3 Pseudorandom Number Generators

Monte Carlo needs numbers quite random
but practicality has mostly banned them.
So in order to score
we need do no more
than generate them pseudo*random!*

At the core of all Monte Carlo calculations is some mechanism to produce a long sequence of random numbers ρ_i that are uniformly distributed over the open interval (0,1). However, digital computers, by design, are incapable of producing random results. A true random sequence could, in principle, be obtained by coupling to our computer some external device that would produce a truly random signal. For example, we could use the time interval between two successive clicks of a Geiger counter placed near a radioactive source or use the "white noise" in some electronic circuit to provide truly random numbers.

However, use of such a random number generator would be disastrous! First, the feasibility of having to couple some external device that generates a random signal to every computer we want to use for Monte Carlo calculations would be impractical. But more important would be the impossibility of writing and debugging a Monte Carlo code if, on every run, a different sequence of random numbers were used. What is needed is a sequence of random numbers that is the same every time the program is run so that code errors can be found and so that the same results are produced when the same code is run on different computers.

One way to use true random numbers that would have the same sequence every time a code is run would be to create somehow a table of random numbers and then read them into the computer as they are needed. In fact, Marsaglia [1995] has produced a CD-ROM that contains 4.8×10^9 truly random numbers (generated by combining two pseudorandom number generators and three devices that use random physical processes). However, this is not a good idea. First, in modern Monte Carlo applications many more random numbers are needed, and second, and more importantly, the time to read a number from an external file is prohibitively long for modern Monte Carlo calculations.

An alternative to producing the same sequence of random numbers each time a program is run is to use a *pseudorandom* number generator. Such a generator is a deterministic algorithm that, given the previous numbers (usually just the last number) in the sequence, the next number can be efficiently calculated. Many pseudorandom number generators, some good but most poor, have been proposed and used over the years in a wide variety of Monte Carlo work. Designing better random number generators (the term "pseudo" is sometimes dropped from now on) is still an active

Exploring Monte Carlo Methods. DOI: 10.1016/B978-0-444-51575-9.00003-8

area of research. In this chapter, the basic properties of some important random number generators are discussed and several important random number generators are presented.

The term *quasirandom* numbers is sometimes encountered. These numbers are designed not so much to be random as to be extremely uniform over the interval (0, 1) so as to allow very accurate integration by the Monte Carlo method. Although the generation of quasirandom numbers is not considered further here, the application of such numbers is considered in Section 5.11.2.

Although not reviewed in this chapter, there have been many highly deficient random number generators used in many studies. Some still continue to be used by researchers unaware of the biases being introduced into their calculations. Indeed, if all publications based on these questionable generators were to disappear, much valuable library shelf space would be recovered. When you embark on Monte Carlo investigations, be well aware of the pedigree of the random number generator you are using. Even more important, don't alter your random number generator unless you are *very* confident your changes are an improvement. The construction of random number generators is best left to the experts.

3.1 Linear Congruential Generators

Perhaps the most widely used random number generators are those based on the *linear congruential generator* proposed by [Lehmer 1951], which has the general form[1]

$$\boxed{x_{i+1} = (ax_i + c) \bmod m, \quad i \geq 0.} \tag{3.1}$$

Here, integer $a > 1$ is called the *multiplier*, integer c the *increment*, and integer m the *modulus* of the generator. Starting with some initial integer *seed* x_0, the sequence of integers x_i will lie in the range $0 \leq x_i < m$. Each x_i is then scaled to the interval [0, 1) by dividing by m, i.e.,

$$\rho_i = \frac{x_i}{m}, \tag{3.2}$$

to obtain random numbers over the interval [0, 1).

In the most frequently programmed congruential generator, the increment c is set to zero and Eq. (3.1) becomes the so-called *multiplicative congruential generator*

$$\boxed{x_{i+1} = ax_i \bmod m, \quad i \geq 0.} \tag{3.3}$$

[1] The notation mod m means that the value of $ax_i + c$ is divided by m and the *remainder* resulting from the division is then kept as x_{i+1}. Two integers a and b are said to be *congruent modulo m* if $a \bmod m = b \bmod m$. Hence, the name of this random number generator. The actual generator Lehmer used was $x_{i+1} = 23x_i \bmod (10^8 + 1)$, which, while better than some successors, is not very good by today's standards.

Because of the modular arithmetic the values of the integers x_i cannot exceed $m - 1$ and, for a nonzero initial seed, x_i can never equal 0. Thus, the $\rho_i = x_i/m$ must lie in the open interval (0, 1). For many Monte Carlo simulations it is important that the random number generator never returns a ρ_i that is exactly 1 or 0.

Just how many distinct integer values $1 \le x_i \le m - 1$ are obtained depends very critically on the relation between a and m. This is the crux of a good generator: pick a and m such that a vast number of different values x_i are obtained and that they and all pairs, triplets, quadruplets, and so on (i.e., all order k-tuples) "appear" to be distributed randomly.

If $k_i = [ax_{i-1}/m]$,[2] then Eq. (3.3) can be written in the alternate fashion:

$$\begin{aligned}
x_1 &= ax_0 - k_1 m \\
x_2 &= a^2 x_0 - k_2 m - ak_1 m \\
x_3 &= a^3 x_0 - k_3 m - ak_2 m - a^2 k_1 m \\
\vdots \quad & \quad \vdots \\
x_j &= a^j x_0 - m(k_j + ak_{j-1} + a^2 k_{j-2} + \cdots + a^{j-1} k_1).
\end{aligned}$$

Because $1 \le x_i < m,\ 1 \le i \le j$, the integer $(k_j + ak_{j-1} + \cdots + a^{j-1}k_1)$ must be the largest integer in $a^j x_0/m$, i.e., it must equal $[a^j x_0/m]$. Hence, an alternative representation of the generator of Eq. (3.3) is

$$x_j = a^j x_0 \bmod m, \quad j \ge 1. \tag{3.4}$$

The initial seed x_0 must not equal 0 or m because these generate a sequence of zeros. Because there are at most $m - 1$ integers x_i in $0 < x_i < m$, the maximum *period* or *cycle length* of the multiplicative consequential generator is $m - 1$. However, for a given m and a, the period may be much less than $m - 1$. Clearly, for a practical random number generator the modulus m must be very large and the multiplier a must be picked to produce a very long (if not full) period.

3.2 Structure of the Generated Random Numbers

All of the $\{\rho_i\}$ produced by a full-period generator are clearly uniformly distributed over the interval (0, 1). However, a useful generator must produce *subsamples* that must also be distributed uniformly over (0, 1). Moreover, there should be no correlation between successive numbers, i.e., each number should appear to be independent of the preceding number. Although such correlations can be minimized with proper choices for a and m, they are never completely eliminated. For a poor choice of a and m, the correlations become obvious.

[2] The notation $[x/y]$, where x and y are integers, means that only the integer portion of the quotient x/y is used, i.e., any fractional part is discarded.

For example, consider a full-period generator with $a = 2$ and a very large m. At some point in the cycle x_i will be returned as 1 (or some other small integer). The subsequent values of $x_{i+1}, x_{i+2}, \ldots$ will then also be small integers. If $x_i = 1$, the subsequent sequence is $1, 2, 4, 8, 16, \ldots$. Clearly, this subsample is far from random.

