

**Elements of Statistical Computing**  
**Volume 2 Draft Notes**

---

Ronald A. Thisted

© 2005 Ronald Thisted. All rights reserved.



---

# Contents

---

<b>List of Algorithms</b>	<b>ix</b>
<b>7 Random Number Generation</b>	<b>1</b>
7.1 An overview of simulation and random number generation	1
7.2 Uniform random number generation	3
7.2.1 Historical notes	3
7.2.2 Linear congruential generators	4
7.2.3 Shuffling and mixing	7
7.2.4 Portable random number generators	9
7.2.5 Feedback shift-register sequences	10
7.2.6 Other methods	11
7.3 Testing uniform generators	12
7.4 Nonlinear random number generation	15
7.4.1 General principles	15
7.4.2 Continuous distributions	17
7.4.3 Discrete distributions	17
7.5 Multivariate random numbers	20
7.6 Other random objects	20
7.7 Assessing quality of random number algorithms	20
7.8 Computing strategies	20
<b>8 Monte Carlo Methods</b>	<b>23</b>
8.1 Design of simulation studies	23
8.1.1 The simulation space	23
8.1.2 Smooth simulation studies	24
8.1.3 Non-convergence	27
8.2 The Metropolis-Hastings algorithm	29
<b>Answers to Exercises</b>	<b>31</b>
<b>References</b>	<b>33</b>
<b>Index</b>	<b>37</b>



---

## List of Algorithms

---

7.1.1 Random sampling with replacement on $\{0, \dots, m - 1\}$	2
7.2.1 Shuffling a uniform generator (MacLaren and Marsaglia, 1965)	8
7.2.2 Mixing linear congruential generators (Knuth, 1998)	9
7.4.1 Acceptance/rejection method for continuous variates	16
7.4.2 Tail of a normal distribution	16
7.4.3 Knuth's method for sampling from nearly linear densities	17
7.4.4 $\alpha$ -contaminated normal	17
7.4.5 Binomial random numbers using sequential search	17
7.4.6 Poisson random numbers using sequential search	18
7.4.7 Geometric random numbers by direct simulation	18
7.4.8 Geometric random numbers by cdf inversion	18
8.1.1 Monte Carlo investigation of $a\bar{X}$	24
8.1.2 Conventional simulation of the normal power function	25
8.1.3 Smooth simulation of the normal power function	26



## Random Number Generation

---

Even very simple questions arising from everyday statistical practice can give rise to complexities that are beyond straightforward mathematical analysis. For example, the distribution of estimated regression coefficients that are obtained from an automated stepwise regression cannot easily be derived. This question can be answered, however, by repeatedly sampling from a specified statistical model (perhaps tens of thousands of times), calculating and recording the stepwise regression estimates, and then observing the distribution directly from these sampled values. For this method to be practical, the repeated sampling is carried out by computer, and for this purpose, we need to be able to generate realizations from random variables with specified distributions.

A moment's reflection makes it clear that this goal as stated is impossible to achieve. The output from a computer algorithm is deterministic, not random. What we require instead are algorithms that produce sequences of output that behave in important ways like sequences of realizations from random variables would behave. Such algorithmically created numerical sequences are termed *random numbers*, *random variates*, or *random deviates* to distinguish them from the (mathematically ideal) random variables they are intended to simulate. The term "random variate" is particularly useful for non-numerical random objects such as random orthogonal matrices or random permutations of a set, but otherwise there is little merit in distinguishing these terms.

Just as random numbers are the fundamental building block of computer simulation studies, uniformly distributed random numbers are the building blocks for random numbers (and other random objects) with arbitrarily specified distributions. In this chapter we shall investigate methods for generating (and studying the quality of) uniform random numbers, and then we shall study how to use them to construct collections of other objects (including variates with nonuniform distributions, multivariate structure, and random permutations).

### 7.1 An overview of simulation and random number generation

The study of simulation begins with two simple observations. First, we note that many properties of statistical interest, including means, variances, and probabilities, can be expressed as expectations (or more generally, functionals of a distribution). The bias of an estimator is a function of its expectation; the mean squared error of an estimator is the expectation of its squared error;

the power of a test is the expectation of the indicator function for the test statistic lying within a critical region, and so forth.

Suppose that  $W$  is a random variable with distribution function  $P(\cdot)$  and density function  $p(\cdot)$ , and suppose that we would like to compute an expectation involving  $W$  such as

$$\mathbb{E}[g(W)] = \int g(w)p(w)dw = \int g(w)dP(w) < \infty. \quad (7.1.1)$$

The second basic observation is that, if

$$W_1, \dots, W_n \text{ i.i.d. } P(w),$$

then

$$\frac{1}{n} \sum g(W_i) \longrightarrow \mathbb{E}[g(W)] \text{ with probability 1,} \quad (7.1.2)$$

by Khinchine's strong law of large numbers. Moreover, if  $g(W)$  has finite variance, the left-hand side converges in distribution to a normal distribution, as well. Thus, even if the integral in equation (7.1.1) is intractable, the calculation on the left-hand side of (7.1.2) can be easily carried out, provided that obtaining samples from  $P$  and computing  $g(\cdot)$  are both easy.

In many cases, sampling from  $P$  turns out to be easy to do (at least in principle), provided that we have a method for sampling from the uniform distribution on  $(0, 1)$ . If  $U \sim U(0, 1)$  and  $P$  is absolutely continuous, then the *probability integral transform* of  $U$ , given by  $X = P^{-1}(U)$ , has the desired distribution. That is,

$$\Pr[X < t] = P(t). \quad (7.1.3)$$

Indeed, this same idea of inverting the cumulative distribution function works for any suitably defined  $P^{-1}$  even for discrete random variables. The use of the probability integral transform for discrete distributions will be a major topic in Section 7.4.3.

Perhaps the simplest application of the probability integral transform is randomly sampling with replacement from a set of  $m$  objects, which we identify with the integers 0 through  $m - 1$ . Algorithm 7.1.1 delivers a uniform random integer in this range.

---

**Algorithm 7.1.1** Random sampling with replacement on  $\{0, \dots, m - 1\}$

---

Assume that  $u()$  generates a  $U(0, 1)$  random number  
 Return  $\lfloor m \cdot u() \rfloor$

---

These considerations lead us to investigate generation of independent  $U(0, 1)$  random variates.

## EXERCISES

**EXERCISE 7.1** [12] Prove the probability integral transform result in equation (7.1.3) under the assumption that  $P$  is monotone increasing and continuous.

**EXERCISE 7.2** [20] (Continuation) Prove the result in equation (7.1.3) if one assumes that  $P$  is absolutely continuous.

**EXERCISE 7.3** [01] How should Algorithm 7.1.1 be modified to simulate the toss of a six-sided die?

## 7.2 Uniform random number generation

Physical methods of generating random numbers were known to the ancients; tossing coins and throwing dice are familiar examples. The advent of digital computers brought a need for lengthy streams of random numbers, and methods for generating them automatically were among the first general-purpose algorithms invented for the computing devices. Two fundamental types of uniform generators are in common use and are the basis for many variants: the linear congruential generators (LCGs) and the feedback shift-register (FSR) generators.

### 7.2.1 Historical notes

The need for simulating random variation was recognized early in the history of digital computation, and a variety of schemes were proposed for obtaining streams of random digits. Some early methods involved specialized machinery that depended on a physically unpredictable mechanism (such as coupling a weak radiation source with a detector and then counting particles detected during short periods of time). It was soon recognized, however, that results from such a mechanism could not be reproduced; each time a program was run to carry out a computation, a different stream of random inputs would be used. That made it impossible to determine whether an unexpected result was actually correct or was the result of a programming error, for instance. Indeed, figuring out a way to reproduce results that were based on random inputs was a challenging puzzle.

In principle, this could be done by tabulating the output of the random generator in a file, and then reading values from the file as needed. Indeed, that was done, and was the basis for the Rand Corporation's best selling book *A Million Random Digits with 100,000 Normal Deviates* (RAND Corporation, 1955). To use a stream of digits from the Rand tables, for instance, one need only specify a starting position in the table. Successive digits could then be used for computation. In the early days of computing, however, storing even a table of modest size was expensive and retrieval was slow. (In the early 1970s, for instance, the Rand tables were too large to store on disk drives. One had to read through them sequentially from a magnetic tape.)

It was natural, then, to seek computer algorithms that could emulate random variables, so that each time a value was needed, it could be computed rather than looked up in a stored table. As with all computer algorithms, the output is a deterministic function of the input. For this reason, computer-generated random numbers are often described as being *pseudorandom*. All

computer algorithms for generating random numbers operate by performing a computation on an input value, called the generator's *state*, generating an output value which is the next number in the sequence, and then updating the generator's state to prepare for the next call to the algorithm.

A typical generator of uniform random numbers on  $[0, 1)$  for computers with 32-bit words will use a 32-bit state  $X_i$ , which can be thought of as an integer  $0 \leq X_i < 2^{32}$ . The state is updated as  $X_{i+1} = f(X_i)$ , and the generated output is simply  $U_{i+1} = X_{i+1}/2^{32}$ . Since there are at most  $2^{32}$  different values for the state, the sequence produced by such a generator is periodic, with period of at most  $2^{32}$ , about 4.3 billion. Indeed, all algorithms for generating pseudorandom sequences are periodic, with the maximal period depending on the amount of storage required for the generator's state.

### 7.2.2 Linear congruential generators

The simple linear congruential generator, the idea for which is due to D. H. Lehmer, has an integer state  $X_i \in \mathcal{Z}^m$ , the integers modulo  $m$ , and a successor rule defined by

$$X_i = (aX_{i-1} + c) \bmod m.$$

Thus, a linear congruential generator is completely defined by four integers: the *multiplier*  $a$ , the *increment* or constant  $c$ , the *modulus*  $m$ , and the initial value of the state, or the *seed*,  $X_0$ . Ideally, we would like the sequence of output from such a generator to simulate independent draws from the uniform distribution on the integers  $\{0, 1, \dots, m-1\}$ .

To obtain  $U(0, 1)$  random numbers from a linear congruential sequence, one can simply scale the current state  $X_i$  using

$$U_i = X_i/m,$$

which implies that the uniform sequence can be written as

$$U_i = (aU_{i-1} + f) \bmod 1. \quad (7.2.1)$$

Note that on real computers, the integer state  $X_i$  will typically have more bits of precision than will the floating-point uniform  $U_i$ , so that expression (7.2.1) cannot generally be implemented in the obvious manner.

