6.26. Provide an intuitive motivation for the definition of MLEs in the continuous case (see Sec. 6.5) by going through steps (*a*) through (*c*) below. As before, the observed data are $X_1, X_2, \ldots, X_n$, and are IID realizations of a random variable X with density f_θ. Bear in mind that the X_i's have already been observed, so are to be regarded as fixed numbers rather than variables.

(*a*) Let ε be a small (but strictly positive) real number, and define the phrase "getting a value of X near X_i" to be the event $\{X_i - \varepsilon < X < X_i + \varepsilon\}$. Use the mean-value theorem from calculus to argue that $P(\text{getting a value of } X \text{ near } X_i) \approx 2\varepsilon f_\theta(X_i)$, for any $i = 1, 2, \ldots, n$.

(*b*) Define the phrase "getting a sample of n IID values of X near the observed data" to be the event (getting a value of X near X_1, getting a value of X near X_2, ..., getting a value of X near X_n). Show that $P(\text{getting a sample of } n \text{ IID values of } X \text{ near the observed data}) \approx (2\varepsilon)^n f_\theta(X_1) f_\theta(X_2) \cdots f_\theta(X_n)$, and note that this is *proportional* to the likelihood function $L(\theta)$.

(*c*) Argue that the MLE $\hat{\theta}$ is the value of θ that maximizes the approximate probability of getting a sample of n IID values of X near the observed data, and in this sense "best explains" the data that were actually observed.

6.27. Why is the average delay in queue approximately equal to the corresponding average number in queue in Table 6.2?

6.28. Show that Eq. (6.17) in Sec. 6.11 is correct.

6.29. Develop the general recursive formula for Newton's method to estimate the shape parameter α for the Weibull distribution in Sec. 6.11 [see Eq. (6.22)]. The formula should be of the following form:

$$\tilde{\alpha}_{k+1} = \tilde{\alpha}_k - \frac{f(\tilde{\alpha}_k)}{f'(\tilde{\alpha}_k)}$$

where f' denotes the derivative of f.

6.30. In the absence of data (Sec. 6.11), show how to specify a triangular distribution based on subjective estimates of a, m, and x_q.

CHAPTER 7

Random-Number Generators

Recommended sections for a first reading: 7.1, 7.2, 7.3.1, 7.3.2, 7.4.1, 7.4.3

7.1 INTRODUCTION

A simulation of any system or process in which there are inherently random components requires a method of generating or obtaining numbers that are *random*, in some sense. For example, the queueing and inventory models of Chaps. 1 and 2 required interarrival times, service times, demand sizes, etc., that were "drawn" from some specified distribution, such as exponential or Erlang. In this and the next chapter, we discuss how random values can be conveniently and efficiently generated from a desired probability distribution for use in executing simulation models. So as to avoid speaking of "generating random variables," which would not be strictly correct since a random variable is defined in mathematical probability theory as a function satisfying certain conditions, we will adopt more precise terminology and speak of "generating random *variates*."

This entire chapter is devoted to methods of generating random variates from the uniform distribution on the interval [0, 1]; this distribution was denoted by U(0, 1) in Chap. 6. Random variates generated from the U(0, 1) distribution will be called *random numbers*. Although this is the simplest continuous distribution of all, it is extremely important that we be able to obtain such independent random numbers. This prominent role of the U(0, 1) distribution stems from the fact that random variates from all other distributions (normal, gamma, binomial, etc.) and realizations of various random processes (e.g., a nonstationary Poisson process) can be obtained by transforming IID random numbers in a way determined by the desired distribution or process. This chapter discusses ways to obtain independent random

numbers, and the following chapter treats methods of transforming them to obtain variates from other distributions, and realizations of various processes.

The methodology of generating random numbers has a long and interesting history; see Hull and Dobell (1962), Morgan (1984, pp. 51–56), and Dudewicz (1975) for entertaining accounts. The earliest methods were essentially carried out by hand, such as casting lots (Matthew 27 : 35), throwing dice, dealing out cards, or drawing numbered balls from a "well-stirred urn." Many lotteries are still operated in this way, as is well known by American males who were of draft age in the late 1960s and early 1970s. In the early twentieth century, statisticians joined gamblers in their interest in random numbers, and mechanized devices were built to generate random numbers more quickly; in the late 1930s, Kendall and Babington-Smith (1938) used a rapidly spinning disk to prepare a table of 100,000 random digits. Some time later, electric circuits based on randomly pulsating vacuum tubes were developed that delivered random digits at rates of up to 50 per second. One such random-number machine, the Electronic Random Number Indicator Equipment (ERNIE), was used by the British General Post Office to pick the winners in the Premium Savings Bond lottery [see Thomson (1959)]. Another electronic device was used by the Rand Corporation (1955) to generate a table of a million random digits. Many other schemes have been contrived, such as picking numbers "randomly" out of phone books or census reports, or using digits in an expansion of π to 100,000 decimal places. There has been more recent interest in building and testing physical random-number "machines"; for example, Miyatake et al. (1983) describe a device based on counting gamma rays.

As computers (and simulation) became more widely used, increasing attention was paid to methods of random-number generation compatible with the way computers work. One possibility would be to hook up an electronic random-number machine, such as ERNIE, directly to the computer. This has several disadvantages, chiefly that we could not reproduce a previously generated random-number stream exactly. (The desirability of being able to do this is discussed later in this section.) Another alternative would be to read in a table, such as the Rand Corporation table, but this would entail either large memory requirements or a lot of time for relatively slow input operations. (Also, it is not at all uncommon for a modern large-scale simulation to use far more than a million random *numbers*, each of which would require several individual random *digits*.) Therefore, research in the 1940s and 1950s turned to *numerical* or *arithmetic* ways to generate "random" numbers. These methods are sequential, with each new number being determined by one or several of its predecessors according to a fixed mathematical formula. The first such arithmetic generator, proposed by von Neumann and Metropolis in the 1940s, is the famous *midsquare method*, an example of which follows.

EXAMPLE 7.1. Start with a four-digit positive integer Z_0 and square it to obtain an integer with up to eight digits; if necessary, append zeros to the left to make it exactly eight digits. Take the *middle four* digits of this eight-digit number as the next four-digit number, Z_1. Place a decimal point at the left of Z_1 to obtain the first "U(0, 1) random number," U_1. Then let Z_2 be the middle four digits of Z_1^2 and let U_2 be Z_2 with a decimal point at the left, and so on. Table 7.1 lists the first few Z_i's and U_i's for $Z_0 = 7182$ (the first four digits to the right of the decimal point in the number e).

TABLE 7.1
The midsquare method

i	Z_i	U_i	Z_i^2
0	7182	—	51,581,124
1	5811	0.5811	33,767,721
2	7677	0.7677	58,936,329
3	9363	0.9363	87,665,769
4	6657	0.6657	44,315,649
5	3156	0.3156	09,960,336
.	.	.	.
.	.	.	.
.	.	.	.