Example 3.1 A Simple Multiplicative Congruential Generator

Consider the sequences produced by the simple generator $x_{i+1} = ax_i \bmod 13$ for $(1 \leq a < 13)$. All are started with $x_0 = 1$. The following results are obtained:

	x_i												
a	***i* = 0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
1	1	1											
2	1	2	4	8	3	6	12	11	9	5	10	7	1
3	1	3	9	1									
4	1	4	3	12	9	10	1						
5	1	5	12	8	1								
6	1	6	10	8	9	2	12	7	3	5	4	11	1
7	1	7	10	5	9	11	12	6	3	8	4	2	1
8	1	8	12	5	1								
9	1	9	3	1									
10	1	10	9	12	3	4	1						
11	1	11	4	5	3	7	12	2	9	8	10	6	1
12	1	12	1										

Notice $a = 1$ produces only a sequence of ones, and hence a must always be > 1. Only four multipliers—namely, $a = 2, 6, 7, 11$—produce full periods. For other multipliers, cycles of lengths 6, 4, 3, and 2 are realized. Generally, the cycle length of the nonfull-period sequences depends on the initial seed. Note also that, except for $a = 12$, the nontrivial sequences occur in pairs, one being the reverse of the other.

As observed by Marsaglia [1968] the sequence of values x_i produced by any linear congruential generator has an extremely rigid structure. If k random numbers at a time (k-tuples) are used as coordinates of points in k-space, the points do not fill the space randomly, but rather they lie on a finite number—and possibly a small number—of parallel hyperplanes. There are at most $m^{1/k}$ such planes. Further, these points form a simple k-dimensional lattice.[3]

As a simple example, consider the generator

$$x_{i+1} = ax_i \bmod 29 \qquad \text{with} \qquad x_0 = 1$$

[3] A k-dimensional lattice is defined by k independent vectors $\mathbf{v}_j$ such that any point on the hyperplanes (lattice) is given by $\sum_{j=1}^{k} n_j \mathbf{v}_j$, where n_j are integers.

and plot the points (x_0, x_1), (x_1, x_2), (x_2, x_3), ... In this two-dimensional plot, the hyperplanes become straight lines. For this generator, $a = 3, 8, 10, 11, 14, 15, 18, 19, 21, 26$, and 27 produce full-period generators. However, the $m - 1 = 28$ points lie on different numbers of lines: two lines for $a = 14, 15$, and 27; three lines for $a = 3, 10, 19$, and 26; and six lines for $m = 8, 11, 18$, and 21. Two examples of the resulting two-dimensional plots are shown in Fig. 3.1.

The left plot uses the generator $x_{n+1} = 3x_n \bmod 29$ to produce the sequence

1 3 9 27 23 11 4 12 7 21 5 15 16 19 28 26 20 2 6 18 25 17 22 8 24 14 13 10 1 ....

The numbering of the points refers to the number pairs, i.e., 1 is the first pair (1, 3), 2 is the second pair (3, 9), and so on. This plot is somewhat alarming. All points in the lattice fall on just three lines with slope 3, and there are also ten lines of slope $-2/9$. The distances between adjacent points on these two sets of lines is very different (3.16 versus 9.22), and the full "volume" (area) of the square is far from being uniformly covered by the points, a property we would expect if we use a good random number generator.

The right plot in Fig. 3.1 uses the generator $x_{n+1} = 8x_n \bmod 29$ to produce the sequence

1 8 6 19 7 27 13 17 20 15 4 3 24 18 28 21 23 10 22 2 16 12 9 14 25 26 5 11 1 ....

This is a much more reassuring picture. The points are more evenly distributed over the plot area and lie along six lines of slope 3/4 or along seven lines of slope $-5/3$. Moreover, the distances between adjacent points of these two sets of lines are comparable (5.00 versus 5.83), indicating a nearly equidistribution of points over the area of the square.

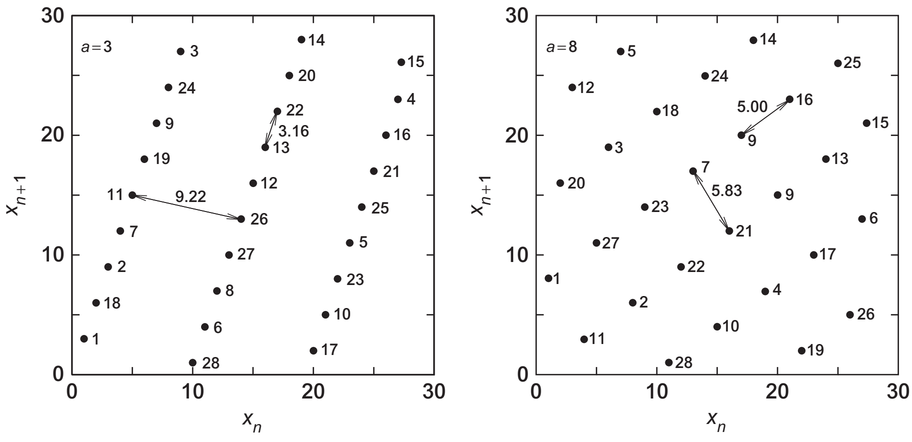


Figure 3.1 Two plots of successive numbers from $x_{n+1} = ax_n \bmod 29$. The left plot has a multiplier $a = 3$ and the right plot $a = 8$. Both are full-period generators, but the right generator fills the sample space much more uniformly than the left generator.

3.3 Characteristics of Good Random Number Generators

There are several properties that any random number generator must possess. First and foremost, the sequence of numbers must be distributed "randomly." Of course, pseudorandom numbers are not truly random; but they should be able to pass the same statistical tests for randomness that a sequence of truly random numbers would pass. However, the number of possible tests for randomness is uncountably infinite. Thus, the finite number of statistical tests used to evaluate and find the "best" set of parameters for a particular generator should be those that emphasize known weaknesses of the generator. For example, tests that examine the effect of the lattice structure should be emphasized for congruential generators. Extensive methods for testing pseudorandom number generators are given by Knuth [1981] and Marsaglia [1985].

A second property a generator must possess is a long period. For 32-bit computers, periods of $2^{30} \simeq 10^9$ are easily obtained. However, for complex applications longer periods of 10^{18} up to 10^{170} are desirable. The very long-period generators, which are now available, are very useful for generating long independent subsequences so that many collaborators can independently work on subsimulations and later combine their results assured of statistical independence.

A third property a generator must have is that of *portability*. This means that the generator must be written in a high-level language like FORTRAN or C without capitalizing on tricks peculiar to a particular computer. In this way, the same results are obtained on different computers. Generators written in machine language are generally not portable.

A final concern is the efficiency of the generator, i.e., the time it takes to generate a random number. In the early days this was a very important criterion and many machine-dependent tricks were used, that make most early generators nonportable. With modern computers, the generation time is almost insignificant, with the function call time usually exceeding the actual calculation time. If generation time is of concern, two methods can be used to reduce it. First, the generator can be coded in-line to avoid a function call. Second, each function call to the generator should return a large vector of random numbers, thereby, greatly reducing the number of function calls.