#### 7.2.2.1 Multiplicative generators

Generators with  $c = 0$  are sometimes called *multiplicative* congruential generators. (In this case,  $X_0 = 0$  is not allowed!)

It is easy to see that the choice for the multiplier  $a$  matters in producing a good multiplicative sequence, but it is not obvious how to choose a "good" value for  $a$ . For example, suppose that  $m = 2^p$  and  $X_0 = 1$ . Choosing  $a = 1$  produces a sequence with period 1. Choosing  $a = 2$  produces a sequence that is unequally spaced, and that then degenerates after a few steps into another degenerate sequence (after generating  $X_i = 0$ ).

The longest possible period for a multiplicative generator with  $m = 2^p$  is  $2^{p-2}$ , which can be achieved if

$$a \equiv \pm 3 \pmod{8}.$$

The low-order bits are not very random, but the high-order bits behave reasonably well. These generators were attractive for old, slow computers, for which the  $U(0,1)$  could be obtained simply by placing the binary point in front of the high-order bits. On many computers this required a single machine instruction to accomplish.

### 7.2.2.2 Structure of linear congruential generators

George Marsaglia (1968) showed that pairs, triples, and in general,  $k$ -tuples, formed using successive elements from a linear congruential sequence fall on points of a lattice in the unit square, cube, or hypercube. This lattice structure is characteristic of all linear congruential generators, and the nature of the lattice structure determines the statistical properties of the sequence. In high dimensions (say,  $k > 6$ ) the number of hyperplanes on which these points lie is very small, so successive values from linear congruential sequences cannot be used directly for generating highly multivariate random deviates.

However, some generators have a lattice structure even in two or three dimensions that is unacceptable. In short, the problem is that the distance between planes is unacceptably large, leaving large areas of the unit cube, for instance, with no generated points. The *discrepancy* is a measure of the largest distance between planes, and can be calculated using number-theoretic methods for any linear congruential generator.

*Example 7.1.* **RANDU** is a generator introduced in the Scientific Subroutine Package, a collection of FORTRAN (later PL/I) subroutines that IBM distributed in the 1960s and 1970s to encourage use of their System/360 computers as a scientific tool. It is a multiplicative LCG of the form

$$X_i = 65539X_{i-1} \pmod{2^{31}}.$$

This generator appeared to have solid theoretical support for having high quality; it minimizes an upper bound for the two-dimensional autocorrelation, that is, for the correlation of  $X_i$  and  $X_{i-1}$ . Unfortunately, it turns out that the lattice structure of **RANDU** is about the poorest possible.

The multiplier for this LCG has  $a = 65539 = 2^{16} + 3$ . As a result, we can easily write out  $X_i$  in terms of the two preceding terms in the sequence.

$$\begin{aligned} X_i &= (2^{16} + 3)X_{i-1} \pmod{2^{31}} \\ &= (2^{16} + 3)^2 X_{i-2} \pmod{2^{31}} \\ &= (2^{32} + 6 \cdot 2^{16} + 9)X_{i-2} \pmod{2^{31}} \\ &= 6X_{i-1} - 18X_{i-2} + 9X_{i-2} \pmod{2^{31}} \\ &= 6X_{i-1} - 9X_{i-2} \pmod{2^{31}}. \end{aligned} \tag{7.2.2}$$

Thus,

$$X_i - 6X_{i-1} + 9X_{i-2} \equiv 0 \pmod{2^{31}}$$

or, equivalently,

$$U_i - 6U_{i-1} + 9U_{i-2} \equiv 0 \pmod{1}.$$

In other words, we can write

$$U_i - 6U_{i-1} + 9U_{i-2} = k,$$

and since each  $U$  must satisfy  $0 < U < 1$ , we can conclude that  $-6 < k < 10$ , implying that the points lie on at most 15 planes.

Unfortunately, the poor structure of **RANDU** was discovered only after the generator had achieved wide popularity as a “good” random number generator. Even today, the **RANDU** generator still makes appearances. Beware of any generator with a multiplier of 65539!

### 7.2.2.3 Mersenne generators

A widely used class of linear congruential generators uses Mersenne prime moduli, that is, moduli of the form  $m = 2^p - 1$ , where  $m$  is also prime. These generators can have periods of length  $m - 1$ , and have the added advantage that they never generate a zero.

It is particularly fortunate that  $2^{31} - 1$  is prime and uses nearly all of the bits available in a 32-bit integer, regardless of the sign convention used by a particular computer architecture. For this reason, generators with this modulus have been well studied, and solid implementations of many of them are available for most computers.

To assure that the generator has the maximal period  $m - 1$ , it is sufficient that  $a$  be a *primitive root* mod  $m$ , that is,  $m - 1$  is the smallest value of  $k$  for which  $a^k \equiv 1 \pmod{m}$ .

*Example 7.2.* Lewis, Goodman, and Miller (1969) investigated a linear congruential generator with multiplier  $a = 7^5 = 16807$  and with modulus  $m = 2^{31} - 1$ , the largest prime number of this form that can be stored in a full-word (32-bit) integer on IBM System/360 computers (a popular mainframe computer architecture introduced in the 1960s). Because the multiplier can be represented in only 15 bits, any integer product will require no more than 46 bits to represent exactly.

Since the IBM 360 automatically stored the product of two 32-bit integers as a 64-bit result, the Lewis-Goodman-Miller generator could be easily implemented using IBM 360 machine-level instructions. Unfortunately, implementations of **FORTRAN** and other high-level languages in use at the time could not access these intermediate double-precision integer results. Schrage (1979) showed how the Lewis-Goodman-Miller generator could be implemented in **FORTRAN** by simulating double precision arithmetic in software and then using a result from number theory to carry out the reduction modulo  $m$ . The result was a short subroutine that could be used on a wide variety of computers with different architectures. The Lewis-Goodman-Miller generator became a widely studied and widely used random number generator.

### 7.2.2.4 Deficiencies of simple LCGs

The simple linear congruential generators described above have two evident drawbacks. First, the period is relatively short. With a 32-bit state, the max-

imal period is  $2^{32} \approx 4.3$  billion, and often less. This is far too small for many of today's applications. Second, the lattice structure described for `RANDU` is characteristic of linear congruential generators. This introduces a systematic characteristic into the random number stream that can be fatal in a serious simulation that depends on higher-dimensional coverage of the simulation space.

Alternatives to the simple LCG that retain much of their simplicity involve both expanding the state space and introducing elements that attempt to “break up” the lattice structure. The next few sections examine some of these approaches.

### 7.2.3 *Shuffling and mixing*

Several approaches have been proposed for “improving” the properties of linear congruential generators, although the rationales for the improvements are based on theoretical arguments that involve uniform random variables—which we can only simulate. As a result, the improved generators are usually not provably better; instead they are implementations of an heuristic argument that suggests that they *should* be superior to an unmodified linear congruential sequence. What can be shown empirically is that they do remove the specific deficiencies each is designed to address; it is harder to demonstrate that new subtle deficiencies have not been introduced. Nonetheless, these methods are often employed.

To improve the statistical performance of early generators, MacLaren and Marsaglia (1965) proposed the method of Algorithm 7.2.1, in which the output from one linear congruential sequence is used to “shuffle” the output from another. Although MacLaren and Marsaglia take the shuffling generator `u()` in the algorithm to be an LCG, the algorithm could be applied to any such sequence. Clearly, this method will break up the lattice structure of successive values from the LCG.

Bays and Durham (1976) suggested a variation on Algorithm 7.2.1 in which a single linear congruential sequence is used. The reservoir `A[·]` is first initialized, then at each stage the next random number from the sequence is used to generate an index into the `A` array, which is then delivered as the output and replaced by the next element from the linear congruential sequence. Thus, two successive uniforms are used to generate each output random number.

Shuffling can dramatically increase the period of the generator, because the generator's state now consists of the combined states of the two constituent random number generators. (This is not the case for the Bays/Durham method, however.) In addition, the contents of the `A` array also affect the state. Calculating the maximal period for such a generator is a surprisingly complex matter, and the shuffle makes it difficult to prove properties of such a generator. The exercises indicate some directions along these lines.\*

\* Need to check out the later work by Bays (1991; 1990).

**Notes on the revision.** Although little may be provable about generators that shuffle their output, empirical tests suggest that they can have much better behavior than their unshuffled counterpart. Nonetheless, it is possible to construct simple examples in which the shuffled generator is less random than the raw sequence. For instance,

This needs to be checked and expanded.

---

**Algorithm 7.2.1** Shuffling a uniform generator (MacLaren and Marsaglia, 1965)

---

```

Assume that  $A[\cdot]$  is a  $k$ -element, 0-indexed array
Assume that  $u()$  generates a  $U(0, 1)$  random number (production)
Assume that  $j()$  generates a random integer mod  $k$  (selection)
if this is the first invocation then
    for  $j \leftarrow 0$  to  $k - 1$  do
         $A[j] \leftarrow u()$ 
    end for
end if
 $i \leftarrow j()$ 
 $U \leftarrow A[i]$ 
 $A[i] \leftarrow u()$ 
Return  $U$ 

```

---

If a generator  $C_i$  is deficient, how can it be made better? If we define

$$X_i = C_i + U_i \bmod 1,$$

where  $U_i \sim U(0, 1)$  are i.i.d. and independent of  $C_i$ , then the properties of the  $C_i$  sequence are entirely irrelevant to the properties of  $X_i$ , which is an i.i.d.  $U(0, 1)$  sequence. In other words, mixing anything with true uniform random variables produces true uniform random variables. This *suggests* that mixing a “good” random number generator with a possibly deficient generator will result in a generator that is also at least “good” and perhaps “better.” L’Ecuyer (1988) proposes mixing LCGs that are suitable for portable implementation to obtain combined generators with long periods and desirable statistical properties.<sup>†</sup>

A generator of this type with a period of approximately  $7 \times 10^{16}$  and with good lattice structure was suggested by Knuth (1998), and is given in Algorithm 7.2.2.

<sup>†</sup> See Gentle, 1ed, 1.7.2.