Intuitively the midsquare method seems to provide a good scrambling of one number to obtain the next, and so we might think that such a haphazard rule would provide a fairly good way of generating random numbers. In fact, it does not work very well at all. One serious problem (among others) is that it has a strong tendency to degenerate fairly rapidly to zero, where it will stay forever. (Continue Table 7.1 for just a few more steps, or try $Z_0 = 1009$, the first four digits from the Rand Corporation tables.) This illustrates the danger in assuming that a good random-number generator will always be obtained by doing something strange and nefarious to one number to obtain the next.

A more fundamental objection to the midsquare method is that it is not "random" at all, in the sense of being unpredictable. Indeed, if we know one number, the next is completely determined since the rule to obtain it is fixed; actually, when Z_0 is specified, the *whole sequence* of Z_i's and U_i's is determined. This objection applies to all arithmetic generators (the only kind we consider in the rest of this chapter), and arguing about it usually leads one quickly into mystical discussions about the true nature of truly random numbers. (Sometimes arithmetic generators are called *pseudorandom*, an awkward term that we avoid, even though it is probably more accurate.) Indeed, in an oft-quoted quip, John von Neumann (1951) declared that:

> Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin. For, as has been pointed out several times, there is no such thing as a random number—there are only methods to produce random numbers, and a strict arithmetic procedure of course is not such a method. . . . We are here dealing with mere "cooking recipes" for making digits. . . .

It is seldom stated, however, that von Neumann goes on in the same paragraph to say, less gloomily, that these "recipes"

> . . . probably . . . can not be justified, but should merely be judged by their results. Some statistical study of the digits generated by a given recipe should be made, but exhaustive tests are impractical. If the digits work well on one problem, they seem usually to be successful with others of the same type.

This more practical attitude was shared by Lehmer (1951), who developed what is probably still the most widely used class of techniques for random-number

generation (discussed in Sec. 7.2); he viewed the idea of an arithmetic random-number generator as

> . . . a vague notion embodying the idea of a sequence in which each term is unpredictable to the uninitiated and whose digits pass a certain number of tests traditional with statisticians and depending somewhat on the use to which the sequence is to be put.

More formal definitions of "randomness" in an axiomatic sense are cited by Ripley (1987, p. 19); Niederreiter (1978) argues that statistical randomness may not even be desirable, and that other properties of the generated numbers, such as "evenness" of the distribution of points, are more important in some applications, such as Monte Carlo integration. We agree with most writers that arithmetic generators, if designed carefully, can produce numbers that *appear* to be independent draws from the U(0, 1) distribution, in that they pass a series of statistical tests (see Sec. 7.4). This is a useful definition of "random numbers," to which we subscribe.

A "good" arithmetic random-number generator should possess several properties:

1. Above all, the numbers produced should appear to be distributed uniformly on [0, 1] and should not exhibit any correlation with each other; otherwise, the simulation's results may be completely invalid.
2. From a practical standpoint, we would naturally like the generator to be fast and avoid the need for a lot of storage.
3. We would like to be able to reproduce a given stream of random numbers exactly, for at least two reasons. First, this can sometimes make debugging or verification of the computer program easier. More important, we might want to use *identical* random numbers in simulating different systems in order to obtain a more precise comparison; Sec. 11.2 discusses this in detail.
4. There should be provision in the generator for easily producing separate "streams" of random numbers. As we shall see, a stream is simply a subsegment of the numbers produced by the generator, with one stream beginning where the previous stream ends. We can think of the different streams as being separate and independent generators (provided that we do not use up a whole stream, whose length is typically chosen to be a very large number). Thus, the user can "dedicate" a particular stream to a particular source of randomness in the simulation. We did this, for example, in the single-server queueing model of Sec. 2.4, where stream 1 was used for generating interarrival times and stream 2 for generating service times. Using separate streams for separate purposes facilitates reproducibility and comparability of simulation results. While this idea has obvious intuitive appeal, there is probabilistic foundation in support of it as well, as discussed in Sec. 11.2. Further advantages of having streams available are discussed in other parts of Chap. 11. The ability to create separate streams for a generator is facilitated if there is an efficient way to jump from the ith random number to the $(i + k)$th random number for large values of k.
5. We would like the generator to be portable, i.e., to produce the same sequence of random numbers (at least up to machine accuracy) for all standard compilers and computers (see Sec. 7.2.2).

Most of the commonly used generators are quite fast, require very little storage, and can easily reproduce a given sequence of random numbers, so that points 2 and 3 above are almost universally met. Furthermore, most generators now have the facility for multiple streams in some way, especially those generators included in modern simulation packages, satisfying point 4. Unfortunately, there are also many generators that fail to satisfy the uniformity and independence criteria of point 1 above, which are absolutely necessary if one hopes to obtain correct simulation results. The abundance of such statistically unacceptable generators is illustrated by the very title of the paper by Sawitzki (1985). Park and Miller (1988) and L'Ecuyer (2001) report several instances of published generators' displaying very poor performance, including one that can even repeat the same "random" number forever.

In Sec. 7.2 we discuss the most common kind of generator, while Sec. 7.3 discusses some alternative methods. Section 7.4 discusses how one can test a given random-number generator for the desired statistical properties. Finally, Apps. 7A and 7B contain portable computer code for two random-number generators in C. The first generator was used for the examples in Chaps. 1 and 2. The second generator is known to have better statistical properties and is recommended for real-world applications.

The subject of random-number generation is a complicated one, involving such disparate disciplines as abstract algebra and number theory, on one hand, and systems programming and computer hardware engineering, on the other. General references on random-number generators are the books by Fishman (1996, 2001, 2006), Gentle (2003), Knuth (1998a), and Tezuka (1995) and, in addition, the book chapters by L'Ecuyer (1998, 2004, 2006).

7.2 LINEAR CONGRUENTIAL GENERATORS

Many random-number generators in use today are *linear congruential generators* (LCGs), introduced by Lehmer (1951). A sequence of integers $Z_1, Z_2, \ldots$ is defined by the recursive formula

$$Z_i = (aZ_{i-1} + c) (\text{mod } m) \tag{7.1}$$

where m (the *modulus*), a (the *multiplier*), c (the *increment*), and Z_0 (the *seed* or *starting value*) are nonnegative integers. Thus, Eq. (7.1) says that to obtain Z_i, divide $aZ_{i-1} + c$ by m and let Z_i be the *remainder* of this division. Therefore, $0 \le Z_i \le m - 1$, and to obtain the desired random numbers U_i (for $i = 1, 2, \ldots$) on [0, 1], we let $U_i = Z_i/m$. We shall concentrate our attention for the most part on the Z_i's, although the precise nature of the division of Z_i by m should be paid attention to due to differences in the way various computers and compilers handle floating-point arithmetic. In addition to nonnegativity, the integers m, a, c, and Z_0 should satisfy $0 < m$, $a < m$, $c < m$, and $Z_0 < m$.