3.4 Tests for Congruential Generators

Because of the lattice formed by congruential generators, the effect of the lattice granularity is of great concern. Not only should the k-dimensional hypercube be uniformly filled with lattice points, but the hyperplanes and points also should be close together to reduce the effect of the granularity. Here, several of the statistical tests that have been used to test various congruent generators are reviewed. More details and the theory behind these tests can be found in Fishman [1996].

3.4.1 Spectral Test

The maximum period for a congruential generator with prime m is $t(m) = m-1$, while for a generation with $m = 2^\alpha$, $t(m) = m/4$. For a given multiplier a and modulus m, the distance between adjacent hyperplanes $d_k(a, m)$ is [Cassels 1959]

$$d_k(a,m) \geq d_k^{\min} \equiv \frac{1}{[t(m)]^{1/k}} \begin{cases} (3/4)^{1/4} & k=2 \\ 2^{-1/6} & k=3 \\ 2^{-1/4} & k=4, \\ 2^{-3/10} & k=5 \\ (3/64)^{1/12} & k=6 \end{cases} \qquad (3.5)$$

where $d_k^{\min}$ is the theoretical lower bound on the plane spacing. For a given modulus m the “best” multiplier a would produce $d_k(a, m)$ that is closest to the minimum possible hyperplane separation. The statistic

$$S_{1,k} = \frac{d_k^{\min}}{d_k(a,m)} \qquad (3.6)$$

is such that $0 < S_{1,k} \leq 1$ for $k \geq 2$, and those values of a that have $S_{1,k}$ as close to unity as possible for $2 \leq k \leq 6$ are sought.

3.4.2 Number of Hyperplanes

The number of hyperplanes $N_k(a, m)$ formed by the lattice in the k-dimensional hypercube is bounded by [Marsaglia 1968]

$$N_k(a,m) \leq [k!t(m)]^{1/k}. \qquad (3.7)$$

Then the statistic $S_{2,k} \equiv N_k(a, m)/[k!t(m)]^{1/k}$ has values between 0 and 1, and the best multiplier a from this test is the one that produces the largest values of $S_{2,k}$, i.e., the closest values to unity for $2 \leq k \leq 6$.

3.4.3 Distance between Points

The distance between the k-tuples in the k-dimensional hypercube should be as large as possible to ensure the best uniform coverage. Let $r_k(a, m)$ be the Euclidean distance between the nearest points. Then it can be shown that [Cassels 1959]

$$r_k(a,m) \leq \frac{1}{d_k^{\min}(a,m)[t(m)]^{2/k}}, \qquad (3.8)$$

where $d_k^{\min}$ is given by Eq. (3.5). The statistic $S_{3,k} \equiv r_k(a, m)d_k^{\min}(a, m)[t(m)]^{2/k}$ has values between 0 and 1, and the best multiplier a from this test is, again, the one that produces the largest values of $S_{3,k}$, i.e., the closest values to unity for $2 \leq k \leq 6$.

3.4.4 Other Tests

Often a test for *discrepancy* is used to evaluate random number generators. This test quantifies how equidistributed the points are in the k-dimensional hypercube compared to that which would occur with truly random numbers. It, thus, provides a benchmark that allows one to assess how closely the deterministic sequence $\{\rho_i\}$ looks random.

The lattice of k-tuples is defined by a set of k basis vectors $\mathbf{v_i}$. The closer the lengths $|\mathbf{v_1}|, \ldots, |\mathbf{v_k}|$ are to each other, the more equidistant are the k-tuples to each other in the k-dimensional hypercube. The *Beyer quotient* [Beyer et al. 1971] is

$$q_k(a, t(m)) \equiv \min_{1 \le i \le k} |\mathbf{v_i}| \Big/ \max_{1 \le i \le k} |\mathbf{v_i}|, \quad k > 1. \tag{3.9}$$

Multipliers with a quotient close to unity are preferred.

Finally, various statistical tests of various hypotheses of samples of length n from the sequence $\{\rho_i\}$ should be made. These include hypotheses that (1) the sample is a sequence of independent identically distributed random variables, (2) the $\{\rho_i\}$, $1 \le i \le n$, are distributed uniformly over (0, 1), (3) $\{\rho_{2i-1}, \rho_{2i}\}$, $1 \le i \le n/2$, are uniformly distributed over the unit square, (4) $\{\rho_{3i-2}, \rho_{3i-1}, \rho_{3i}\}$, $1 \le i \le (n-2)/2$, are distributed uniformly over the unit cube, and (5) all of the previous hypotheses are true simultaneously. Tests for these hypotheses are provided by Fishman [1996].

3.5 Practical Multiplicative Congruential Generators

The congruential random number generator is very fast, requiring very few operations to evaluate x_{i+1}, and, consequently, is now widely used. With some care this generator can be written in high-level computer languages and made portable. To produce very long periods and to reduce the granularity between the random numbers, the modulus m must be a very large integer, typically as large as the computer precision allows.

The generator of Eq. (3.3) can also be written in an alternate form by multiplying it by a^{-1}, the multiplicative inverse of a modulo m, defined as $a^{-1}a = 1 \bmod m$, to obtain

$$x_i = a^{-1} x_{i+1} \bmod m. \tag{3.10}$$

This means that for every sequence of random numbers there is another sequence that is the same but in reverse order provided $a \ne a^{-1} \bmod m$. This is what is observed in Example 3.1.

3.5.1 Generators with $m = 2^\alpha$

When digital computers were first introduced, computing efficiency was of major concern and often m was taken as some large power of 2, thereby enabling rapid calculation of the right side of Eq. (3.3). Just as multiplying (dividing) a decimal number by 10^n is easily done by moving the decimal point n places to the right (left), so is multiplying (dividing) a base 2 (or modulo 1) binary number by 2^n simply a matter of moving the binary point n places to the right (left). Shifting the binary point of a number in a computer register is much faster than performing multiplication or division. However, such algorithms usually require machine language coding, and, thus, usually are not portable to different computer architectures. However, today the time needed to compute a random number is usually negligible. But $m = 2^\alpha$ is still of major interest, because with large values of α, very long periods can be obtained.

The maximum period of a generator with a modulus 2^α, $(\alpha \geq 3)$ is $m/4$ or $2^{\alpha-2}$. This maximum cycle length is realized for any multiplier if a mod $8 = 3$ or 5 and the initial seed is an odd integer [Knuth 1981]. It is found that multipliers with $a = 5$ mod 8 produce numbers that are more uniformly distributed than those produced with $a = 3$ mod 8. That the seed must be odd is a nuance that a user must remember when using this type of generator.

What multipliers should be used for a given modulus? This is a topic that has received considerable attention with many early proposals having subsequently been shown to be somehow deficient. Fishman [1990] has exhaustively investigated congruential generators with $m = 2^{32}$ and with $m = 2^{48}$ by using the tests of Section 3.4 to find the "best" multipliers. A summary of his findings is given in Table 3.1. The first five multipliers for both $m = 2^{32}$ and $m = 2^{48}$ performed the best in all the statistical tests. For $m = 2^{32}$, the widely used multiplier 69,069 had significantly poorer test scores. Details and test scores are given by Fishman [1996].