**Algorithm 7.2.2** Mixing linear congruential generators (Knuth, 1998)

---

```

Initialize state  $(x, y)$ 
 $x \leftarrow 48271 x \bmod 2^{31} - 1$ 
 $y \leftarrow 40692 y \bmod 2^{31} - 249$ 
 $z \leftarrow (x - y) \bmod 2^{31} - 1$ 
Return  $U = z / (2^{31} - 1)$ 

```

---

*7.2.4 Portable random number generators*

“Portable” random number generators are intended to be programs whose output streams are independent of the compiler, operating system, and hardware architecture of the computer on which the program is run. Linear congruential generators, which rely on exact integer modular arithmetic, can be implemented portably only when the language in which the program is written permits these exact computations over the full range of integers that the algorithm can produce. Programming languages such as FORTRAN can make this a surprisingly elusive goal to achieve. Nonetheless, several random number generators, with varying degrees of portability, have been proposed. The Lewis-Goodman-Miller generator discussed above was introduced as such a portable generator. The limitations of either the programming language or underlying differences in computer hardware behavior were circumvented by such methods as using small multipliers or small moduli for congruential generators and emulating multiple-precision integer arithmetic in software.

Wichmann and Hill proposed a method for generating a very long pseudorandom sequence in a portable fashion. Their method uses three linear congruential streams with mutually prime periods, which are then combined at each stage to produce a single output between 0 and 1. The combining method is based on the observation that, if  $U_1, U_2, \dots, U_n$  are i.i.d.  $U(0,1)$ , then the fractional part of  $S = \sum_1^n U_i$  also has a  $U(0,1)$  distribution, regardless of  $n$ . The three linear congruential generators have moduli 30269, 30307, and 30323, and multipliers 171, 172, and 170, respectively. At each stage the fractional part of the sum of the three uniforms is returned. Zeisel notes that the stream of triples generated by the three congruences of the Wichmann-Hill generator is isomorphic to the stream from a single linear congruential generator whose parameters can be deduced from the Chinese Remainder Theorem. However, the uniform random number returned by the Wichmann-Hill algorithm is not simply the ratio of the single generator’s output to its modulus. While the properties of the latter would be easily studied, there have been no analytic studies of the actual Wichmann-Hill generator. Nonetheless, it appears to perform well in practice.

**Notes on the revision.** Wichmann and Hill (1982; 1985).

Some additional references: Deng and Lin (2000); Dagpunar (1989); Gentle (1981); Hörmann and Derflinger (1993); L’Ecuyer (1988); McLeod (1985); Sharp and Bays (1992, 1993); Tezuka and L’Ecuyer (1991); Wollan (1992); Zeisel (1986)

### 7.2.5 Feedback shift-register sequences

One of the primary drawbacks of linear congruential generators is their relatively short periods, a consequence of the typically small states for easily implemented versions of these generators. A natural approach is to generalize the linear congruence to a similar recursion relationship with a larger state.

An example is the *Fibonacci generator*, defined by the recurrence

$$X_i = X_{i-1} + X_{i-2} \bmod m. \quad (7.2.3)$$

This generator has state  $(X_{i-1}, X_{i-2})$  which, for  $p$ -bit words raises the possibility of sequences with period of  $2^{2p}$ . Unfortunately, the three dimensional structure of all Fibonacci generators is so deficient that they should never be used by themselves for generating random numbers (see Exercise 7.17).

The *generalized Fibonacci generators* given by the recurrence

$$X_i = X_{i-j} + X_{i-k} \bmod m, \quad (7.2.4)$$

where  $1 < j < k$ , can produce much better random number sequences than the generator of 7.2.3 above. The state of the generator in 7.2.4 requires the  $k$  values  $(X_{i-1}, X_{i-2}, \dots, X_{i-k})$ .

Another generalization of the simple Fibonacci sequence is defined by the recurrence

$$X_i = a_1 X_{i-1} + a_2 X_{i-2} \bmod m, \quad (7.2.5)$$

which is called a *multiple recursive generator* (see, for example, L'Ecuyer et al., 1993; L'Ecuyer, 1996). The GNU Scientific Library contains the routine `knuthran2`, which implements a particularly attractive example of such a generator, suggested by Knuth (1998, page 108). This generator passes a version of the spectral test with flying colors, and has a period of  $(2^{31} - 1)^2 - 1$ .

The linear congruential recurrence can easily be generalized to

$$X_i = a_1 X_{i-1} + a_2 X_{i-2} + \dots + a_k X_{i-k} \bmod m. \quad (7.2.6)$$

Although such generators hold out the promise of quite long periods, they are also quite expensive to compute, except in special cases. One particular special case is that of generating random bits, that is, with  $m = 2$ . In this case, each element of the sequence, and each coefficient in Equation 7.2.6, is either zero or one, making computation very easy. Tausworthe (1965) proposed such an algorithm for generating random numbers constructed by concatenating bits generated in this way; his method could take advantage of basic machine instructions that could shift the bits in a computer word and that could perform bit-wise operations such as exclusive-or on full computer words. (The exclusive-or operator is equivalent to addition modulo 2.) These generators were called feedback shift register generators, after the hardware features that facilitated implementation.

A generalization of the feedback shift register method, called inventively enough, the generalized feedback shift register (GFSR) method, produces generators with very long periods. In effect, The  $X$ 's, which are binary bits in expression 7.2.6, are replaced by 32-bit integers and the addition modulo 2 is

replaced by the exclusive-or operation. The result is 32 feedback shift registers (one for each bit position) that are being updated in parallel. The assembled collection of bits in  $X_i$  then becomes the output random number. A four-register version of this approach with largest lag equal to 9689 is implemented in the `gfsr4` generator of the GNU Scientific Library. This implementation has a period of  $2^{9689} - 1$ ; some of its properties are investigated in (Ziff, 1998).

These generators have two major drawbacks. They require large state vectors that must be carefully constructed (as many possible state vectors can lead to degenerate sequences), and they typically must be run for many iterations (called a “burn-in” period) before their statistical behavior is adequate.

### 7.2.6 Other methods

A number of other algorithms have been proposed, implemented, and studied. A few of those that one is most likely to encounter are discussed briefly here.

#### 7.2.6.1 Super-Duper and KISS

George Marsaglia has constructed random number generators that are portable, with long periods, and that pass a battery of tests. Marsaglia developed the Super-Duper generator along these lines in the early 1970s. This generator combined the output of a simple linear congruential generator mod  $2^{32}$  with that of a double-shift generator. (A shift generator operates by exclusive-ORing the bits of a computer word with a copy of that same word in which all of the bits have been shifted left or right a fixed number of bits. The particular double-shift generator Marsaglia used in Super-Duper first calculates  $Y \leftarrow Y \text{ xor } (Y \ll 15)$ , and then computes  $Y \leftarrow Y \text{ xor } (Y \gg 17)$ , where  $Y \ll a$  denotes shifting the bits of  $Y$   $a$  positions to the left (and filling the vacated positions by zeros). Similarly,  $Y \gg a$  denotes a shift of  $a$  bits to the right.

The KISS generator (for “keep it simple, stupid”) is actually a family of slightly more elaborate portable generators that Marsaglia recommends. Described in an unpublished technical report in 1993 by Marsaglia and A. Zaman, KISS generators are formed by combining the results of a simple linear congruential generator, a three-shift generator, and a lagged multiply-with-carry generator. With appropriately chosen constants, these generators have period of approximately  $2^{123}$ . Robert and Casella (1999, Section 2.1.2) give a detailed description of the method. KISS is used as the standard uniform generator in *Stata*.

**Notes on the revision.** The following code was posted by Marsaglia to `sci.math.num-analysis` on 17 January 2003 as a version of KISS suitable for C compilers that support long double (unsigned long long) integers. It is reproduced here without permission; permission will be needed to include in the book.

```
/*
 * Four random seeds are required, 0<=x<2^32, 0<y<2^32, 0<=z<2^32,
 * 0<=c<698769069. If the static x,y,z,c is placed outside the KISS proc,
```

```

* a seed-set routine may be added to allow the calling program to set
* the seeds.
*/
unsigned long KISS(void){
    static unsigned long x=123456789,y=362436,z=521288629,c=7654321;
    unsigned long long t,
    a=698769069LL;
    x=69069*x+12345;
    y^=(y<<13);
    y^=(y>>17);
    y^=(y<<5);
    t=a*z+c;
    c=(t>>32);
    return x+y+(z=t);}

```

In the same post, Marsaglia gives C code for a multiply-with-carry generator (MWC1038) with period  $3056868392^{33216} - 1$  which also passes all standard tests for randomness. That code is given here, too:

```

/*
* You need to assign random 32-bits seeds to the
* static array Q[1038] and to the initial carry c,
* with 0<=c<61137367.
*/
static unsigned long Q[1038],c=362436;

unsigned long MWC1038(void){
    static unsigned long i=1037;
    unsigned long long t, a=611373678LL;
    t=a*Q[i]+c;
    c=(t>>32);
    if(--i) return(Q[i]=t);
    i=1037; return(Q[0]=t);
}

```

### 7.2.6.2 Mersenne Twister

The “Mersenne Twister” is an example of a “twisted” GFSR generator (Matsumoto and Kurita, 1994, 1992; Matsumoto and Nishimura, 1998), and one such generator is used as the default random number generator in R. The generator `mt19937` in the GNU Scientific Library is an implementation with period  $2^{19937} - 1$ , which is a Mersenne prime (Matsumoto and Nishimura, 1998).

Marsaglia notes, with some pride, that `MWC1038` is faster and simpler than `mt19937`, and has a period that is about  $10^{4005}$  times longer as well.

## 7.3 Testing uniform generators

There are three criteria commonly use to select random number generators: theoretical, empirical, and reputational. Of the three, relying on the latter

(“The `fab5` generator is the one I use, and it is a really good one”) is most uncertain. Like urban legends and herbal medicines, many highly recommended generators do not live up to their press, and the truth is often hard to determine. It is much better to rely on provable or otherwise demonstrable properties of specific generators.

Theoretical tests essentially measure specific aspects of the mathematical structure of a generator relevant to the statistical properties that the generated sequence will exhibit. One such measure examines (hyper-)rectangular regions in the unit cube and compares the actual number of  $N$  successive  $k$ -tuples  $(U_i, U_{i+1}, \dots, U_{i+k-1})$  that fall in the rectangle to the expected number assuming that the  $k$ -tuples were independent draws from a  $k$ -dimensional uniform random variable. This maximum of the quantity over all rectangles is called the *discrepancy* (see section 3.3.4F of Knuth, 1998). This measure can be subsumed by the *spectral test* (Coveyou and MacPherson, 1967; Golder, 1976; Hopkins, 1983), which measures the greatest distance between hyperplanes in the lattice structure for  $k$ -tuples, in the sense that when the latter number is small, the former must be as well. Moreover, the spectral test computations can be carried out for a number of classes of generators, including the linear congruential and generalized feedback shift register generators. The spectral test, described in detail with examples by Knuth (1998, section 3.3.4), has proven to be an important and versatile tool for identifying good (and weeding out bad) generators.

