the same sequence on each run).

G05AAFP returns a single random number from the sequence on each logical processor.

G05ABFP selects one of the generators and initializes the seeds to a repeatable state, dependent on a vector of 4 integers: two calls of G05ABFP with the same argument values will result in the same subsequent sequences of random numbers.

As mentioned in Section 2, it is important to note that the statistical properties of pseudo-random numbers from a given generator are only guaranteed within sequences and not between sequences. Repeated initialization of the same generator will thus render the numbers obtained less rather than more independent. In a simple case there should be only one call to G05ABFP, which should be before any call to an actual generation routine.

3.2 Selection of Routine

Only the uniform distribution on (0,1) is provided at this release.

Distribution	**Function returning a single random number**
uniform over (0,1)	G05AAFP

3.3 Programming Advice

Take care when programming calls to those routines in this chapter which are Fortran functions.

For example, if you wish to assign to Z the difference between two successive random numbers generated by G05AAFP, beware of writing

```
Z = G05AAFP() - G05AAFP()
```

It is quite legitimate for a Fortran compiler to compile zero, one or two calls to G05AAFP; if there are 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 = G05AAFP()
Y = G05AAFP()
Z = X-Y
```

Another problem that can occur is that an optimising compiler may move a call to a function out of a loop. Thus, the same value would be used for all iterations of the loop, instead of a different random number being generated at each iteration. If this problem occurs, consult an expert on your Fortran compiler.