A very practical example of a multiplicative congruential generator with modulus $m = 2^{48} \simeq 2.81 \times 10^{14}$ is that employed by the widely used particle transport code

Table 3.1 Recommended multipliers for generators of the form $x_{i+1} = ax_i \bmod 2^\alpha$. The second multiplier in the pairs shown in the first column is $a^{-1} \bmod 2^{32}$, and hence produces the same sequence as a, but in reverse order (after Fishman [1996]).

$m = 2^{32}$		$m = 2^{48}$	
$a = 5^n \bmod 2^{32}$	n	$a = 5^n \bmod 2^{48}$	n
1099087573[a]	9649599	68909602460261[a]	528329
4028795517	93795525		
2396548189[a]	126371437	33952834046453[a]	8369237
3203713013	245509143		
2824527309[a]	6634497	43272750451645[a]	99279091
1732073221	96810627		
3934873077[a]	181002903	127107890972165[a]	55442561
1749966429	190877677		
392314069[a]	160181311	55151000561141[a]	27179349
2304580733	211699269		
410092949[b]	–	44485709377909[c]	66290390456821
69069[d]	–	19073486328125[e]	19

[a]Fishman [1990], [b]Borosh and Niederreiter [1983], [c]Durst as reported by Fishman [1996], [d]Marsaglia [1972], [e]Beyer as reported by Fishman [1996].

MCNP [2003]. This generator has a multiplier $a = 5^{19} = 19{,}073{,}486{,}328{,}125$. This generator, which has a period of $2^{46} \simeq 7.04 \times 10^{13}$, also uses a stride (see Section 3.7) of 152,917 for each new particle history.[4]

3.5.2 Prime Modulus Generators

Many congruential random number generators take m as a large prime number (or a number with a large prime number as a factor). With a prime number modulus, a full-period generator is obtained for a multiplier a that is a *primitive root* modulo m [Fuller 1976].[5]

To find a primitive root for a given prime modulus, one can test successive values of $a = 2, 3, \ldots$ to find a relatively small primitive root a_1. However, this root is a poor choice for a prime modulus generator because a small value of x_i results in an ascending sequence of small values, hardly what we would expect from random numbers. To avoid such low-order correlations, a primitive root multiplier a should be chosen such that $a \lesssim \sqrt{m}$. To find larger primitive roots from a small primitive root a_1 it can be shown that [Fishman and Moore 1986]

$$a = a_1^n \bmod m \tag{3.11}$$

is also a primitive root if integer n is relative prime to $m - 1$, i.e., if n and $m - 1$ have no factors in common (other than 1). An illustration of finding the primitive roots is given in Example 3.2.

One widely used prime modulus is the Mersenne prime $2^{31} - 1 = 2{,}147{,}483{,}647$.[6] For this modulus, the smallest primitive root is $k_1 = 7$. The factors of $2^{31} - 1$ are 2, 3, 7, 11, 31, 151, and 331. Thus, $7^5 = 16{,}807$ is also a primitive root and produces a full cycle length. First, introduced by Lewis, Goodman, and Miller [1969], the random number generator

$$x_{i+1} = 7^5 x_i \bmod (2^{31} - 1) \tag{3.12}$$

has become well accepted. The modulus $m = 2^{31} - 1$ is an obvious choice for the 32-bit architecture of today's personal computers. This generator has passed many empirical tests for randomness, its theoretical properties have been well studied, and, more importantly, it has a vast history of successful applications.

[4] Beginning with MCNP5, an alternative, optional 63-bit generator is available that has a period of 9.2×10^{18}.

[5] If the smallest integer k that satisfies $a^k = 1 \bmod m$ is $k = m - 1$, then a is called a primitive root, modulo m.

[6] Mersenne primes have the form $2^p - 1$, where p is a prime number. Numbers of this form are prime for primes $p \leq 31$ (except for $p = 11$, 23, and 29). However, most large p do not yield primes. One exception is for $p = 32{,}582{,}657$, which produces a Mersenne prime number with 9,808,358 digits.

Example 3.2 Primitive Roots

Consider the simple generator $x_{i+1} = ax_i \bmod 13$. What values of a produce a full-cycle generator? For such a prime modulus generator all primitive roots produce full cycles. Thus, first find a small primitive root, i.e., find an a such that the smallest integer k that satisfies $a^k \bmod 13 = 1$ is $k = m-1 = 12$. It is easily verified that $2^k \bmod 13 = 2, 4, 8, 3, 6, 12, 11, 9, 5, 10, 7, 1$ for $k = 1, 2, \ldots, 12$. Hence, $a = 2$ is the smallest primitive root.

The factors of $m - 1 = 12$ are 3, 2, and 2. Then the values of $n < m - 1$ that have no factors (except 1) in common with 3 and 2 are $n = 5$, 7, and 11. Hence, the primitive roots for $m = 13$ are 2, $2^5 \bmod 13 = 6$, $2^7 \bmod 13 = 11$, and $2^{11} \bmod 13 = 7$. Thus, $a = 2, 6, 7$, and 11 produce full-period generators, a result previously seen in Example 3.1.

3.5.3 A Minimal Standard Congruential Generator

Park and Miller [1988] have proposed Eq. (3.3) with $m = 2^{31} - 1$ and $a = 16{,}807$ as a "minimal standard" generator because it passes three important tests: (1) it is a full-period generator, (2) its sequence $\ldots x_1, x_2, \ldots, x_{m-1}, x_1, \ldots$ is "random," and (3) $ax_i \bmod m$ can be efficiently implemented (with some clever coding discussed in Section 3.5.4) on a computer using 32-bit integer arithmetic. This is not to say that this particular generator is "ideal." In fact, it does have some flaws (see Section 3.5.5). But it is widely used and, as such, is a standard against which other random number generators can be compared.

Is the multiplier $m = 16807$ the optimum multiplier for use with $m = 2^{31} - 1$? Probably not. There are 534,600,000 primitive roots of $m = 2^{31} - 1$, which produce a full period. Hence, test 1 eliminates about 75% of the possible multipliers. Fishman and Moore [1986] examined the remaining 25% for passing test 2. To pass this test, the Euclidean distance between adjacent hyperplanes has to be no more than 25% of the theoretical minimum for dimensions $2 \leq k \leq 6$ (the *lattice test*). Only 410 multipliers met this criterion. The multiplier 16,807 is not among them (nor are other commonly used multipliers). However, *none* of these 410 optimal test-2 multipliers lend themselves to the clever programming trick of test 3 that permits a coding of the generator on machines using 32-bit integers. Park and Miller [1988] found that only 23,093 multipliers pass both tests 1 and 3. By replacing the original 25% criterion for test 2 by a 30% criterion, several good 32-bit multipliers were found, the two best being $a = 48{,}271$ and $a = 69{,}621$. Until further testing is done and confidence is gained with these two new multipliers, the original $a = 16{,}807$ will continue to be used widely.