Empirical tests essentially calculate goodness-of-fit tests (such as chi-squared or Kolmogorov-Smirnov) for quantities generated from the random number sequence. These range from simple tests of equidistribution (convert the uniform deviate to a random integer mod  $k$ , count the number of times each of the  $k$  possibilities is generated, and compare to expectation using the  $\chi^2$  test), to the silly (generate random permutations of 52 elements—identified with playing cards, select the first five “cards” in the permutation, evaluate as a poker hand, then compare the number of times each type of poker hand is generated to expectation, again using the  $\chi^2$  test), to the incredibly elaborate (Marsaglia’s `DIEHARD` battery of tests). Such tests, while in some respects *ad hoc*, have several virtues: they have identified deficient generators; when generators fail, they clarify the nature of the discrepancy; they can be applied to any random number generator; and they can be specially constructed to examine problem-specific requirements.

## EXERCISES

**EXERCISE 7.4** [32] Radiation counts using Geiger detectors typically follow a Poisson distribution, which leads to the following model for a physical random digit generator. If uniformly distributed digits  $\{0, 1, \dots, m-1\}$  are desired, a natural method would be to take  $X_i = G_i \bmod m$  as the output, where  $G_i$  are radiation counts in successive seconds. Investigate the distribution of this sequence as a function of  $m$  and of  $\lambda$ , under the assumption that the  $G_i$  values are i.i.d. Poisson with rate  $\lambda$ .

**EXERCISE 7.5** [18] Show that if the selector generator  $j()$  in Algorithm 7.2.1 is a uniform random variable on  $\{0, 1, \dots, k-1\}$ , then the resulting generator is aperiodic.

**EXERCISE 7.6** [44] Assume that the selector generator  $j()$  in Algorithm 7.2.1 has a state with at most  $m_1$  values, and that the production generator  $u()$  has a state with at most  $m_2$  values. Determine whether the period of the resulting shuffled generator can exceed  $m_1 m_2 (k!)$ .

**EXERCISE 7.7** [48] (Continuation) Investigate the maximal period and other properties of the Bays/Durham variant of Algorithm ??.

**EXERCISE 7.8** [47] Analyze the properties of the following variant of the Bays/Durham/Maclaren/Marsaglia shuffler, which uses a single linear congruential sequence  $X_i$ , and which uses the same element of the sequence both to select the element of  $A[\cdot]$  to deliver and to refill this element of the array.

**EXERCISE 7.9** [28] Write a subroutine or function that implements the Lewis-Goodman-Miller linear congruential generator with  $a = 7^5 = 16807$  and  $m = 2^{31} - 1$ . This is the “minimal standard” generator as defined by Park and Miller (1988). To verify the correctness of your generator, set  $x_0 = 1$  and use your function to produce  $x_{100} = 892053144$ .

**EXERCISE 7.10** [25] Repeat the previous exercise, using the multiplier  $a = 742938285$ , a choice found in an exhaustive search for the best linear congruential generators with modulus  $2^{31} - 1$  (Fishman and Moore, 1986). This LCG has been used in some portable random number generator implementations.

**EXERCISE 7.11** [23] (Wichmann and Hill, 1982) Let  $U_1, U_2, \dots$ , be independent and identically distributed  $U[0, 1)$ . Define  $S_n = \sum_1^n U_i$ , and let  $Z_n = \{S_n\} \equiv S_n - \lfloor S_n \rfloor$  be the fractional part of  $S_n$ . Show that  $Z_n$  has the distribution  $U[0, 1)$  for all  $n$ .

**EXERCISE 7.12** [35] (Continuation) (Patrick Billingsley) Prove that if the  $U_i$  of the previous exercise are i.i.d. according to any distribution with an absolutely continuous density with respect to Lebesgue measure, then  $Z_n$  converges in law to  $U[0, 1)$ .

**EXERCISE 7.13** [27] (Zeisel, 1986) Show that the collection of triples  $(X_i^{(1)}, X_i^{(2)}, X_i^{(3)})$ , where  $X_i^{(j)} = a_j X_{i-1}^{(j)} \bmod m_j$  denotes the  $j$ -th congruence from the Wichmann-Hill random number generator, is isomorphic to a single linear congruential generator of the form  $X_i = a X_{i-1} \bmod m$ , and explicitly obtain values for the parameters  $a$  and  $m$ .

**EXERCISE 7.14** [30] Identify the *specific* random number generator that is “built in” to a particular mathematical or statistical software product (such as SAS, Stata, S-Plus, Minitab, Matlab, Maple, Mathematica, etc) or a particular implementation of a programming language (such as C, C++, or FORTRAN). Be sure to identify the particular version of the software to which your research

applies, and provide enough details of the random number generation so that, in principle, its specific properties could be investigated.

**EXERCISE 7.15** [15] (Continuation) For the generator you investigated in the previous exercise, how do you set the seed (or the initial state)? How do you recover the current value of the seed (or state) in order to restart the sequence from its current position?

**EXERCISE 7.16** [12] Why would it be undesirable for the output from a uniform random number generator to take on the values (exactly) zero or one?

**EXERCISE 7.17** [15] If  $\{U_i\}$  is a sequence of independent uniform random variables, then each of the six ways in which  $(U_i, U_{i-1}, U_{i-2})$  can be ordered is equally likely. Let  $U_i = U_{i-1} + U_{i-2} \bmod 1$ . This is the so-called Fibonacci generator. Show that Fibonacci generators can never produce the ordering  $U_{i-1} > U_i > U_{i-2}$  or the ordering  $U_{i-2} > U_i > U_{i-1}$ .

**EXERCISE 7.18** [04] Why is the state for the generalized Fibonacci generator of Equation 7.2.4 the set of  $k$  values  $(X_{i-1}, X_{i-2}, \dots, X_{i-k})$  and not just  $(X_{i-j}, X_{i-k})$ ?

## 7.4 Nonlinear random number generation

### 7.4.1 General principles

#### 7.4.1.1 Inversion methods

#### 7.4.1.2 Acceptance/rejection methods

Suppose that  $X$  is a random variable with cumulative distribution function  $G(\cdot)$  and density function  $g(\cdot)$ , and suppose that  $f(\cdot)$  is another function that is majorized by the  $X$  density, that is,  $0 \leq f(t) \leq g(t)$  over the support of  $g(\cdot)$ . Let  $U$  denote a  $U(0, 1)$  random variable independent of  $X$ . Then the conditional distribution of  $X$  given that  $U \leq f(X)/g(X)$  is that of the random variable whose density is proportional to  $f(\cdot)$  over the support of  $g(\cdot)$ .

This observation suggests a method for generating random variates that may be hard to generate directly, but whose densities may be “close” to that of another random variate that is easy to sample from. A straightforward implementation of the conditioning argument above generates a trial candidate  $X$  from  $G$ , then accepts this candidate with probability  $f(X)/g(X)$ , where  $f$  is proportional to the desired sampling density (Algorithm 7.4.1).

The probability of acceptance, and hence the efficiency of the rejection step, depends on how closely  $g$  matches  $f$ ; a straightforward argument shows that this probability is equal to  $c = \int f(t)dt$ , where the range of integration is the support of  $g$ . The expected number of rejection tests per delivered random variate is  $1/c$ .

**Algorithm 7.4.1** Acceptance/rejection method for continuous variates

---

Assume that  $0 \leq f(t) \leq g(t)$  for  $t \in \text{support } g$   
 Assume that  $g(\cdot)$  generates a r.v. with density  $g(\cdot)$   
**repeat**  
      $Y \leftarrow g(\cdot)$   
      $U \leftarrow \text{unif}(\cdot)$   
**until**  $U \leq f(Y)/g(Y)$   
 Return  $Y$  [A r.v. with density proportional to  $f(\cdot)$ ]

---

Note that if  $f$  is replaced by  $af$  with  $0 < a \leq 1$  then the distribution of the algorithm's output is unchanged, but the acceptance probability is  $ac$ . This implies that the efficiency of the algorithm is maximized if  $\max\{f(t)/g(t)\} = 1$  over the support of  $g$ . It also means that we do not need to compute the normalizing constant  $c = \int f(t)dt$ , we need only be able to compute a function  $f$  that is proportional to the desired density.

**Algorithm 7.4.2** Tail of a normal distribution

---

Assume  $a > 0$   
**repeat**  
      $U \leftarrow \text{u}(\cdot)$   
      $V \leftarrow \text{u}(\cdot)$   
      $X \leftarrow \sqrt{a^2 - 2 \log U}$   
**until**  $VX \leq a$   
 Return  $X$

---

A simple, but useful example of the acceptance/rejection method generates a sample from the tail of the normal distribution, that is, sampling from the conditional distribution of  $Z$  given that  $Z > a$ , where  $Z \sim N(0, 1)$ . Algorithm 7.4.2 (Devroye, 1986) works because the density of  $X$  in the acceptance test can be written as

$$g(x) = xe^{-x^2/2} \geq ae^{-x^2/2} \equiv f(x), \text{ for } x \geq a,$$

which is proportional to the desired density. Moreover, because  $g(a) = f(a)$ , the majorized density is already scaled to make the acceptance test maximally efficient.

COMMENT. The simplest tail-area sampling method is to sample normal deviates until obtaining one larger than  $a$ , but the expected number of rejections makes this method prohibitive except for the smallest values of  $a$ . By contrast, Algorithm 7.4.2 becomes more and more efficient as  $a$  increases, approaching 1 as  $a \rightarrow \infty$ . Even for small  $a$ , the algorithm isn't too bad, accepting at least half the time provided  $a > 0.6121$ . At  $a = 3$  the acceptance probability is over 90%, and exceeds 95% whenever  $a \geq 4.1539$ .