Immediately, two objections could be raised against LCGs. The first objection is one common to all (pseudo) random-number generators, namely, that the Z_i's defined by Eq. (7.1) are not really random at all. In fact, one can show by

mathematical induction that for $i = 1, 2, \ldots,$

$$Z_i = \left[a^i Z_0 + \frac{c(a^i - 1)}{a - 1}\right] \pmod{m}$$

so that *every* Z_i is completely determined by m, a, c, and Z_0. However, by careful choice of these four parameters we try to induce behavior in the Z_i's that makes the corresponding U_i's *appear* to be IID U(0, 1) random variates when subjected to a variety of tests (see Sec. 7.4).

The second objection to LCGs might be that the U_i's can take on only the rational values $0, 1/m, 2/m, \ldots, (m-1)/m$; in fact, the U_i's might actually take on only a fraction of these values, depending on the specification of the constants m, a, c, and Z_0, as well as on the nature of the floating-point division by m. Thus there is no possibility of getting a value of U_i between, say, $0.1/m$ and $0.9/m$, whereas this *should* occur with probability $0.8/m > 0$. As we shall see, the modulus m is usually chosen to be very large, say 10^9 or more, so that the points in [0, 1] where the U_i's can fall are very dense; for $m \geq 10^9$, there are at least a billion possible values.

EXAMPLE 7.2. Consider the LCG defined by $m = 16$, $a = 5$, $c = 3$, and $Z_0 = 7$. Table 7.2 gives Z_i and U_i (to three decimal places) for $i = 1, 2, \ldots, 19$. Note that $Z_{17} = Z_1 = 6$, $Z_{18} = Z_2 = 1$, and so on. That is, from $i = 17$ through 32, we shall obtain *exactly* the same values of Z_i (and hence U_i) that we did from $i = 1$ through 16, and in *exactly* the same order. (We do not seriously suggest that anyone use this generator since m is so small; it only illustrates the arithmetic of LCGs.)

The "looping" behavior of the LCG in Example 7.2 is inevitable. By the definition in Eq. (7.1), whenever Z_i takes on a value it has had previously, exactly the same sequence of values is generated, and this cycle repeats itself endlessly. The length of a cycle is called the *period* of a generator. For LCGs, Z_i depends *only* on the previous integer Z_{i-1}, and since $0 \leq Z_i \leq m - 1$, it is clear that the period is at most m; if it is in fact m, the LCG is said to have *full period*. (The LCG in Example 7.2 has full period.) Clearly, if a generator is full-period, any choice of the initial seed Z_0 from $\{0, 1, \ldots, m-1\}$ will produce the entire cycle in some order. If, however, a generator has less than full period, the cycle length could in fact depend on the particular value of Z_0 chosen, in which case we should really refer to the period of the *seed* for this generator.

Since large-scale simulation projects can use millions of random numbers, it is manifestly desirable to have LCGs with long periods. Furthermore, it is comforting

TABLE 7.2
The LCG $Z_i = (5Z_{i-1} + 3) \pmod{16}$ with $Z_0 = 7$

i	Z_i	U_i	i	Z_i	U_i	i	Z_i	U_i	i	Z_i	U_i
0	7	—	5	10	0.625	10	9	0.563	15	4	0.250
1	6	0.375	6	5	0.313	11	0	0.000	16	7	0.438
2	1	0.063	7	12	0.750	12	3	0.188	17	6	0.375
3	8	0.500	8	15	0.938	13	2	0.125	18	1	0.063
4	11	0.688	9	14	0.875	14	13	0.813	19	8	0.500

to have full-period LCGs, since we are assured that every integer between 0 and $m - 1$ will occur exactly once in each cycle, which should contribute to the uniformity of the U_i's. (Even full-period LCGs, however, can exhibit nonuniform behavior in segments within a cycle. For example, if we generate only $m/2$ consecutive Z_i's, they may leave large gaps in the sequence $0, 1, \ldots, m-1$ of possible values.) Thus, it is useful to know how to choose m, a, and c so that the corresponding LCG will have full period. The following theorem, proved by Hull and Dobell (1962), gives such a characterization.

THEOREM 7.1. The LCG defined in Eq. (7.1) has full period if and only if the following three conditions hold:

(*a*) The only positive integer that (exactly) divides both m and c is 1.
(*b*) If q is a prime number (divisible by only itself and 1) that divides m, then q divides $a - 1$.
(*c*) If 4 divides m, then 4 divides $a - 1$.

[Condition (*a*) in Theorem 7.1 is often stated as "c is relatively prime to m."]

Obtaining a full (or at least a long) period is just one desirable property for a good LCG; as indicated in Sec. 7.1, we also want good statistical properties (such as apparent independence), computational and storage efficiency, reproducibility, facilities for separate streams, and portability (see Sec. 7.2.2). Reproducibility is simple, for we must only remember the initial seed used, Z_0, and initiate the generator with this value again to obtain the same sequence of U_i's exactly. Also, we can easily resume generating the Z_i's at any point in the sequence by saving the final Z_i obtained previously and using it as the new seed; this is a common way to obtain nonoverlapping, "independent" sequences of random numbers.

Streams are typically set up in a LCG by simply specifying the initial seed for each stream. For example, if we want streams of length 1,000,000 each, we set Z_0 for the first stream to some value, then use $Z_{1,000,000}$ as the seed for the second stream, $Z_{2,000,000}$ as the seed for the third stream, and so on. Thus, we see that streams are actually nonoverlapping adjacent subsequences of *the single* sequence of random numbers being generated; if we were to use more than 1,000,000 random numbers from one stream in the above example, we would be encroaching on the beginning of the next stream, which might already have been used for something else, resulting in unwanted correlation.

In the remainder of this section we consider the choice of parameters for obtaining good LCGs and identify some poor LCGs that are still in use. Because of condition (*a*) in Theorem 7.1, LCGs tend to behave differently for $c > 0$ (called *mixed* LCGs) than for $c = 0$ (called *multiplicative* LCGs).

7.2.1 Mixed Generators

For $c > 0$, condition (*a*) in Theorem 7.1 is possible, so we might be able to obtain full period m, as we now discuss. For a large period and high density of the U_i's on [0,1], we want m to be large. Furthermore, in the early days of computer simulation when computers were relatively slow, dividing by m to obtain the remainder in Eq. (7.1) was a relatively slow arithmetic operation, and it was desirable to avoid

having to do this division explicitly. A choice of m that is good in all these respects is $m = 2^b$, where b is the number of bits (*bi*nary digi*ts*) in a word on the computer being used that are available for actual data storage. For example, most computers and compilers have 32-bit words, the leftmost bit being a sign bit, so $b = 31$ and $m = 2^{31} > 2.1$ billion. Furthermore, choosing $m = 2^b$ does allow us to avoid explicit division by m on most computers by taking advantage of *integer overflow*. The largest integer that can be represented is $2^b - 1$, and any attempt to store a larger integer W (with, say, $h > b$ bits) will result in loss of the left (most significant) $h - b$ bits of this oversized integer. What remains in the retained b bits is precisely $W \pmod{2^b}$.

With the choice of $m = 2^b$, Theorem 7.1 says that we shall obtain a full period if c is odd and $a - 1$ is divisible by 4. Furthermore, Z_0 can be any integer between 0 and $m - 1$ without affecting the period. We will, however, focus on multiplicative LCGs in the remainder of Sec. 7.2, because they are much more widely used.