3.5.4 Coding the Minimal Standard

Coding Park and Miller's minimal standard random number generator,

$$x_{i+1} = ax_i \bmod m \text{ with } a = 16807 \text{ and } m = 2^{31} - 1 = 2147483647, \tag{3.13}$$

for a computer based on 32-bit integers is not straightforward because, for $1 \leq x_i \leq m-1$, the product ax_i often exceeds the maximum value expressible as a 32-bit integer. One could avoid this problem by using the extended-precision option available in most computer languages. For example, in FORTRAN one could use

```
DOUBLE PRECISION FUNCTION randm(dseed)
DOUBLE PRECISION dseed
  dseed = DMOD(16807.d0*dseed,2147483647.d0)
  randm = dseed/2147483647.d0
RETURN
END
```

However, using double-precision floating-point arithmetic is considerably less efficient than integer arithmetic, especially because this function can be called billions of times in a typical Monte Carlo calculation.

It is much better to use 32-bit integer arithmetic and a trick proposed by Schrage [1979] and improved in 1983 [Bratley et al. 1983] to avoid the integer overflow problem. In this method, the modulus m is approximately factored as

$$m = aq + r, \quad \text{where} \quad q = [m/a] \quad \text{and} \quad r = m \bmod a. \tag{3.14}$$

With this factorization, the generator of Eq. (3.13) can be evaluated as [Press et al. 1996]

$$x_{i+1} = ax_i \bmod m = \begin{cases} a(x_i \bmod q) - rk & \text{if } \geq 0 \\ a(x_i \bmod q) - rk + m & \text{if above is } < 0 \end{cases}, \tag{3.15}$$

where $k = [x_i/q]$. How does this reformulation avoid integer overflows? Recall that $1 \leq x_i < m - 1$. If $r < q$, then it can be shown [Park and Miller 1988] that both $a(x_i \bmod q)$ and rk are in the range $0, \ldots, m-1$. This factorization with $r < q$ is the requirement for a generator to pass Park and Miller's test 3. For the minimal standard generator ($m = 2^{31} - 1$ and $a = 16{,}807$), the Schrage's factorization uses $q = 127{,}773$ and $r = 2836$.

Here is a very efficient FORTRAN routine that implements this algorithm for the minimal standard generator. The value of x_i is input as `iseed` and the value of x_{i+1} is returned in `iseed`. The returned value of `ran` is $\rho_{i+1} = x_{i+1}/m$.

```
 FUNCTION ran(iseed)
 INTEGER iseed,ia,im,iq,it,k
 REAL ran,am
 PARAMETER (ia=16807,im=2147483647,am=1./im,
&           iq=127773,it=2836)
 k = iseed/iq
 iseed = ia*(iseed-k*iq)-it*k
 IF(iseed.LT.0) iseed=iseed+im
 ran = am*iseed
 RETURN
 END
```

It is easy to modify this program to use the two optimal multipliers found by Park and Miller [1988], namely, $a = 48{,}271$ (with $q = 44{,}488$ and $r = 3399$) and $a = 69{,}621$ (with $q = 30{,}845$ and $r = 23{,}902$).

One final note about programming the minimal standard generator in other computer languages. It is easy to make mistakes, and to check the proper operation of the code, one should start with an initial seed of $x_1 = 1$ and ensure that the generator returns $x_{10001} = 1{,}043{,}618{,}065$. Implementation of this generator in other programming languages is considered in Appendix E.

3.5.5 Deficiencies of the Minimal Standard Generator

The minimal standard generator of Eq. (3.13) is not perfect. Indeed, a small value of ρ_i, say less than 10^{-6}, is produced one time in 10^6. This value is followed by another small number less than 0.017. For very improbable events, this serial correlation of small ρ_i values can lead to incorrect simulation results. Even more subtle low-order serial correlations exist in this generator [Press et al. 1996]. With any small multiplier a (needed to pass Park and Miller's test 3), all congruential generators have low-order serial correlations.

To minimize low-order correlations, the multiplier of a multiplicative congruential generator should be approximately equal to the square root of the modulus, which is more restrictive than the requirement that $r < q$ in order to pass Park and Miller's test 3.

3.5.6 Optimum Multipliers for Prime Modulus Generators

As Park and Miller [1988] suggest, the minimal standard's multiplier $a = 16{,}807$ is probably not the best multiplier for $m = 2^{31} - 1$. In a massive effort, Fishman and Moore [1986] exhaustively studied all possible primitive roots for this modulus, including the values 48,271 and 69,621 suggested by Park and Miller [1988] as being superior to the minimal standard multiplier. The results of the "best" multipliers are summarized in Table 3.2. The multiplier $a = 62{,}089{,}911$ was overall the best, with $a = 69{,}621$ the best for multipliers with $r < q$. Although 16,807 is not unreasonable,

Table 3.2 Recommended multipliers for prime modulus congruential random number generators (after Fishman [1996]).

$m = 2^{31} - 1$	multipliers a with $r < q$			
muliplier a	modulus m	multiplier a	$q = [m/a]$	$r = m \bmod a$
742938285[a]	$2^{31} - 1$	48271[b]	44488	3399
950706376[a]	$2^{31} - 1$	69621[b]	30845	23902
1226874159[a]	$2^{31} - 1$	16807[c]	127773	2836
62089911[a]	$2^{31} - 1$	39373[d]	54542	1481
1343714438[a]	$2^{31} - 85$	40014[d]	53668	12211
630360016[e]	$2^{31} - 249$	40692[d]	52774	3791

[a]Fishman [1996], [b]Park and Miller [1988], [c]Lewis et al. 1969, [d]L'Ecuyer [1988], [e]Payne et al. [1969].

it did significantly worse in the tests for randomness than did all the other multipliers shown in this table.

3.6 Shuffling a Generator's Output

The simplicity and ease of implementation of congruential random number generators have resulted in their widespread use despite known limitations such as low-order correlations and potentially imperfect equidistributedness. Generators with carefully chosen multipliers for a given modulus can produce k-tuples that are somewhat equidistributed. Is there some minor change that could be made in the generation process to make the k-tuples (or lattice) more equidistributed and random?

One simple idea is to shuffle the numbers produced by a generator so that all generated numbers appear in the output stream, but in an order different from the order in which they were generated. To accomplish this, one could use the output of a second (and perhaps simpler) generator to shuffle or permute the output sequence from the main generator. Such shuffling can increase the period because no longer does the same value follow a given value every time it occurs. Moreover, the shuffling breaks up the original lattice structure replacing it with a different one with a different number of hyperplanes. Another benefit of shuffling is that it removes low-order correlations.

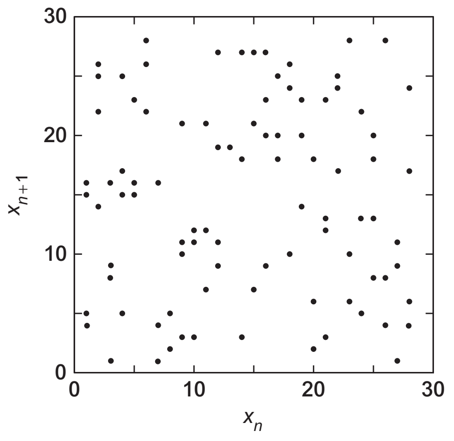
Bays and Durham [1976] have shown that a single generator can be used to first fill a table (vector) with x_i values and then use subsequent values from the random number generator to select and replace particular table entries. In this way, the output from the random number generator are shuffled and low-ordered correlations are removed. This shuffling can also increase the period of the generator.