**Notes on the revision.** *Examples: simple mixtures, alias method of Walker.*

---

**Algorithm 7.4.3** Knuth's method for sampling from nearly linear densities  
KNUTH GOES HERE

---

#### 7.4.1.3 Transformations

**Notes on the revision.** *Example: Exponential as  $-2\log(U)$ , Cauchy as ratio of two normals, ratio-of-uniforms methods.*

#### 7.4.1.4 Mixture methods

**Notes on the revision.** *Examples: simple mixtures, alias method of Walker.*

#### 7.4.2 Continuous distributions

---

**Algorithm 7.4.4**  $\alpha$ -contaminated normal

---

```

 $x \leftarrow \text{normal}(0, 1)$ 
if  $u() < \alpha$  then
     $x \leftarrow 3x$ 
end if
Return  $x$ 

```

---

#### 7.4.3 Discrete distributions

---

**Algorithm 7.4.5** Binomial random numbers using sequential search

---

```

Generate Binomial( $n, \theta$ ) on each call
 $p_0 \leftarrow (1 - \theta)^n$ 
 $x \leftarrow 0$ 
 $P \leftarrow p_0$ 
 $F \leftarrow p_0$ 
 $U \leftarrow u()$ 
while  $U > F$  do
     $x \leftarrow x + 1$ 
     $P \leftarrow P \left( \frac{\theta}{1-\theta} \right) \binom{n-x+1}{x}$ 
     $F \leftarrow F + P$ 
end while
Return  $x$ 

```

---

**Algorithm 7.4.6** Poisson random numbers using sequential search

---

```

Generate Poisson( $\lambda$ ) on each call
 $p_0 \leftarrow e^{-\lambda}$ 
 $x \leftarrow 0$ 
 $P \leftarrow p_0$ 
 $F \leftarrow p_0$ 
 $U \leftarrow \mathbf{u}()$ 
while  $U > F$  do
     $x \leftarrow x + 1$ 
     $P \leftarrow P\lambda/x$ 
     $F \leftarrow F + P$ 
end while
Return  $x$ 

```

---

**Algorithm 7.4.7** Geometric random numbers by direct simulation

---

```

Generate Geometric( $\theta$ ) on each call
 $x \leftarrow 0$ 
repeat
     $U \leftarrow \mathbf{u}()$ 
     $x \leftarrow x + 1$ 
until  $U < \theta$ 
Return  $x$ 

```

---

**Algorithm 7.4.8** Geometric random numbers by cdf inversion

---

```

Generates Geometric( $\theta$ ) on each call
 $x \leftarrow \lceil \log(\mathbf{u}()) / \log(1 - \theta) \rceil$ 
Return  $x$ 

```

---

**EXERCISES**

**EXERCISE 7.19** [22] The IBM Scientific Subroutine Package (SSP) was a collection of FORTRAN subroutines distributed with IBM System/360 computers in the late 1960s and 1970s. Two random number generators were included in the SSP. The uniform generator, RANDU, is an infamous linear congruential generator. The other generator produced normally distributed random numbers, by calculating  $W = \sum_{i=1}^{12} U_i - 6$ , where the  $U_i$ 's were generated as uniform random numbers. Why might  $W$  be considered as a standard normal deviate? What deficiencies can you identify in this method?

**EXERCISE 7.20** [15] Construct an algorithm for generating  $X = \min(U_1, U_2, \dots, U_n)$ , where the  $U_i$  are independent uniform random variables

on  $(0, 1)$ . The expected amount of time your algorithm takes to generate a single value should be independent of  $n$ .

**EXERCISE 7.21** [21] Describe how you would generate values from the distribution of the maximum of  $k$  exponential random variables. An exponential random variable has distribution function  $F(t) = 1 - \exp(-t/\lambda)$ , where  $\lambda > 0$ . The expected time for each variate should not depend on  $k$ .

**EXERCISE 7.22** [18] Suppose that  $X$  is a discrete random variable, taking the value  $i$  with probability  $p_i$ ,  $i = 1, \dots, k$ . Show that the expected number of comparisons in a simple table-lookup algorithm for generating  $X$  is  $E(X)$ .

**EXERCISE 7.23** [21] Consider an acceptance/rejection algorithm for generating  $X$  that uses  $cg(y)$  as the majorizing function, where  $g(y)$  is the density function for the provisional value  $Y$ . Show that the probability of accepting  $Y$  on the first attempt is  $1/c$ .

**EXERCISE 7.24** [19] Let  $\delta$  be a fixed constant in  $(-1, 1)$ . Construct an algorithm that generates random numbers with density  $f(x) = (1 + \delta x)/2$ , for  $x \in (-1, 1)$ .

**EXERCISE 7.25** [23] (Continuation) Implement the algorithm from the preceding exercise. Obtain a histogram calculated from 10,000 variates generated by your algorithm, and compare the result to the theoretical distribution.

**EXERCISE 7.26** [23] Suppose that `rnd()` generates a stream of uniform random numbers. Consider the following pseudocode:

```
repeat {
  v1 = -log( 1-rnd() )
  v2 = -log( 1-rnd() )
} until ( 2*v1 < (v2-1)^2 )
if (rnd() > 0.5) return v2 else return -v2
```

Prove or disprove the assertion that the code produces standard normal random deviates.

**EXERCISE 7.27** [36] Let  $Z \sim \mathcal{N}(0, \sigma^2)$ , and set  $V = Z \bmod 1$ . Investigate the claim that  $V$  has a uniform distribution. How good is the claim when  $\sigma = 1$ ? [Note that  $-1.234 \equiv 0.766 \pmod{1}$ , *not* 0.234.]

**EXERCISE 7.28** [23] (Gentle 2003, exercise 4.2) Prove that Algorithm 7.4.1 (= Gentle's Algorithm 4.6) produces output with the claimed distribution.

**EXERCISE 7.29** [26] (Gentle 2003, exercise 4.8) Prove that Knuth's method for sampling from a nearly linear density (Algorithm 7.4.3) produces output with the claimed distribution. [Hint: What is the distribution of  $(v-u)/(1-u)$  in Gentle's Algorithm 4.7?]

**EXERCISE 7.30** [16] Prove that if  $U$  is a uniform random variable on  $(0, 1)$ , then  $\lceil \ln U / \ln(1 - \theta) \rceil$  has a geometric distribution with parameter  $\theta$ .

**EXERCISE 7.31** [30] A geometric random number can be generated in one

step using the direct inversion of the cumulative distribution function studied in the preceding exercise. An alternative is to use sequential search or direct simulation. Although more steps are required, each step may be less costly to calculate. Conduct an investigation (empirical or theoretical) to determine the values of  $\theta$  for which direct inversion is more efficient than the other methods.

**EXERCISE 7.32** [15] Prove that Algorithm 7.4.2 generates a random variate from the upper tail of a standard normal distribution.

**EXERCISE 7.33** [21] (Continuation) Derive an expression for  $P(a)$ , the probability of acceptance in Algorithm 7.4.2, and evaluate this expression numerically at  $a = 3$  and  $a = 6$

**EXERCISE 7.34** [25] (Continuation) Show that  $\lim_{a \rightarrow \infty} P(a) = 1$ .

## 7.5 Multivariate random numbers

## 7.6 Other random objects

## 7.7 Assessing quality of random number algorithms

## 7.8 Computing strategies

How should one use the resources in this book? The idea is to collect basic techniques, for two reasons. First, if one is using programs written by others, it makes it possible to see what is “under the hood” and to understand the potential strengths and weaknesses of the approach(es) built in to the software one is using. When that software gives you choices of what to use, for instance, a choice among random number generators or a choice of methods for initializing the state of an rng, the material here can be used to help you make an informed decision. The second reason is to provide general methods that will enable you to create your own software when high-quality public software is not available or not appropriate.

When approaching a problem in computational statistics, you should first look to standard resources. Statistical packages such as **Stata** or **R** provide their users with easily programmed access to high-quality numerical procedures. For instance, generating a million observations from a 5 percent contaminated normal distribution (see page 17) and then looking at the empirical percentiles from this distribution can be generated in **Stata** with just the following lines of code:

```
set obs 1000000
generate x = invnorm(uniform())
replace x=3*x if uniform() $<$ 0.05
summarize x, detail
```

Such packages have facilities for doing simulations, and they have the bonus that the resulting datasets of simulated results can be readily analyzed, graphed, and reported using the analytic capabilities of the package. The downside to this approach is that complex simulations may be too difficult to program or

may require too much computing time. That will often be the case for detailed analysis of new statistical methods, but almost all of the time a good deal of preliminary testing work can be carried out quickly using off-the-shelf software. It is almost always a good idea to test out basic ideas and simple cases first using tools that make it easy to revise quickly and to verify that the results are correct.

It is not unusual for the standard package approach to be adequate for small-scale simulations, but not for larger ones, either in terms of the number of replications needed for adequate precision or the amount of computation needed for each step. In these cases, one must resort to writing a computer program directly in a language such as C++ or C. In this case, you should try to find high-quality numerical libraries that implement the algorithms you want to use.

One such source is the GNU Scientific Library (GSL), an open-source numerical library of C and C++ functions and utilities. The GSL can be found at <http://www.gnu.org/software/gsl/>. The GSL provides access to a number of high-quality random-number generation algorithms. A feature of the library is that it is very easy to exchange one algorithm for another without altering any of the other code. Thus, it is very easy to replicate an entire simulation using a different generating mechanism for the underlying random number sequences.

The GSL documentation pages also contain an informative and succinct article about uniform random number generation algorithms, together with references.



# Monte Carlo Methods

---

**Notes on the revision.** The bootstrap obviously fits into chapter 8. But where do cross-validation and sample re-use fit?

Sections

Design of Simulation Studies

Discrete Event Simulation

Markov Chain Monte Carlo

The Gibbs Sampler

This chapter deals with the design, implementation, and analysis of experiments that rely on simulation. Every simulation involves constructing a (simplified) model of a (generally more complex) system.

## 8.1 Design of simulation studies

### 8.1.1 The simulation space

The accuracy and completeness of the representation used in the simulation can be examined in terms of the idea of the “simulation space” that corresponds to the problem being studied. Loosely speaking, the simulation space is a  $k$ -tuple whose components together describe the scope of the simulation experiment relative to all possible simulation experiments that could be done for a problem.

Let’s consider a simple simulation to investigate the bias and variance of a particular class of estimators of a univariate normal mean. The statistical problem can be set up like this:

$$X_1, X_2, \dots, X_n \sim N(\mu, \sigma^2) \quad (8.1.1)$$

$$\bar{X} = \frac{1}{n} \sum_{i=1}^n X_i, \quad (8.1.2)$$

$$\hat{\mu} = a\bar{X} \quad (8.1.3)$$

A simple simulation study to investigate the squared bias of estimator (8.1.3) is described in Algorithm (8.1.1). Of course, this problem can be solved without recourse to simulation.