7.2.2 Multiplicative Generators

Multiplicative LCGs are advantageous in that the addition of c is not needed, but they cannot have full period since condition (*a*) of Theorem 7.1 cannot be satisfied (because, for example, m is positive and divides both m and $c = 0$). As we shall see, however, it is possible to obtain period $m - 1$ if m and a are chosen carefully.

As with mixed generators, it's still computationally efficient to choose $m = 2^b$ and thus avoid explicit division. However, it can be shown [see, for example, Knuth (1998a, p. 20)] that in this case the period is at most 2^{b-2}, that is, only *one-fourth* of the integers 0 through $m - 1$ can be obtained as values for the Z_i's. (In fact, the period is 2^{b-2} if Z_0 is odd and a is of the form $8k + 3$ or $8k + 5$ for some $k = 0, 1, \ldots$.) Furthermore, we generally shall not know *where* these $m/4$ integers will fall; i.e., there might be unacceptably large gaps in the Z_i's obtained. Additionally, if we choose a to be of the form $2^l + j$ (so that the multiplication of Z_{i-1} by a is replaced by a shift and j adds), poor statistical properties can be induced. The generator usually known as RANDU is of this form ($m = 2^{31}$, $a = 2^{16} + 3 = 65{,}539$, $c = 0$) and has been shown to have very undesirable statistical properties (see Sec. 7.4). Even if one does not choose $a = 2^l + j$, using $m = 2^b$ in multiplicative LCGs is probably not a good idea, if only because of the shorter period of $m/4$ and the resulting possibility of gaps.

Because of these difficulties associated with choosing $m = 2^b$ in multiplicative LCGs, attention was paid to finding other ways of specifying m. Such a method, which has proved to be quite successful, was reported by Hutchinson (1966), who attributed the idea to Lehmer. Instead of letting $m = 2^b$, it was proposed that m be the largest prime number that is less than 2^b. For example, in the case of $b = 31$, the largest prime that is less than 2^{31} is, very agreeably, $2^{31} - 1 = 2{,}147{,}483{,}647$. Now for m prime, it can be shown that the period is $m - 1$ if a is a *primitive element modulo* m; that is, the smallest integer l for which $a^l - 1$ is divisible by m is $l = m - 1$; see Knuth (1998a, p. 20). With m and a chosen in this way, we obtain each integer 1, 2, . . . , $m - 1$ exactly once in each cycle, so that Z_0 can be any integer from 1 through $m - 1$ and a period of $m - 1$ will still result. These are called *prime modulus multiplicative LCGs* (PMMLCGs).

Two issues immediately arise concerning PMMLCGs: (1) How does one obtain a primitive element modulo m? Although Knuth (1998a, pp. 20–21) gives some characterizations, the task is quite complicated from a computational standpoint. We shall, in essence, finesse this point by discussing below two widely used PMMLCGs. (2) Since we are not choosing $m = 2^b$, we can no longer use the integer overflow mechanism directly to effect division modulo m. A technique for avoiding explicit division in this case, which also uses a type of overflow, was given by Payne, Rabung, and Bogyo (1969) and has been called *simulated division*. Marse and Roberts' portable generator, which we discuss below, uses simulated division.

Considerable work has been directed toward identifying good multipliers a for PMMLCGs that are primitive elements modulo $m^* = 2^{31} - 1$, which result in a period of $m^* - 1$. In an important set of papers, Fishman and Moore (1982, 1986) evaluated *all* multipliers a that are primitive elements modulo m^*, numbering some 534 million. They used both empirical and theoretical tests (see Sec. 7.4 below), and they identified several multipliers that perform well according to a number of fairly stringent criteria.

Two particular values of a that have been widely used for the modulus m^* are $a_1 = 7^5 = 16{,}807$ and $a_2 = 630{,}360{,}016$, both of which are primitive elements modulo m^*. [However, neither value of a was found by Fishman and Moore to be among the best (see Sec. 7.4.2).] The multiplier a_1 was originally suggested by Lewis, Goodman, and Miller (1969), and it was used by Schrage (1979) in a clever FORTRAN implementation using simulated division. The importance of Schrage's code was that it provided at that time a reasonably good and portable random-number generator.

The multiplier a_2, suggested originally by Payne, Rabung, and Bogyo (1969), was found by Fishman and Moore to yield statistical performance better than does a_1 (see Sec. 7.4.2). Marse and Roberts (1983) provided a highly portable FORTRAN routine for this multiplier, and a C version of this generator is given in App. 7A. This is the generator that we used for all the examples in Chaps. 1 and 2, and it is the one built into the simlib package in Chap. 2.

The PMMLCG with $m = m^* = 2^{31} - 1$ and $a = a_2 = 630{,}360{,}016$ may provide acceptable results for some applications, particularly if the required number of random numbers is not too large. However, many experts [see, e.g., L'Ecuyer, Simard, Chen, and Kelton (2002) and Gentle (2003, p. 21)] recommend that LCGs with a modulus of around 2^{31} should no longer be used as the random-number generator in a general-purpose software package (e.g., for discrete-event simulation). Not only can the period of the generator be exhausted in a few minutes on many computers, but, more importantly, the relatively poor statistical properties of these generators can bias simulation results for sample sizes that are *much smaller* than the period of the generator. For example, L'Ecuyer and Simard (2001) found that the PMMLCGs with modulus m^* and multipliers a_1 or a_2 exhibit a certain departure from what would be expected in a sample of independent observations from the U(0, 1) distribution if the number of observations in the sample is approximately

8 times the *cube root* of the period of the generator. Thus, the "safe" period of these generators is actually approximately 10,000 [see also L'Ecuyer et al. (2000)].

If a random-number generator with a larger period and better statistical properties is desired, then the combined multiple recursive generator of L'Ecuyer or the Mersenne twister (see Secs. 7.3.2 and 7.3.3, respectively) should be considered.

7.3 OTHER KINDS OF GENERATORS

Although LCGs are probably the most widely used and best understood kind of random-number generator, there are many alternative types. (We have already seen one alternative in Sec. 7.1, the midsquare method, which is not recommended.) Most of these other generators have been developed in an attempt to obtain longer periods and better statistical properties. Our treatment in this section is meant not to be an exhaustive compendium of all kinds of generators, but only to indicate some of the main alternatives to LCGs.

7.3.1 More General Congruences

LCGs can be thought of as a special case of generators defined by

$$Z_i = g(Z_{i-1}, Z_{i-2}, \ldots)(\text{mod } m) \tag{7.2}$$

where g is a fixed deterministic function of previous Z_j's. As with LCGs, the Z_i's defined by Eq. (7.2) lie between 0 and $m - 1$, and the U(0, 1) random numbers are given by $U_i = Z_i/m$. [For LCGs, the function g is, of course, $g(Z_{i-1}, Z_{i-2}, \ldots) = aZ_{i-1} + c$.] Here we briefly discuss a few of these kinds of generators and refer the reader to Knuth (1998a, pp. 26–36) or L'Ecuyer (2004) for a more detailed discussion.