The Bays and Durham shuffling algorithm is described below. The generator produces random values x_i and the output stream from the shuffling is denoted by y_i. The shuffling table (vector) has length k.

1. (Initialization) Fill the table vector T_j with $x_j,\ j = 1, \dots, k$; place $y_1 = x_{k+1}$ as the first entry in the output stream; and set $i = 1$.
2. Compute the table pointer $j = 1 + [ky_i/(m-1)]$.
3. Set $i = i + 1$.
4. Place $y_i = T_j$ into output stream.
5. Get the next number x_{k+i} and place it in T_j.
6. Loop back to step 2 to place another number into the output stream.

As an example of the effectiveness of this shuffling consider the example shown in the left plot of Fig. 3.1 where the full-cycle generator $x_{i+1} = 3x_i \bmod 29$ produces far from uniform distributions of 2-tuples over the square. With the Bays and Durham shuffling algorithm applied to this generator, the results shown in Fig. 3.2 are obtained. The cycle length is at least several thousand, and the plot area is much more uniformly covered.

Shuffling programs for the minimal standard random number generator are given by Press et al. [1996]. With such shuffling, even more statistical tests for randomness are met.



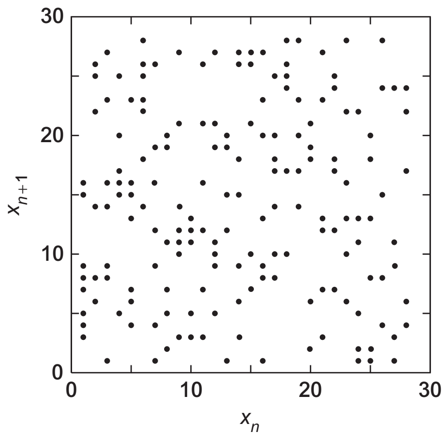


Figure 3.2 The effect of using the Bays and Durham shuffling scheme on the random number generator $x_{i+1} = 3x_i \bmod 29$ (used to generate the pair distribution of the left plots in Fig. 3.1). The left plot is for the first 101 pairs and the right plot is for the first 201 pairs. Besides a much longer cycle length, a more uniform distribution is achieved.

3.7 Skipping Ahead

Often it is useful to generate separate independent sequences of random numbers from the same congruential generator. For example, with a very long period generator, independent substrings could be obtained so that different machines could work simultaneously on the same simulation and the independent results from each machine can, therefore, be combined.

It is not a good idea to simply use a different seed for each sequence, because with bad choices the resulting sequences may overlap significantly. The best way to generate separate independent sequences is to skip ahead a specified distance in the sequence to start a new subsequence.

This skipping ahead is easily done for a congruential generator

$$x_{i+1} = ax_i \bmod m. \tag{3.16}$$

By recursive application of this relation it is easily shown (see problem 2) that

$$x_{i+k} = a^k x_i \bmod m. \tag{3.17}$$

With this relation, it is easy to start a new sequence some number of integers ahead of the start of the previous sequence.

Suppose the nth sequence started with x_n. Then the next sequence must begin with x_{n+s}, where s is the jump or *stride* between the two sequences. From Eq. (3.17) the $(n+1)$th sequence is begun with the integer

$$x_{n+s} = a^s x_n \bmod m = bx_n \bmod m, \quad \text{where} \quad b = a^s \bmod m. \tag{3.18}$$

The remainder of the new sequence is then obtained from Eq. (3.16).

3.8 Combining Generators

An obvious way to increase the period and randomness of a random number generator is to use two or more generators and merge their outputs. For example, Collins [1987] suggested using many generators each producing an output stream of random numbers. At each step, the output from one generator is used to select randomly one of the other generators of which the number is selected as the random number for that step. Such a procedure would vastly increase the cycle length. However, it is rather inefficient in that it requires many wasted calculations.

A note of caution. When one combines generators of which the output is in the open interval $(0, 1)$, it is possible, because of the finite machine precision, to obtain a 0 or a 1 (see problem 10). Special care must be exercised when coding such generators to prevent an endpoint being produced.

3.8.1 *Bit Mixing*

A very common way of improving randomness and increasing cycle length is to combine the integers x_i and y_i produced by two different generators to generate a new integer $z_i = x_i \odot y_i$, where $\odot$ is some binary operator, typically addition or subtraction modulo one, to produce a mixing of the bits in the base 2 representations of x_i and y_i. This is very efficient and easily implemented.

3.8.2 *The Wichmann-Hill Generator*

Wichmann and Hill [1982] proposed the following generator:

$$\begin{aligned}
x_{i+1} &= 171 x_i \bmod 30269 \\
y_{i+1} &= 172 y_i \bmod 30307 \\
z_{i+1} &= 170 z_i \bmod 30323
\end{aligned}$$

with the ith random number in $(0, 1)$ calculated as

$$\rho_i = \left(\frac{x_i}{30269} + \frac{y_i}{30307} + \frac{z_i}{30323} \right) \bmod 1. \tag{3.19}$$

This generator requires three seeds (x_0, y_0, z_0) and because the moduli are relatively prime, this generator cannot yield an exact zero (although finite machine precision requires careful programming to avoid zero). The period of this generator is about 10^{12}. This generator has been investigated for its long-range correlations and found to be comparable to other good generators [De Matteis and Pagnutti 1993]. Moreover, it is easily programmed for a 16-bit machine.

3.8.3 *The L'Ecuyer Generator*

L'Ecuyer [1988] has suggested a way to combine several different sequences with different periods so as to generate a sequence of which the period is the least common multiple of the two. Such a combination greatly helps reduce the serial correlations present in each generator. In particular, L'Ecuyer proposed the following generator [Gentle 1998]:

$$x_i = 40001x_{i-1} \bmod 2147483563 \tag{3.20}$$

$$y_i = 40692y_{i-1} \bmod 2147483399 \tag{3.21}$$

$$z_i = x_i - y_i, \tag{3.22}$$

where, if $z_i < 1$, then $z_i = z_i + 2{,}147{,}483{,}562$. The random number uniformly distributed in (0, 1) is given by

$$\rho_i = \frac{z_i}{2147483589} = 4.656613 z_i \times 10^{-10}. \tag{3.23}$$

Because x_i and y_i are uniform distributions over almost the same interval, the difference is also uniform. Moreover, a value of $\rho_i = 1$ cannot be obtained because the normalizing constant is slightly greater than 2,147,483,536. The period of this generator is of order 10^{18}, and L'Ecuyer claims this generator produces a good sequence of random numbers. It can also be programmed on a 32-bit machine (see James [1990] for a portable FORTRAN version). It has one minor limitation. It is not easy to generate separate independent sequences because it is not easy to skip ahead with this algorithm.

Press et al. [1996] offer the RAN2 program that is based on this L'Ecuyer generator with a final Bays-Durham shuffle added. This generator produces a sequence with a period of $> 2.3 \times 10^{18}$. The authors claim this generator is "perfect."