---

**Algorithm 8.1.1** Monte Carlo investigation of  $a\bar{X}$ 


---

```

 $\sigma \leftarrow 1$ 
for  $\mu$  in  $\mathcal{M}$  do {Examine results for several choices of  $\mu$ }
  for  $n$  in  $\mathcal{N}$  do {Examine various sample sizes}
    for  $r \leftarrow 1$  to  $R$  do {Replicate each  $(\mu, n)$  setting  $R$  times}
      draw  $X_1, X_2, \dots, X_n$  from  $N(\mu, \sigma^2)$ 
       $\hat{\mu}_r \leftarrow a(\sum_1^n X_i/n)$ 
    end for
     $\hat{B}^2(\mu, \sigma, n) \leftarrow \frac{1}{R} \sum_{r=1}^R (\hat{\mu}_r - \mu)^2$ 
    report  $\mu, n, \hat{B}^2(\mu, \sigma, n)$ 
  end for
end for

```

---

The simulation space here consists of  $\mathcal{R} \otimes \mathcal{R}^+ \otimes \mathcal{Z}^+ \otimes \mathcal{Z}^+$ , corresponding to the

1. the possible choices for  $\mu$  ( $\mathcal{R}$ ),
2. the possible choices for  $\sigma$  ( $\mathcal{R}^+$ ),
3. the possible choices for  $n$  ( $\{1, 2, \dots\}$ ), and
4. the possible choices for  $R$  ( $\{1, 2, \dots\}$ ).

If we were considering the same estimator in a broader context, we could extend the simulation space to include the possible choices for error distribution, or perhaps to allow other dependence structures than i.i.d. observations.

A particular simulation can do well or badly in terms of “covering” the simulation space. Algorithm (8.1.1) actually covers the subset  $\mathcal{M} \otimes \{1\} \otimes \mathcal{N} \otimes \{R\}$ . We are interested in understanding  $B^2(\mu, \sigma, n)$  for any choice of  $\mu, \sigma$ , and  $n$ . How well does this algorithm do?

At first glance, the algorithm studies the bias of  $\hat{\mu}$  for only one choice of  $\sigma$ . However, a quick calculation shows that

$$B^2(\mu, \sigma, n) = \sigma^2 B^2\left(\frac{\mu}{\sigma}, 1, n\right). \quad (8.1.4)$$

This means that we can learn about the squared bias in  $\hat{\mu}$  by combining results of the simulation study with equation (8.1.4).

### 8.1.2 Smooth simulation studies

Simulation studies are often conducted to investigate the sensitivity of an estimator or procedure to various assumptions or to various values of an underlying parameter, and a set of design points are selected that vary these features. All too frequently, the Monte Carlo precision of performance is too great to learn much about the fine structure of the problem. This can be alleviated by introducing positive correlation between the results at different design

points so that comparisons based on differences in performance between design points will have greater precision than would independent comparisons. We call study designs that do this, smooth smooth studies.

As a simple example to illustrate the point, consider a simulation study to calculate the power function of the two-sided  $z$  test based on a single observation from  $N(\mu, 1)$  with  $\alpha = 0.05$ . The power function, of course, is given by

$$\pi(\mu) = 1 + \Phi(\mu + \Phi^{-1}(0.025)) - \Phi(\mu - \Phi^{-1}(0.025)), \quad (8.1.5)$$

which we can readily calculate for comparison purposes.

---

**Algorithm 8.1.2** Conventional simulation of the normal power function

---

```

R ← 100
M ← {0, 1/8, 2/8, ..., 12/8}
for μ in M do {Examine results for selected choices of μ}
  for r ← 1 to R do {Replicate each μ setting R times}
    Xr ← μ + normal(0, 1)
    Pr ← (|Xr| ≥ 1.96)
  end for
   $\hat{\pi}(\mu) \leftarrow \left( \sum_{r=1}^R P_r \right) / R$ 
end for

```

---

A straightforward simulation design is that of Algorithm 8.1.2, which generates  $R$  replicates of  $Z \sim N(\mu, 1)$ , calculates  $\hat{\pi}(\mu)$  as the fraction of these replicates for which  $|Z| > -\Phi^{-1}(0.025)$ , and then repeats this process over a suitably chosen set of values for  $\mu$ . The first panel of Figure 8.1.1 shows the results with  $R = 100$  replications at each value of  $\mu$  from 0 to 1.5 in steps of 0.125. The basic model is thus replicated 1300 times. The true power function given by expression (8.1.5) is represented by the dashed curve. The standard errors range from about 0.024 near  $\mu = 0$  to about 0.046 near  $\mu = 1.5$ .

The estimated power function in Figure 8.1.1(a) is quite irregular. That is, the simulated power at adjacent values of  $\mu$  are often quite different even though the actual power function is very similar. This is a result of the independence of the estimates for each value of  $\mu$ .

The second panel of Figure 8.1.1 tells quite a different story, even though it is based on the same number of function evaluations and generated normal deviates (1300). The correlation of adjacent values in Figure 8.1.1(b) is approximately 0.88. Moreover, the standard errors range from about 0.007 on the left to roughly 0.013 on the right of the figure. So for the same amount of computer work, we have improved our precision at every point, and obtained a smoother version of the curve we are trying to estimate.

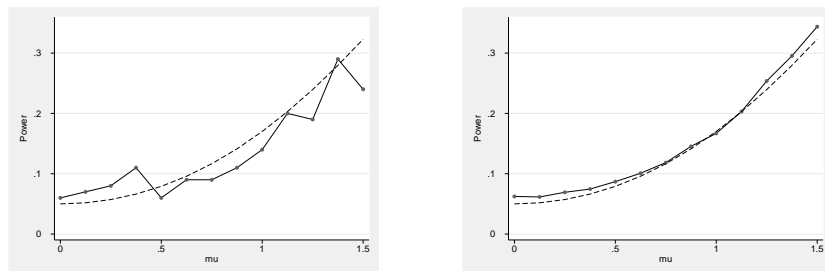


Figure 8.1.1 *Conventional and smoothed simulations of the z-test power function.  $R = 100$  independent replications are generated for each of the 13 design point in the left-hand panel.  $R = 1300$  normal deviates (errors) are generated for the right-hand panel, and the 13 values for  $\mu$  are successively added to the same errors, introducing substantial correlation in adjacent points on the curve.*

---

**Algorithm 8.1.3** Smooth simulation of the normal power function

---

```

 $R \leftarrow 1300$ 
 $\mathcal{M} \leftarrow \{0, \frac{1}{8}, \frac{2}{8}, \dots, \frac{12}{8}\}$ 
for  $r \leftarrow 1$  to  $R$  do {Initialize error vector}
     $\epsilon_r \leftarrow \text{normal}(0, 1)$ 
end for
for  $\mu$  in  $\mathcal{M}$  do {Calculate results for each  $\mu$ }
    for  $r \leftarrow 1$  to  $R$  do {Add each  $\mu$  setting to errors}
         $X_r \leftarrow \mu + \epsilon_r$ 
         $P_r \leftarrow (|X_r| \geq 1.96)$ 
    end for
     $\hat{\pi}(\mu) \leftarrow (\sum_{r=1}^R P_r) / R$ 
end for

```

---

Figure 8.1.1(b) was obtained from the study described in Algorithm 8.1.3. The difference between the two algorithms is small. In the conventional simulation, we generate

$$X_{ir} = \mu_i + \epsilon_{ir},$$

while in the smooth simulation, we generate

$$X_{ir} = \mu_i + \epsilon_r.$$

By using the same errors for all settings of  $\mu$  we simultaneously introduce the desired positive correlation and we reduce the amount of random-number generation by a factor of  $M$ , the number of values of  $\mu$  we examine. As a result, we can increase the number of replications by that same factor and (almost) break even in terms of computational effort.

In general, we can think of each replication as combining a systematically varying component (as we step through the points in the simulation space) and a random component (obtained from simulated random variates). The idea of

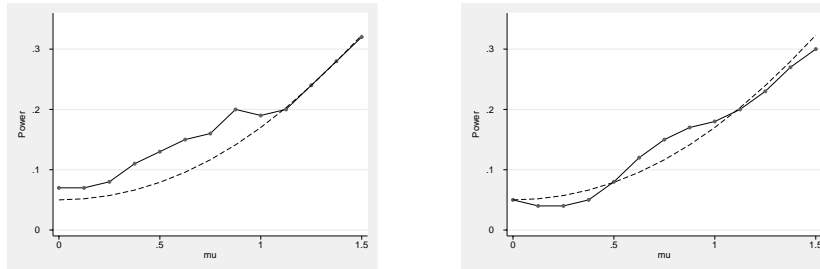


Figure 8.1.2 *Smoothed simulations of the z-test power function, each based on  $R = 100$  replicates of the underlying power curve.*

smooth simulation is for the same generated random components to be used in conjunction with each of the points in the simulation space. When analysis of the simulation data set takes place, we can then condition on the random configurations to remove substantial variability from comparisons across the simulation design space.

If we do not compensate for the reduced number of random deviates by increasing the number of replications in the smooth simulation, the result can be smooth but (consistently) away from the underlying function being estimated. Figure 8.1.2 shows the results from two smooth simulations with only  $R = 100$  replications each.

It is sometimes helpful to think of smooth simulation as generating one entire *curve* estimate for each replicate. These curves are then averaged to produce the final result.

### 8.1.3 Non-convergence

#### Notes on the revision.

Date: Wed, 20 May 1998 15:01:19 -0400  
 From: "Steichen, Thomas" <STEICHT@RJRT.com>  
 Subject: RE: statalist: MC simulation

**\*\* Send unsubscribe or help commands to majordomo@hsphsun2.harvard.edu \*\***

Andrzej Niemierko writes:

```
> I am using Stata's Monte Carlo simulation command "simul" and each
> repetition performs nonlinear least squares fitting using the "nl"
> command on artificially created (according to some binomial model) data
> sets. It works fine but sometimes there is a data set on which the nl
> algorithm cannot converge. The nl stops with error 430 and,
> unfortunately, the whole simulation is aborted with all the previously
> calculated results gone.
>
> Is there a way to drop such rare but unpleasant cases without aborting
```

```

> "simul"?
>
[ TJS, in reply >>> ] Just capture the return code from -nl- then use
an -if / else if / else- structure to act on the return code value,
_rc, as appropriate: if _rc==0 (a successful run), do your
calculations and post as usual; if _rc==430 (no convergence), skip the
calculations and post nothing or missing; if _rc==anything else,
report the error and quit.

capture nl ...
if _rc == 0 {
  do any calculations
  post ...'results' as usual
}
else if _rc == 430 {
  no calculations
  post ...missing (or don't post anything)
}
else {
  di in re "error " _rc " in -nl-"
  exit
}

```

## EXERCISES

**EXERCISE 8.1** [02] What is the bias of estimator (8.1.3)?

**EXERCISE 8.2** [13] Prove equation (8.1.4).

► For exercises 8.3 through 8.10, you should prepare a report describing your study, its objectives, the study design you selected, a summary of the results, and the conclusions that can be drawn from the study. You should include an appendix with your computer programs and representative output.