One obvious generalization of LCGs would be to let $g(Z_{i-1}, Z_{i-2}, \ldots) = a'Z_{i-1}^2 + aZ_{i-1} + c$, which produces a *quadratic* congruential generator. A special case that has received some attention is when $a' = a = 1$, $c = 0$, and m is a power of 2; although this particular generator turns out to be a close relative of the midsquare method (see Sec. 7.1), it has better statistical properties. Since Z_i still depends only on Z_{i-1} (and not on earlier Z_j's), and since $0 \le Z_i \le m - 1$, the period of quadratic congruential generators is at most m, as for LCGs.

A different choice of the function g is to maintain linearity but to use earlier Z_j's; this gives rise to generators called *multiple recursive generators* (MRGs) and defined by

$$g(Z_{i-1}, Z_{i-2}, \ldots) = a_1Z_{i-1} + a_2Z_{i-2} + \cdots + a_qZ_{i-q} \tag{7.3}$$

where $a_1, a_2, \ldots, a_q$ are constants. A period as large as $m^q - 1$ then becomes possible if the parameters are chosen properly [see Knuth (1998a, pp. 29–30)]. L'Ecuyer, Blouin, and Couture (1993) investigated such generators, and included a generalization of the spectral test (see Sec. 7.4.2) for their evaluation; they also identified several specific generators of this type that perform well, and give portable implementations. Additional attention to generators with g of the form in Eq. (7.3) used in Eq. (7.2) has focused on g's defined as $Z_{i-1} + Z_{i-q}$, which includes the old Fibonacci generator

$$Z_i = (Z_{i-1} + Z_{i-2})(\text{mod } m)$$

This generator tends to have a period in excess of m but is completely unacceptable from a statistical standpoint; see Prob. 7.12.

A generalization of LCGs along a different line was proposed by Haas (1987), who suggested that in the basic LCG of Eq. (7.1) we change both the multiplier a and the increment c according to congruential formulas before generating each new Z_i. Statistical tests of this type of generator (see Sec. 7.4.1) appeared favorable, and his analysis indicated that we can readily obtain very large periods, such as 800 trillion in one example.

7.3.2 Composite Generators

Several researchers have developed methods that take two or more *separate* generators and combine them in some way to generate the final random numbers. It is hoped that this *composite* generator will exhibit a longer period and better statistical behavior than any of the simple generators composing it. The disadvantage in using a composite generator is, of course, that the cost of obtaining each U_i is more than that of using one of the simple generators alone.

Perhaps the earliest kind of composite generators used a second LCG to *shuffle* the output from the first LCG; they were developed by MacLaren and Marsaglia (1965) and extended by Marsaglia and Bray (1968), Grosenbaugh (1969), and Nance and Overstreet (1975). Initially, a vector $\mathbf{V} = (V_1, V_2, \ldots, V_k)$ is filled sequentially with the first k U_i's from the first LCG ($k = 128$ was originally suggested). Then the second LCG is used to generate a random integer I distributed uniformly on the integers 1, 2, . . . , k (see Sec. 8.4.2), and V_I is returned as the first U(0, 1) variate; the first LCG then replaces this Ith location in $\mathbf{V}$ with its next U_i, and the second LCG randomly chooses the next returned random number from this updated $\mathbf{V}$, etc. Shuffling has a natural intuitive appeal, especially since we would expect it to break up any correlation and greatly extend the period. Indeed, MacLaren and Marsaglia obtained a shuffling generator with very good statistical behavior even though the two individual LCGs were quite poor. In a subsequent evaluation of shuffling, Nance and Overstreet (1978) confirm that shuffling one bad LCG by another bad LCG can result in a good composite generator, e.g., by extending the period when used on computers with short word lengths, but that little is accomplished by shuffling a good LCG. In addition, they found that a vector of length $k = 2$ works as well as much larger vectors.

Several variations on this shuffling scheme have been considered; Bays and Durham (1976) and Gebhardt (1967) propose shuffling a generator by itself rather than by another generator. Atkinson (1980) also reported that simply applying a *fixed* permutation (rather than a random shuffling) of the output of LCGs

which is denoted by $b_i = b_{i-r} \oplus b_{i-q}$. To initialize the $\{b_i\}$ sequence, $b_1, b_2, \ldots, b_q$ must be specified somehow; this is akin to specifying the seed Z_0 for LCGs.

To form a sequence of binary integers $W_1, W_2, \ldots$, we string together l consecutive b_i's and consider this as a *number in base 2*. That is,

$$W_1 = b_1 b_2 \cdots b_l$$

and

$$W_i = b_{(i-1)l+1} b_{(i-1)l+2} \cdots b_{il} \qquad \text{for } i = 2, 3, \ldots$$

Note that the recurrence for the W_i's is the same as the recurrence for the b_i's given by (7.5), namely

$$W_i = W_{i-r} \oplus W_{i-q} \tag{7.6}$$

where the exclusive-or operation is performed bitwise. The ith U(0, 1) random number U_i is then defined by

$$U_i = \frac{W_i}{2^l} \qquad \text{for } i = 1, 2, \ldots$$

The maximum period of the $\{b_i\}$ sequence is $2^q - 1$, since $b_{i-1}, b_{i-2}, \ldots, b_{i-q}$ can take on 2^q different possible states and the occurrence of the q-tuple $0, 0, \ldots, 0$ would cause the $\{b_i\}$ sequence to stay in that state forever. Let

$$f(x) = x^q + c_1 x^{q-1} + \cdots + c_{q-1} x + 1$$

be the characteristic polynomial of the recurrence given by (7.4). Tausworthe (1965) showed that the period of the b_i's is, in fact, $2^q - 1$ if and only if the polynomial $f(x)$ is *primitive* [see Knuth (1998a, pp. 29–30)] over the Galois field $\mathcal{F}_2$, which is the set $\{0, 1\}$ on which the binary operations of addition and multiplication modulo 2 are defined. If l is relatively prime to $2^q - 1$, then the period of the W_i's (and the U_i's) will also be $2^q - 1$. Thus, for a computer with 31 bits for actual data storage, the maximum period is $2^{31} - 1$, which is the same as that for a LCG.

EXAMPLE 7.3. Let $r = 3$ and $q = 5$ in Eq. (7.5), and let $b_1 = b_2 = \cdots = b_5 = 1$. Thus, for $i \geq 6$, b_i is the "exclusive-or" of b_{i-3} with b_{i-5}. In this case, $f(x)$ is the trinomial $x^5 + x^2 + 1$, which is, in fact, primitive over $\mathcal{F}_2$. The first 40 b_i's are then

1111100011011101010000100101100111110001

Note that the period of the bits is $31 = 2^5 - 1$, since b_{32} through b_{36} are the same as b_1 through b_5. If $l = 4$ (which is relatively prime to 31), then the following sequence of W_i's is obtained:

15, 8, 13, 13, 4, 2, 5, 9, 15, 1, ...

which also has a period of 31 (see Prob. 7.15). The corresponding U_i's are obtained by dividing the W_i's by $16 = 2^4$.