3.9 Other Random Number Generators

Variations to the congruential generators and some other random number generators have been proposed. The basic idea is that by making the generator more complex better randomness and longer periods can be achieved. The properties of many of these generators are often not well established, and the problem of finding optimum generators (in some sense) is still an area of active research. In the following, a few of these generator variants are briefly outlined. For a more in-depth discussion and for summaries of even more generators, the reader is referred to Gentle [1998] and Fishman [1996].

3.9.1 Multiple Recursive Generators

Instead of using only x_i to generate the next random number x_{i+1}, one could use several earlier numbers by using

$$x_i = (a_1 x_{i-1} + a_2 x_{i-2} + \cdots + a_n x_{i-n}) \bmod m. \tag{3.24}$$

Some of these *multiple recursive* multiplicative congruential generators can have much longer cycles than a simple multiplicative congruential generator. Such generators have been studied by L'Ecuyer et al. [1993] who make recommendations for various moduli (including $m = 10^{31} - 1$), give a portable code for $n = 5$ that does not yield a 0 or 1, and show how to skip ahead in the sequence.

3.9.2 Lagged Fibonacci Generators

Reducing the famous Fibonacci sequence $x_i = x_{i-1} + x_{i-2}$ modulo m has been proposed as the basis for a random number generator. However, it has poor random properties. Much better results are obtained if two earlier results some distance apart are combined. Thus, the *lagged Fibonacci congruential generator* is

$$x_i = (x_{i-j} + x_{i-k}) \bmod m. \tag{3.25}$$

If the lags j and k and the modulus m are chosen properly good random sequences can be obtained [Altman 1989] that can have cycle lengths as large as $m^k - 1$ [Gentle 1998]. Knuth [1981] gives suggested values for the lags j and k that produce periods for $j > k$ of $(2^j - 1)(2^{m-1})$.

Marsaglia et al. [1990] have proposed a lagged Fibonacci generator with lags of 97 and 33 that has a period of $2^{144} \simeq 10^{43}$. Moreover, it is portable and gives bit-identical results on all 32-bit machines and has passed extensive statistical tests despite limited theoretical understanding of this type of generator. James [1990] gives a very clever FORTRAN program RANMAR for this generator. A very useful feature of RANMAR is the ease with which independent subsequences can be obtained. The initialization of this generator requires a 32-bit integer. Each different integer produces an independent (nonoverlapping) sequence of average length $\simeq 10^{30}$, enough for very complex simulations.

3.9.3 Add-with-Carry Generators

A variant of the lagged Fibonacci generator, introduced by Marsaglia and Zaman [1991], is the *add-with-carry* generator

$$x_i = (x_{i-j} + x_{i-k} + c_i) \bmod m, \tag{3.26}$$

where the "carry" $c_1 = 0$ and c_{i+1} is given by

$$c_{i+1} = \begin{cases} 0, & \text{if } x_{i-j} + x_{i-k} + c_i < m \\ 1, & \text{otherwise} \end{cases}.$$

With proper choices for j, k, and m, very long period generators can be obtained. Marsaglia and Zaman give values that produce periods of the order of 10^{43}. Moreover, this generator can be implemented in base-2 arithmetic. It has since been shown that the add-with-carry generator is equivalent to a linear congruential generator with a very large prime modulus [Tezuka and L'Ecuyer 1992], which gives one some confidence in this generator and also shows how to program congruential generators with large prime moduli.

Other variations of this generator are the subtract-with-carry and the multiply-with-carry, also introduced by Marsaglia and Zaman [1991]. Much work still needs to be done to determine the random properties and optimal parameter values for these generators.

James [1990] gives a FORTRAN program for 32-bit machines for the subtract-with-carry variant. He uses $j = 24$, $k = 10$, and $m = 2^{24}$ to produce a generator with a period of $\simeq 2^{570} \simeq 10^{171}$!

3.9.4 Inversive Congruential Generators

The multiplicative inverse x^- of a number $x \bmod m$ is defined by $1 = x^- x \bmod m$. Eichenauer and Lehn [1986] proposed a random number generator based on the multiplicative inverse, namely,

$$x_i = (a x_{i-1}^- + c) \bmod m. \tag{3.27}$$

This generator does not yield regular lattice hyperplanes as does a linear congruential generator, and it appears to have better uniformity and fewer serial correlation problems. However, calculation of multiplicative inverses is computationally expensive.

3.9.5 Nonlinear Congruential Generators

Knuth [1981] suggested the generator

$$x_i = (d x_{i-1}^2 + a x_{i-1} + c) \bmod m \quad \text{with} \quad 0 \le x_i < m. \tag{3.28}$$

Other polynomials could be used as well. But the properties of such generators are not well established. Eichenauer et al. [1988] consider a more general variant, namely,

$$x_i = f(x_{i-1}) \bmod m, \tag{3.29}$$

with a wide variety of functions $f(x)$. Although such generators can produce very good uniformity properties, not enough is known yet for serious application of these generators. There needs to be much more study about nonlinear generators if they are to become useful.

3.10 Summary

Although Monte Carlo methods depend critically on procedures to generate efficiently a sequence of pseudorandom numbers that exhibit many of the qualities of true random numbers, a vast number of studies have relied on random number generators that are now known to be highly deficient. Many computer languages and code packages have built-in random number generators and, often, they are far from optimal. Indeed, it would be very interesting to see how many studies based on these inferior generators would have produced different results if they were based on modern proven random number generators. *Know thy random number generator!*

Bibliography

ALTMAN, N.S. 1989, "Bit-wise Behavior of Random Number Generators," *SIAM J. Sci. Stat. Comp.*, **9**, 839–847.

BAYS, C. AND S.D. DURHAM 1976, "Improving a Poor Random Number Generator," *ACM Trans. Mathematical Software*, **2**, 59–64.

BEYER, W.A., R.B. ROOF, AND D. WILLIAMSON 1971, "The Lattice Structure of Multiplicative Congruential Pseudorandom Vectors," *Math. Comput.*, **25**, 345–363.

BOROSH, I. AND H. NIEDERREITER 1983, "Optimal Multipliers for Pseudorandom Number Generation by the Linear Congruential Method," *BIT*, **23**, 65–74.

BRATLEY, P., B.L. FOX, AND E.L. SCHRAGE 1983, *A Guide to Simulation*, Springer-Verlag, New York.

CASSELS, J.W.S. 1959, *An Introduction to the Geometry of Numbers*, Springer-Verlag, Berlin.

COLLINS, B.J. 1987, "Compound Random Number Generators," *J. Am. Stat.*, **82**, 525–527.

DE MATTEIS, A. AND S. PAGNUTTI 1993, "Long-Range Correlation Analysis of the Wichmann-Hill Random Number Generator," *Stat. Comput.*, **3**, 67–70.

EICHENAUER, J., H. GROTHE, AND J. LEHN 1988, "Marsaglia's Lattice Test and Non-Linear Congruential Pseudo Random Number Generators," *Metrika*, **35**, 241–250.

EICHENAUER, J. AND J. LEHN 1986, "A Nonlinear Congruential Pseudo Random Number Generator," *Statistische Hefte*, **27**, 315–255.

FISHMAN, G.S. 1990, "Multiplicative Congruential Random Number Generators with Modulus 2^{β}: an Exhaustive Analysis for $\beta = 32$ and a Partial Analysis for $\beta = 48$," *Math. Comput.*, **54**, 331–334.