**EXERCISE 8.3** [34] Conduct a simulation study of Bartlett's test for equality of two variances. If  $s_1^2$  and  $s_2^2$  are sample variances from i.i.d. normal samples of size  $n_1$  and  $n_2$ , with variances  $\sigma_1^2$  and  $\sigma_2^2$ , respectively, then  $F = s_1^2/s_2^2$  will have an  $F$  distribution if  $\sigma_1^2 = \sigma_2^2$ . Investigate the sampling distribution of  $F$  and the size of a nominal 5% test, when the variances are in fact equal, but the distribution from which the samples are drawn varies. You may wish to investigate distributions with varying amounts of asymmetry, or with tail behavior that ranges from lighter to heavier than normal.

**EXERCISE 8.4** [33] Carry out a simulation study to compare two queueing disciplines for a two-server situation. Both situations have customers arriving at a rate of  $\lambda$  per hour, and two servers are always working, each able to service  $\mu$  customers per hour on average. (Note: you may wish to make sure that  $2\mu > \lambda$ .) In case 1, customers get in the line for the server with the fewer persons in line; in case 2, each server serves the customer at the head of a single line. Consider the case of Poisson arrivals and exponential service times.

**EXERCISE 8.5** [24] (Continuation) Modify the simulation study of the previous exercise to allow the servers to be able to serve  $\mu_1$  and  $\mu_2$  clients per hour, respectively. (Here,  $\mu_1 + \mu_2 > \lambda$ .)

**EXERCISE 8.6** [21] Outline how you would design a simulation study to investigate the power of a standard statistical test under nonstandard conditions. For example, consider testing that the mean of a distribution  $F$  is zero using the  $t$  test, when  $F$  has a contaminated normal distribution.

**EXERCISE 8.7** [30] (Continuation) Carry out the simulation study designed in the preceding exercise.

**EXERCISE 8.8** [99] Carry out a simulation study of the power of the  $t$ -test with 5 degrees of freedom over the simulation space  $\mu \in \mathcal{M} = \{0, \frac{1}{8}, \frac{2}{8}, \dots, \frac{12}{8}\}$ , comparing conventional and smooth simulation designs. By how much can the number of replications at each point in simulation space be increased by using a smooth simulation design?

**EXERCISE 8.9** [99] (Continuation) Suppose we wish to enlarge the simulation space for the  $t$ -test power function of the previous exercise by varying the degrees of freedom over the set  $\mathcal{D} = \{1, 2, 5, 10, 30, 60\}$ . Describe how the principles of smooth simulation could be incorporated into such a study.

**EXERCISE 8.10** [99] (Continuation) Carry out the simulation from the preceding exercise using  $\mathcal{M}$  from Algorithm 8.1.3, the six values of  $d$  from the previous exercise, and  $R = 7800$ . Compare your results, including timings, for each  $d$  with those obtained using conventional simulation with  $R = 100$  at each of the 78 points in the simulation space.

## 8.2 The Metropolis-Hastings algorithm

**Notes on the revision.** See the excellent short history by Hitchcock (2003).

### EXERCISES

**EXERCISE 8.11** [26] Consider the “batch means” algorithm described on pages 237–238 of Gentle (2003). Gentle indicates that the “size of subsamples should be as small as possible and still have means that are independent.” Suppose that the  $\{f_i\}$  have MA(1) structure. Obtain a better approximation for the variance of  $\bar{F}$  than the one given in Gentle’s expression (7.9). [A sequence of random variables  $X_i$  has MA( $q$ ) structure if  $X_i = \mu_i + e_i$ , and  $e_i = a_i + \sum_{j=1}^q \beta_j a_{i-j}$ , where the  $\{a_k\}$  are mean zero and uncorrelated random variables.]

**EXERCISE 8.12** [99] (Continuation) Repeat the previous exercise, assuming that the  $\{f_i\}$  have AR( $p$ ) structure. [A sequence of random variables  $X_i$  has AR( $p$ ) structure if  $X_i = \alpha + \sum_{j=1}^p X_{i-j} + \epsilon_i$ , where the  $\{\epsilon_i\}$  are uncorrelated random variables with zero mean.] Consider more general structures.



---

## Answers to Exercises

---

**7.1** Since  $P$  is increasing and continuous, it is a one-to-one function, so that  $P^{-1}$  is unique. Thus, we can write  $\Pr[X < t] = \Pr[P^{-1}(U) < t] = \Pr[U < P(t)] = P(t)$ .

**7.2** Absolute continuity does not guarantee uniqueness of  $P^{-1}(u)$ , but any choice of definitions (such as the  $\liminf\{x : P(x) > u\}$ ) will do, since absolute continuity guarantees that the probability contained in all sets for which  $P(x) = u$  for some  $x$  is zero.

**7.3** Return  $\lfloor 6u() \rfloor + 1$ .

**7.4** If  $\lambda$  is very small relative to  $m$ , it is clear that the distribution will be essentially the same as the Poisson distribution truncated at  $m - 1$ , which is decidedly nonuniform. For example, with  $m = 2$  (generation of random bits), we have

$$\Pr(X_i = 0) = \sum_{j=0}^{\infty} \frac{e^{-\lambda} \lambda^{2j}}{(2j)!} = e^{-\lambda} \cosh(\lambda) = \frac{1}{2}(1 + e^{-2\lambda}).$$

As  $\lambda \rightarrow \infty$ , this goes to 0.5, but for  $\lambda = 1, 2$ , and 3 we have probabilities of 0.56767, 0.50916, and 0.50124, respectively. In general,  $\Pr(X_i = j) \rightarrow (1/m)$  as  $\lambda \rightarrow \infty$ .

**7.6** The states for  $u()$  and  $j()$  can take on at most  $m_1 m_2$  different values. For any one of these values, the order in which the  $A[\cdot]$  array is filled is completely determined by the  $j()$  sequence. Once a state value previously encountered occurs for the second time, the reservoir will be filled by the same values, in the same order, as the previous time through the sequence. Thus, the counting problem comes down to enumerating how many different possibilities there are for the contents of the  $A[\cdot]$  array.

**7.7** Bays and Durham (1976) analyze an *ideal* generator that requires (truly) uniform random integers to obtain an estimate of the period of  $c\sqrt{mk!}$ , where  $m$  is the period of the linear congruential generator used for both production and selection.

**7.8** Note that after some time, the  $A$  array will be sorted in ascending order.

**7.9** Your solution should contain both the code for your function, as well as a program that initializes your function and calls it 100 times, and the computer output that results. There are *many* ways to write such a function that appear “obvious,” but which fail to produce the correct sequence. Park and Miller (1988) describe implementation approaches. The GNU Scientific Library contains an implementation of this generator in the `minstd` subroutine.

**7.10** This problem is more difficult than the previous one, because the larger seed value makes certain methods that work for implementing the Lewis-Goodman-Miller generator (such as using 64-bit integer multiplication) fail to work here. For checking purposes,  $x_{100} = 160159497$  when  $x_0 = 1$ .

**7.11** We begin with a lemma: if  $U$  and  $V$  are iid  $U[0,1)$ , then so is  $\{U + V\}$ . The distribution function of  $\{U + V\}$  is  $P(\{U + V\} < t) = EI_{\{U+V\} < t}$ . Now the set

of points for which the fractional part of  $U + V < t$  is the union of three sets:  $\{U < t, V \leq t - U\}$ ,  $\{U < t, V \geq 1 - U\}$ , and  $\{U \geq t, 1 - U \leq V < 1 - U + t\}$ . The probabilities of these three sets are, respectively,  $t^2/2$ ,  $t^2/2$  and  $t - t^2$ . Their sum is then  $t$ , which is the cumulative distribution function of the uniform distribution on  $(0, 1)$ . An inductive proof is now straightforward. The result clearly holds for  $n = 1$ . Suppose it holds for  $n \leq k$ . Then  $Z_{k+1} = \{Z_k + U_{k+1}\}$ , and the first of the two summands has a uniform distribution by the induction hypothesis. The conditions of the lemma are satisfied, giving the result.

**7.13** The Chinese Remainder Theorem can be used to show that the Wichmann-Hill generator is equivalent to the pure linear congruential generator with  $m = 27817185604309$  and  $a = 16555425264690$ . In particular,  $X_i^{(j)} = X_i \bmod m_j$ .

In *Mathematica*, this can be calculated by loading the `NumberTheory` package and then calculating  $a$  as `ChineseRemainder[{171, 172, 170}, {30269, 30307, 30323}]`. The value for  $m$  is the product of the three moduli. One can also verify that the moduli are mutually prime.

**7.16** When using cdf inversion methods for generating a nonuniform random number,  $F^{-1}(0)$  may correspond, through a limiting argument, to the (unrealizable) variate value of  $-\infty$ ; similarly,  $F^{-1}(1)$  may equal  $+\infty$ .

**7.17** Note that  $U_{i-1} + U_{i-2} = k + U_i$ , where  $k = 0$  or  $1$ . Suppose it were the case that  $U_{i-1} > U_i > U_{i-2}$ . This would imply that  $U_{i-1} > U_{i-1} + U_{i-2} - k > U_{i-2}$ . The first of these inequalities implies that  $k > U_{i-2}$ , so that  $k$  would have to equal  $1$ . But the second inequality implies that  $k = 0$ , so that both inequalities cannot hold simultaneously.

**7.19** Exact methods such as Marsaglia's version of the polar method are much faster than this approximation, typically by factors of 3–5. More importantly, the distribution of the sum of 12 uniforms places too much mass near the center of the distribution, and far too little in the tails. This behavior is especially noticeable beyond about 2.4 standard deviations from the mean.

**7.20** We demonstrated that  $U^{1/n}$  has the same distribution as the maximum of  $n$  uniforms. Thus,  $1 - U^{1/n}$  must have the same distribution as the minimum. (The same result is obtained using the inverse cdf transformation coupled with the observation that  $U$  and  $1 - U$  have the same distribution.)