The original motivation for suggesting that the b_i's be used as a source of U(0, 1) random numbers came from the observation that the recurrence given by (7.4) can be implemented on a binary computer using a switching circuit called a *linear feedback*

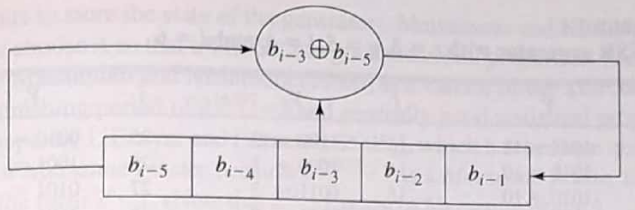

FIGURE 7.1
A LFSR corresponding to the recurrence (7.5) with $r = 3$ and $q = 5$.

shift register (LFSR). This is an array of q bits that is shifted, say, to the left one position at a time, with the bit shifted out on the left combined with other bits in the array to form the new rightmost bit. Because of the relationship between the recurrence (7.4) and a feedback shift register, Tausworthe generators are also called *LFSR generators*.

EXAMPLE 7.4. The generator discussed in Example 7.3 can be represented by the LFSR shown in Fig. 7.1. The bits b_{i-3} and b_{i-5} are combined using the exclusive-or operation to produce a new bit that goes into the rightmost location (i.e., the one that previously contained bit b_{i-1}) of the array. The bit that was in the leftmost location of the array (i.e., b_{i-5}) is removed from the array. The values of b_{i-5}, b_{i-4}, b_{i-3}, b_{i-2}, and b_{i-1} for $i = 6, 7, \ldots, 15$ are given in Table 7.3.

Unfortunately, LFSR generators are known to have statistical deficiencies, as discussed by Matsumoto and Kurita (1996) and Tezuka (1995). However, L'Ecuyer (1996b, 1999b) considered combined LFSR generators, which have better statistical properties and a larger period.

Lewis and Payne (1973) introduced a modification of the LFSR generator that they called a *generalized feedback shift register* (GFSR) generator. To obtain a sequence of l-bit binary integers $Y_1, Y_2, \ldots$, the sequence of bits $b_1, b_2, \ldots$ produced

TABLE 7.3
Successive states of the LFSR for $r = 3$ and $q = 5$

i	$b_{i-5}^{\dagger}$	b_{i-4}	$b_{i-3}^{\dagger}$	b_{i-2}	b_{i-1}
6	1	1	1	1	1
7	1	1	1	1	0
8	1	1	1	0	0
9	1	1	0	0	0
10	1	0	0	0	1
11	0	0	0	1	1
12	0	0	1	1	0
13	0	1	1	0	1
14	1	1	0	1	1
15	1	0	1	0	1

†Source of bits.

7.4.1 Empirical Tests

Perhaps the most direct way to test a generator is to *use* it to generate some U_i's, which are then examined statistically to see how closely they resemble IID U(0, 1) random variates. We discuss four such empirical tests; several others are treated in Banks et al. (2001, pp. 264–284), Fishman (1978, pp. 371–386), Knuth (1998a, pp. 41–75), L'Ecuyer et al. (2000), and L'Ecuyer and Simard (2001).

The first test is designed to check whether the U_i's appear to be uniformly distributed between 0 and 1, and it is a special case of a test we have seen before (in Sec. 6.6.2), the chi-square test with all parameters known. We divide [0, 1] into k subintervals of equal length and generate $U_1, U_2, \ldots, U_n$. (As a general rule, k should be at least 100 here.) For $j = 1, 2, \ldots, k$, let f_j be the number of the U_i's that are in the jth subinterval, and let

$$\chi^2 = \frac{k}{n} \sum_{j=1}^{k} \left(f_j - \frac{n}{k} \right)^2$$

Then for large n, χ^2 will have an approximate chi-square distribution with $k-1$ df under the null hypothesis that the U_i's are IID U(0, 1) random variables. Thus, we reject this hypothesis at level α if $\chi^2 > \chi^2_{k-1,1-\alpha}$, where $\chi^2_{k-1,1-\alpha}$ is the upper $1 - \alpha$ critical point of the chi-square distribution with $k - 1$ df. [For the large values of k likely to be encountered here, we can use the approximation

$$\chi^2_{k-1,1-\alpha} \approx (k-1)\left\{ 1 - \frac{2}{9(k-1)} + z_{1-\alpha} \sqrt{\frac{2}{9(k-1)}} \right\}^3$$

where $z_{1-\alpha}$ is the upper $1 - \alpha$ critical point of the N(0, 1) distribution.]

EXAMPLE 7.6. We applied the chi-square test of uniformity to the PMMLCG $Z_i = 630{,}360{,}016 Z_{i-1} (\mathrm{mod}\, 2^{31} - 1)$, as implemented in App. 7A, using stream 1 with the default seed. We took $k = 2^{12} = 4096$ (so that the most significant 12 bits of the U_i's are being examined for uniformity) and let $n = 2^{15} = 32{,}768$. We obtained $\chi^2 = 4141.0$; using the above approximation for the critical point, $\chi^2_{4095,0.90} \approx 4211.4$, so the null hypothesis of uniformity is not rejected at level $\alpha = 0.10$. Therefore, *these particular* 32,768 U_i's produced by this generator do not behave in a way that is significantly different from what would be expected from truly IID U(0, 1) random variables, so far as this chi-square test can ascertain.

Our second empirical test, the *serial test*, is really just a generalization of the chi-square test to higher dimensions. If the U_i's were really IID U(0, 1) random variates, the nonoverlapping d-tuples

$$\mathbf{U}_1 = (U_1, U_2, \ldots, U_d), \qquad \mathbf{U}_2 = (U_{d+1}, U_{d+2}, \ldots, U_{2d}), \qquad \ldots$$

should be IID random *vectors* distributed uniformly on the d-dimensional unit hypercube, $[0, 1]^d$. Divide [0, 1] into k subintervals of equal size and generate $\mathbf{U}_1, \mathbf{U}_2, \ldots, \mathbf{U}_n$ (requiring nd U_i's). Let $f_{j_1 j_2 \ldots j_d}$ be the number of $\mathbf{U}_i$'s having first component in subinterval j_1, second component in subinterval j_2, etc. (It is easier to tally the $f_{j_1 j_2 \ldots j_d}$'s than might be expected; see Prob. 7.7.) If we let

$$\chi^2(d) = \frac{k^d}{n} \sum_{j_1=1}^{k} \sum_{j_2=1}^{k} \cdots \sum_{j_d=1}^{k} \left(f_{j_1 j_2 \ldots j_d} - \frac{n}{k^d} \right)^2$$

then $\chi^2(d)$ will have an approximate chi-square distribution with $k^d - 1$ df. [See L'Ecuyer, Simard, and Weggenkittl (2002) for further discussion of the serial test.] The test for d-dimensional uniformity is carried out exactly as for the one-dimensional chi-square test above.