FISHMAN, G.S. 1996, *Monte Carlo: Concepts, Algorithms, and Applications*, Springer-Verlag, New York.

FISHMAN, G.S. AND L.R. MOORE 1986, "An Exhaustive Analysis of Multiplicative Congruential Random Number Generators with Modulus $2^{31} - 1$," *SIAM J. Sci. Stat. Comp.*, **7**, 24–25.

FULLER, A.T. 1976, "The Period of Pseudo-Random Numbers Generated by Lehmer's Congruential Method," *Comput. J.*, **19**, 173–177.

GENTLE, J.E. 1998, *Random Number Generation and Monte Carlo Methods*, Springer-Verlag, New York.

JAMES, F. 1990, "A Review of Pseudorandom Number Generators," *Comput. Phys. Comm.*, **60**, 329–344.

KNUTH, D.E. 1981, Semi-Numerical Algorithms, Vol. 2, in *The Art of Computer Programming*, 2nd ed., Addison-Wesley, Reading, MA.

L'ECUYER, P. 1988, "Efficient and Portable Combined Random Number Generators," *Commun. ACM*, **31**, 742–749.

L'ECUYER, P., F. BLOUIN, AND R. COUTURE 1993, "A Search for Good Multiplicative Recursive Random Number Generators," *ACM T. Model. Comput. S.*, **3**, 87–98.

LEHMER, D.H. 1951, "Mathematical Methods in Large-Scale Computing Units," *Annu. Comput. Lab. Harvard Univ.*, **26**, 141–146.

LEWIS, P.A.W., A.S. GOODMAN, AND J.M. MILLER 1969, " A Pseudo-Random Number Generator for the System/360," *IBM Syst. J.*, **8**, 136–146.

MARSAGLIA, G. 1968, "Random Numbers Fall Mainly in the Plane," *P. Natl. Acad. Sci. USA*, **61**, 25–28.

MARSAGLIA, G. 1972, "The Structure of Linear Congruential Sequences," in *Applications of Number Theory to Numerical Analysis*, S.K. ZAREMBA, ed., Academic Press, New York.

MARSAGLIA, G. 1985, "A Current View of Random Number Generators," in *Computer Science and Statistics: The Interface*, L. BILLARD, ed., Elsevier, Amsterdam.

MARSAGLIA, G. 1995, *The Marsaglia Random Number CDROM, Including the DIEHARD Battery of Tests of Randomness*, Dept. Statistics, Florida State University, Tallahassee.

MARSAGLIA, G., A. ZAMAN, AND W.-W. TSANG 1990, *Stat. Probabil. Lett.*, **9**, 35.

MARSAGLIA, G. AND A. ZAMAN 1991, "A New Class of Random Number Generators," *Ann. Appl. Probab.*, **1**, 462–480.

MCNP 2003, *A General Monte Carlo N-Particle Transport Code, Version 5*, Vol. I *Overview and Theory*, LA-UR-03-1987, Los Alamos National Lab., Los Alamos, NM.

PARK, S.K. AND K.W. MILLER 1988, "Random Number Generators: Good Ones Are Hard to Find," *Commun. ACM*, **31**, 1192–1201.

PAYNE, W.H., J.R. RABUNG, AND T.P. BOGYO 1969, "Coding the Lehmer Pseudorandom Number Generator," *Commun. ACM*, **12**, 85–86.

PRESS, W.H., S.A. TEUKOLSKY, W.T. VETTERLING, AND B.P. FLANNERY 1996, *Numerical Recipes*, 2nd ed., Cambridge University Press, Cambridge.

SCHRAGE, L. 1979, "A More Portable FORTRAN Random Number Generator," *ACM T. Math. Software*, **5**, 132–138.

TEZUKA, SHU AND P. L'ECUYER 1992, "Analysis of Add-with-Carry and Subtract-with-Borrow Generators," *Proc. 1992 Winter Simulation Conference*, 443–447, Assoc. Comp. Mach., New York.

WICHMANN B.A. AND I.D. HILL 1982, "An Efficient and Portable Pseudorandom Number Generator," *Applied Statistics*, **31**, 188–190 (corrections, 1984 *ibid.* **33**, 123).

Problems

1. Repeat the analysis for Example 3.1 using different seeds.
2. All variables in this problem are integers. The congruence $a \equiv b \bmod m$ means that $(b - a)$ is divisible by m. (a) From this observation, show that

$$ia = ib \bmod m \quad \text{and} \quad a^j = b^j \bmod m,$$

and (b) if $x_{n+1} \equiv ax_n \bmod m \quad n = 0, 1, \ldots$ show that

$$x_n = a^n x_0 \bmod m.$$

3. Show that the congruential generator $x_{i+1} = ax_i \bmod m$ implies $x_{i+k} = a^k x_i \bmod m$. Illustrate with a simple example.
4. In Fig. 3.1 overlapping pairs of numbers $(x_1, x_2),\ (x_2, x_3), \ldots$ were used to define the points on the plane. What would the plots look like if adjacent pairs $(x_1, x_2),\ (x_3, x_4), \ldots$ were used?
5. What is the minimum distance between hyperplanes in the hypercube of dimensions $k = 2, \ldots, 6$ for a congruential generator with modulus (a) $m = 2^{31} - 1$ and (b) $m = 2^{48}$? Comment on the granularity of the lattice points.
6. What is the maximum number of hyperplanes in the hypercube of dimensions $k = 2, \ldots, 6$ for a congruential generator with modulus (a) $m = 2^{31} - 1$ and (b) $m = 2^{48}$? What do these results tell you about the equidistributedness of the lattice points?
7. Using C or FORTRAN, write a multiplicative congruential generator based on the Mersenne prime $m = 2^8 - 1 = 31$. (a) What multipliers produce full periods? (b) Make plots of overlapping pairs of numbers similar to those in Fig. 3.1 and determine which multiplier(s) give the most uniform covering of the plane.

8. Determine the primitive roots for the generator $x_{i+1} = ax_i \bmod 17$ that produce full-cycle generators.
9. For many years the program RANDU was widely used. This generator is defined by

$$x_{i+1} = 65539x_i \bmod 2^{31}.$$

Although this generator has good random properties for the two-dimensional lattice, it has a very poor three-dimensional lattice structure because all triplets (x_i, x_{i+1}, x_{i+2}) lie on only 15 planes. Clearly, if such a triplet were used to define a starting location for a source particle in a transport calculation, the source would be poorly modeled.

Write a FORTRAN or C program to implement this generator and generate a sequence of length 30,002. For all triplets in this sequence, (x_i, x_{i+1}, x_{i+2}), in which $0.5 \le x_{i+1} \le 0.51$, plot x_i versus x_{i+2}. What does this plot tell you about this generator [Gentle 1998]?

10. Consider the combined generator of Eq. (3.19). Because of the relative primeness of the moduli, an exact 0 cannot be obtained. But could a code using integer arithmetic produce a 0? First, consider a computer with a very small binary integer, say, 2 bits; then extend to larger integer precisions. For a 32-bit computer, what is the probability a 0 is obtained?