**7.21** Since  $-\lambda \log(U)$  has the same distribution as a single exponential variate, and since this is a monotone decreasing function in  $U$ , substituting the minimum of  $k$  uniforms in the preceding expression would correspond to the maximum of the corresponding exponential variates that those uniforms would generate. Thus  $-\lambda \log(1 - U^{1/k})$  does the trick.

**7.22** The probability of reaching the  $i$ th value in the table, which requires exactly  $i$  comparisons, and stopping there is  $p_i$ . So the expected number of lookups is  $\sum i \cdot p_i = E(X)$ .

**7.27** If  $\sigma$  is small, say about  $1/3$  or less, most of the mass of  $Z$  will lie between  $-1$  and  $+1$ , so that we can essentially ignore the mass outside this region. This makes it easy to see that the density of  $V$  will be U-shaped and not uniform in this case, and it is clear that the claim cannot hold in general.

Nonetheless, the claim is remarkably good for  $\sigma \geq 1$  — so good, in fact, that investigation of the empirical distribution of  $V$  using simulation could never reveal the discrepancy.

---

## References

---

- C. Bays. Improving a random number generator: A comparison between two shuffling methods. *Journal of Statistical Computation and Simulation*, 36: 57–59, 1990.
- Carter Bays and S. D. Durham. Improving a poor random number generator. *ACM Transactions on Mathematical Software*, 2:59–64, 1976.
- Carter Bays and W. E. Sharp. Generating random numbers in the field. *Mathematical Geology*, 23:541–548, 1991.
- Robert R. Coveyou and Robert D. MacPherson. Fourier analysis of uniform random number generators. *J. ACM*, 14:100–119, 1967.
- J. S. Dagpunar. A compact and portable Poisson random variate generator. *Journal of Applied Statistics*, 16:391–393, 1989.
- Lih-Yuan Deng and Dennis K. J. Lin. Random number generation for the new century. *The American Statistician*, 54(2):145–150, 2000.
- L. Devroye. *Non-uniform random variate generation*. Springer-Verlag Inc, 1986.
- George S. Fishman and III Moore, Louis R. An exhaustive analysis of multiplicative congruential random number generators with modulus  $2^{31} - 1$  (Corr: V7 p1058). *SIAM Journal on Scientific and Statistical Computing*, 7:24–45, 1986.
- James E. Gentle. *Portability considerations for random number generators*. Computer Science and Statistics: Proceedings of the 13th Symposium on the Interface, 1981.
- James E. Gentle. *Random number generation and Monte Carlo methods*. Springer-Verlag Inc, second edition, 2003.
- E. R. Golder. [Algorithm AS 98] the spectral test for the evaluation of congruential pseudo-random generators. *Applied Statistics*, 25:173–180, 1976.
- P. Griffiths and I. D. Hill. *Applied Statistics Algorithms*. Ellis Horwood Limited, Chichester, 1985.
- D. B. Hitchcock. A history of the Metropolis-Hastings algorithm. *The American Statistician*, 57(4):254–257, 2003.
- T. R. Hopkins. [Algorithm AS 193] a revised algorithm for the spectral test. *Applied Statistics*, 32:328–335, 1983.

- W. Hörmann and G. Derflinger. A portable random number generator well suited for the rejection method. *ACM Transactions on Mathematical Software*, 19:489–495, 1993.
- Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading, Massachusetts, third edition edition, 1998.
- Pierre L’Ecuyer. Efficient and portable combined random number generators. *Communications of the Association for Computing Machinery*, 31:742–749, 1988.
- Pierre L’Ecuyer. Combined multiple recursive random number generators. *Operations Research*, 44(5):816–822, 1996.
- Pierre L’Ecuyer, François Blouin, and Raymond Couture. A search for good multiple recursive random number generators. *ACM Transactions on Modeling and Computer Simulation*, 3:87–98, 1993.
- Peter A. W. Lewis, A. S. Goodman, and J. M. Miller. A pseudo-random number generator for the System/360. *IBM Systems Journal*, 8(2):136–146, 1969.
- M. Donald MacLaren and George Marsaglia. Uniform random number generators. *JACM*, 12(1):83–89, 1965.
- George Marsaglia. Random numbers fall mainly in the planes. *Proceedings of the National Academy of Sciences (US)*, 60:25–28, 1968.
- Makoto Matsumoto and Yoshiharu Kurita. Twisted GFSR generators. *ACM Transactions on Modeling and Computer Simulation*, 2:179–194, 1992.
- Makoto Matsumoto and Yoshiharu Kurita. Twisted GFSR generators, ii. *ACM Transactions on Modeling and Computer Simulation*, 4:254–266, 1994.
- Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.
- A. Ian McLeod. [AS R58] A remark on Algorithm AS 183: “An efficient and portable pseudo-random number generator”. *Applied Statistics*, 34:198–200, 1985.
- Stephen K. Park and Keith W. Miller. Random number generators: Good ones are hard to find. *Communications of the Association for Computing Machinery*, 31:1192–1201, 1988.
- RAND Corporation. *A Million Random Digits with 100,000 Normal Deviates*. Free Press, New York, 1955.
- Christian P. Robert and George Casella. *Monte Carlo Statistical Methods*. Springer Texts in Statistics. Springer, New York, 1999.
- Linus Schrage. A more portable FORTRAN random number generator. *ACM Transactions on Mathematical Software*, 5:132–138, 1979.
- W. E. Sharp and C. Bays. A review of portable random number generators. *Computers & Geosciences*, 18:79–87, 1992.

- W. E. Sharp and C. Bays. A portable random number generator for single-precision floating-point arithmetic. *Computers & Geosciences*, 19:593–600, 1993.
- R. C. Tausworthe. Random numbers generated by linear recurrence modulo two. *Mathematics of Computation*, 19:201–209, 1965.
- Shu Tezuka and Pierre L’Ecuyer. Efficient and portable combined Tausworthe random number generators. *ACM Transactions on Modeling and Computer Simulation*, 1:99–112, 1991.
- B. A. Wichmann and I. D. Hill. [Algorithm AS 183] An efficient and portable pseudo-random number generator. *Applied Statistics*, 31:188–190 (Correction, 33:123, 1984), 1982.
- B. A. Wichmann and I. D. Hill. Algorithm AS 183. An efficient and portable pseudo-random number generator. In *Applied Statistics Algorithms* Griffiths and Hill (1985), pages 238–242.
- Peter C. Wollan. A portable random number generator for parallel computers. *Communications in Statistics, Part B – Simulation and Computation*, 21:1247–1254, 1992.
- H. Zeisel. [AS R61] A remark on Algorithm AS 183: “An efficient and portable pseudo-random number generator”. *Applied Statistics*, 35:89–89, 1986.
- Robert M. Ziff. Four-tap shift-register-sequence random-number generators. *Computers in Physics*, 12(4):385–392, July/August 1998.



---

# Index

---

- Bays and Durham, 7, 31, 33
- Bays and Sharp, 7, 33
- Bays, 7, 33
- Coveyou and MacPherson, 13, 33
- Dagpunar, 9, 33
- Deng and Lin, 9, 33
- Devroye, 16, 33
- Fishman and Moore, 14, 33
- Gentle, 9, 19, 29, 33
- Golder, 13, 33
- Griffiths and Hill, 33, 35
- Hitchcock, 29, 33
- Hopkins, 13, 33
- Hörmann and Derflinger, 9, 33
- Knuth, ix, 8–10, 13, 34
- L'Ecuyer et al., 10, 34
- L'Ecuyer, 8–10, 34
- Lewis et al., 6, 34
- MacLaren and Marsaglia, ix, 7, 8, 34
- Marsaglia, 5, 34
- Matsumoto and Kurita, 12, 34
- Matsumoto and Nishimura, 12, 34
- McLeod, 9, 34
- Park and Miller, 14, 31, 34
- RAND Corporation, 3, 34
- Robert and Casella, 11, 34
- Schrage, 6, 34
- Sharp and Bays, 9, 34
- Tausworthe, 10, 35
- Tezuka and L'Ecuyer, 9, 35
- Wichmann and Hill, 9, 14, 35
- Wollan, 9, 35
- Zeisel, 9, 14, 35
- Ziff, 11, 35
  
- Acceptance/rejection
  - algorithm, 16
  
- C (computer language), 11, 12, 21
  
- C++ (computer language), 21
- cdf inversion, *see* probability integral transform
- computer programs
  - DIEHARD, 13
  - gfsr4, 11
  - KISS, 11
  - knuthran2, 10
  - minstd, 31
  - mt19937, 12
  - MWC1038, 12
  - RANDU, 5–7, 18
  
- discrepancy, 5, 13
  
- feedback shift register generator, 10
- Fibonacci generator, 10, 15
  - generalized, 10, 15
- FORTRAN (computer language), 5, 6, 9, 18
  
- generalized feedback shift register, 10
- GNU Scientific Library, 10–12, 21, 31
- GSL, *see* GNU Scientific Library
  
- inversion of cdf, *see* probability integral transform
  
- Knuth's method, 19
  - algorithm, 17
  
- linear congruential generator, 4–7
  - lattice structure of, 5
  - Lewis-Goodman-Miller, 6, 14
  - Mersenne, 6
  - minimal standard, 14
  - multiple, 10
  - periodicity of, 4

- portable, 14
- RANDU, 5
- Wichmann-Hill, 9, 14
  
- Marsaglia, George, 11–12
- Mathematica (software package), 32
- Mersenne Twister, 12
- mixing uniform generators, 8–9
- multiple recursive generator, 10
  
- nearly linear density
  - sampling from, 17
- nonuniform random numbers
  - binomial, 17
  - contaminated normal, 17
  - geometric, 18, 19
  - maximum of exponentials, 19
  - minimum of uniforms, 18
  - normal, 19
  - Poisson, 18
  - triangular, 19
- Normal distribution
  - sampling from the tail of, 16
  
- permutations
  - random, 15
- PL/I (computer language), 5
- probability integral transform, 2
- pseudorandom numbers, 3
  
- R (computer language), 12, 20
- Rand tables, 3
- random deviates, 1
- random numbers
  - defined, 1
  - Rand tables, 3
- random variates, 1
- RANDU, 5–7, 18
  
- sampling finite set
  - with replacement, 2
- Scientific Subroutine Package, 5, 18
- Shuffling
  - uniform generators, 7–8
- simulation space, 23–24
- smooth smooth studies, 25
- spectral test, 13