EXAMPLE 7.7. For $d = 2$, we tested the null hypothesis that the pairs (U_1, U_2), $(U_3, U_4), \ldots, (U_{2n-1}, U_{2n})$ are IID random vectors distributed uniformly over the unit square. We used the generator in App. 7A, but starting with stream 2, and generated $n = 32{,}768$ pairs of U_i's. We took $k = 64$, so that the degrees of freedom were again $4095 = 64^2 - 1$ and the level $\alpha = 0.10$ critical value was the same, 4211.4. The value of $\chi^2(2)$ was 4016.5, indicating acceptable uniformity in two dimensions for the first two-thirds of stream 2 (recall from Sec. 2.3 that the streams are of length 100,000 U_i's, and we used $2n = 65{,}536$ of them here). For $d = 3$, we used stream 3, took $k = 16$ (keeping the degrees of freedom as $4095 = 16^3 - 1$ and the level $\alpha = 0.10$ critical value at 4211.4), and generated $n = 32{,}768$ nonoverlapping *triples of* U_i's. And $\chi^2(3)$ was 4174.5, again indicating acceptable uniformity in three dimensions.

Why should we care about this kind of uniformity in *higher* dimensions? If the individual U_i's are correlated, the distribution of the d-vectors $\mathbf{U}_i$ will deviate from d-dimensional uniformity; thus, the serial test provides an indirect check on the assumption that the individual U_i's are independent. For example, if adjacent U_i's tend to be positively correlated, the pairs (U_i, U_{i+1}) will tend to cluster around the southwest-northeast diagonal in the unit square, and $\chi^2(2)$ should pick this up. Finally, it should be apparent that the serial test for $d > 3$ could require a lot of memory to tally the k^d values of $f_{j_1 j_2 \ldots j_d}$. (Choosing $k = 16$ in Example 7.7 when $d = 3$ is probably not a sufficiently fine division of [0, 1].)

The third empirical test we consider, the *runs* (or *runs-up*) *test*, is a more direct test of the independence assumption. (In fact, it is a test of independence *only*; i.e., we are not testing for uniformity in particular.) We examine the U_i sequence (or, equivalently, the Z_i sequence) for unbroken subsequences of maximal length within which the U_i's increase monotonically; such a subsequence is called a *run up*. For example, consider the following sequence $U_1, U_2, \ldots, U_{10}$: 0.86, 0.11, 0.23, 0.03, 0.13, 0.06, 0.55, 0.64, 0.87, 0.10. The sequence starts with a run up of length 1 (0.86), followed by a run up of length 2 (0.11, 0.23), then another run up of length 2 (0.03, 0.13), then a run up of length 4 (0.06, 0.55, 0.64, 0.87), and finally another run up of length 1 (0.10). From a sequence of n U_i's, we count the number of runs up of length 1, 2, 3, 4, 5, and ≥ 6, and then define

$$r_i = \begin{cases} \text{number of runs up of length } i & \text{for } i = 1, 2, \ldots, 5 \\ \text{number of runs up of length } \geq 6 & \text{for } i = 6 \end{cases}$$

(See Prob. 7.8 for an algorithm to tally the r_i's. For the 10 U_i's above, $r_1 = 2$, $r_2 = 2$, $r_3 = 0$, $r_4 = 1$, $r_5 = 0$, and $r_6 = 0$.) The test statistic is then

$$R = \frac{1}{n} \sum_{i=1}^{6} \sum_{j=1}^{6} a_{ij} (r_i - n b_i)(r_j - n b_j)$$

where a_{ij} is the (i, j)th element of the matrix

$$\begin{bmatrix} 4{,}529.4 & 9{,}044.9 & 13{,}568 & 18{,}091 & 22{,}615 & 27{,}892 \\ 9{,}044.9 & 18{,}097 & 27{,}139 & 36{,}187 & 45{,}234 & 55{,}789 \\ 13{,}568 & 27{,}139 & 40{,}721 & 54{,}281 & 67{,}852 & 83{,}685 \\ 18{,}091 & 36{,}187 & 54{,}281 & 72{,}414 & 90{,}470 & 111{,}580 \\ 22{,}615 & 45{,}234 & 67{,}852 & 90{,}470 & 113{,}262 & 139{,}476 \\ 27{,}892 & 55{,}789 & 83{,}685 & 111{,}580 & 139{,}476 & 172{,}860 \end{bmatrix}$$

and the b_i's are given by

$$(b_1, b_2, \ldots, b_6) = \left(\tfrac{1}{6}, \tfrac{5}{24}, \tfrac{11}{120}, \tfrac{19}{720}, \tfrac{29}{5040}, \tfrac{1}{840}\right)$$

[See Knuth (1998a, pp. 66–69) for derivation of these constants.* The a_{ij}'s given above are accurate to five significant digits.] For large n (Knuth recommends $n \geq 4000$), R will have an approximate chi-square distribution with 6 df, under the null hypothesis that the U_i's are IID random variables.

EXAMPLE 7.8. We subjected stream 4 of the generator in App. 7A to the runs-up test, using $n = 5000$, and obtained $(r_1, r_2, \ldots, r_6) = (808, 1026, 448, 139, 43, 4)$, leading to a value of $R = 9.3$. Since $\chi^2_{6,0.90} = 10.6$, we do not reject the hypothesis of independence at level $\alpha = 0.10$.

The runs-up test can be reversed in the obvious way to obtain a runs-down test; the a_{ij} and b_i constants are the same. There are several other kinds of runs tests, such as counting runs up or down in the same sequence, or simply counting the number of runs without regard to their length; we refer the reader to Banks et al. (2001, pp. 270–278) and Fishman (1978, pp. 373–376), for example. Recall as well our discussion of runs tests in Sec. 6.3. Since runs tests look solely for independence (and not specifically for uniformity), it would probably be a good idea to apply a runs test *before* performing the chi-square or serial tests, since the last two tests implicitly assume independence.

The final type of empirical test we consider is a direct way to assess whether the generated U_i's exhibit discernible correlation: Simply compute an estimate of the correlation at lags $j = 1, 2, \ldots, l$ for some value of l. Recall from Sec. 4.3 that the correlation at lag j in a sequence $X_1, X_2, \ldots$ of random variables is defined as $\rho_j = C_j/C_0$, where

$$C_j = \mathrm{Cov}(X_i, X_{i+j}) = E(X_i X_{i+j}) - E(X_i)E(X_{i+j})$$

is the covariance between entries in the sequence separated by j; note that $C_0 = \mathrm{Var}(X_i)$. (It is assumed here that the process is covariance-stationary; see Sec. 4.3.) In our case, we are interested in $X_i = U_i$, and under the hypothesis that the U_i's are uniformly distributed on $[0, 1]$, we have $E(U_i) = \frac{1}{2}$ and $\mathrm{Var}(U_i) = \frac{1}{12}$, so that $C_j = E(U_i U_{i+j}) - \frac{1}{4}$ and $C_0 = \frac{1}{12}$. Thus, $\rho_j = 12E(U_i U_{i+j}) - 3$ in this case. From a sequence $U_1, U_2, \ldots, U_n$ of generated values, an estimate of ρ_j can thus be obtained by estimating $E(U_i U_{i+j})$ directly from U_1, U_{1+j}, U_{1+2j}, etc., to obtain

$$\hat{\rho}_j = \frac{12}{h+1} \sum_{k=0}^{h} U_{1+kj} U_{1+(k+1)j} - 3$$

where $h = \lfloor (n-1)/j \rfloor - 1$. Under the further assumption that the U_i's are independent, it turns out [see, for example, Banks et al. (2001, p. 279)] that

$$\mathrm{Var}(\hat{\rho}_j) = \frac{13h + 7}{(h+1)^2}$$

Under the null hypothesis that $\rho_j = 0$ and assuming that n is large, it can be shown that the test statistic

$$A_j = \frac{\hat{\rho}_j}{\sqrt{\mathrm{Var}(\hat{\rho}_j)}}$$

has an approximate standard normal distribution. This provides a test of zero lag j correlation at level α, by rejecting this hypothesis if $|A_j| > z_{1-\alpha/2}$. The test should probably be carried out for several values of j, since it could be, for instance, that there is no appreciable correlation at lags 1 or 2, but there is dependence between the U_i's at lag 3, due to some anomaly of the generator.

EXAMPLE 7.9. We tested streams 5 through 10 of the generator in App. 7A for correlation at lags 1 through 6, respectively, taking $n = 5000$ in each case; i.e., we tested stream 5 for lag 1 correlation, stream 6 for lag 2 correlation, etc. The values of $A_1, A_2, \ldots, A_6$ were 0.90, −1.03, −0.12, −1.32, 0.39, and 0.76, respectively, none of which is significantly different from 0 in comparison with the N(0, 1) distribution, at level $\alpha = 0.10$ (or smaller). Thus, the first 5000 values in these streams do not exhibit observable autocorrelation at these lags.

As mentioned above, these are just four of the many possible empirical tests. For example, the Kolmogorov-Smirnov test discussed in Sec. 6.6.2 (for the case with all parameters known) could be applied instead of the chi-square test for one-dimensional uniformity. Several empirical tests have been developed around the idea that the generator in question is used to simulate a relatively simple stochastic system with *known* (population) performance measures that the simulation estimates. The simulated results are compared in some way with the known exact "answers," perhaps by means of a chi-square test. A simple application of this idea is seen in Prob. 7.10; Rudolph and Hawkins (1976) test several generators by using them to simulate Markov processes. In general, we feel that as many empirical tests should be performed as are practicable. In this regard, it should be mentioned that there are several comprehensive test suites for evaluating random-number generators. These include DIEHARD [Marsaglia (1995)], the NIST Test Suite [Rukhin et al. (2001)], and *TestU01* [L'Ecuyer and Simard (2005)], with the last package containing more than 60 empirical tests. These test suites are also described in Gentle (2003, pp. 79–85).

Lest the reader be left with the impression that the empirical tests we have presented in this section have no discriminative power at all, we subjected the infamous generator RANDU [defined by $Z_i = 65{,}539 Z_{i-1} (\mathrm{mod}\ 2^{31})$], with seed

*Knuth, D. E., *The Art of Computer Programming*, Vol. 2, p. 67, © 1998, 1981 Pearson Education, Inc. Reproduced by permission of Pearson Education, Inc. All rights reserved.

$Z_0 = 123{,}456{,}789$, to the same tests as in Examples 7.6 through 7.9. The test statistics were as follows:

Chi-square test:	χ^2	= 4,202.0
Serial tests:	$\chi^2(2)$	= 4,202.3
	$\chi^2(3)$	= 16,252.3
Runs-up test:	R	= 6.3
Correlation tests:	All A_j's were insignificant	

While uniformity appears acceptable on [0, 1] and on the unit square, note the enormous value of the three-dimensional serial test statistic, indicating a severe problem for this generator in terms of uniformity on the unit cube. RANDU is a fatally flawed generator, due primarily to its utter failure in three dimensions; we shall see why in Sec. 7.4.2.

One potential disadvantage of empirical tests is that they are only *local*; i.e., only that segment of a cycle (for LCGs, for example) that was actually used to generate the U_i's for the test is examined, so we cannot say anything about how the generator might perform in other segments of the cycle. On the other hand, this local nature of empirical tests can be advantageous, since it might allow us to examine the actual random numbers that will be used later in a simulation. (Often we can calculate ahead of time how many random numbers will be used in a simulation, or at least get a conservative estimate, by analyzing the model's operation and the techniques used for generating the necessary random variates.) Then this entire random-number stream can be tested empirically, one would hope without excessive cost. (The tests in Examples 7.6 through 7.9 were all done together in a single program that took just a few seconds on an old, modest computer.) A more global empirical test could be performed by replicating an entire test several times and statistically comparing the observed values of the test statistics against the distribution under the null hypothesis; Fishman (1978, pp. 371–372) suggests this approach. For example, the runs-up test of Example 7.8 could be done, say, 100 times using 100 separate random-number streams from the same generator, each of length 5000. This would result in 100 independent values for R, which could then be compared with the chi-square distribution with 6 df using, for example, the K-S test with all parameters known. Fishman's approach could be used to identify "bad" segments within a cycle of a LCG; this was done for the PMMLCG with $m = 2^{31} - 1$ and $a = 630{,}360{,}016$, and a "bad" segment was indeed discovered and eliminated from use in SIMSCRIPT II.5 [see Fishman (1973a, p. 183)]. However, we would expect that even a "perfect" random-number generator would occasionally produce an "unacceptable" test statistic; in fact this *ought* to happen with probability α = the level of the test being done. Thus, it can be argued that such hand-picking of segments to avoid "bad" ones is in fact a poor idea.

7.4.2 Theoretical Tests

We now discuss theoretical tests for random-number generators. Since these tests are quite sophisticated and mathematically complex, we shall describe them somewhat qualitatively; for detailed accounts see Fishman (1996, pp. 607–628), Knuth (1998a, pp. 80–115), and L'Ecuyer (1998, pp. 106–114). As mentioned earlier, theoretical tests do not require that we generate any U_i's at all but are *a priori* in that they indicate how well a generator *can* perform by looking at its structure and defining constants. Theoretical tests also differ from empirical tests in that they are *global*; i.e., a generator's behavior over its *entire* cycle is examined. As we mentioned at the end of Sec. 7.4.1, it is debatable whether local or global tests are preferable; global tests have a natural appeal but do not generally indicate how well a specific segment of a cycle will behave.

It is sometimes possible to compute the "sample" mean, variance, and correlations over an entire cycle directly from the constants defining the generator. Many of these results are quoted by Kennedy and Gentle (1980, pp. 139–143). For example, in a full-period LCG, the average of the U_i's, taken over an entire cycle, is $\frac{1}{2} - 1/(2m)$, which is seen to be very close to the desired $\frac{1}{2}$ if m is one of the large values (in the billions) typically used; see Prob. 7.14. Similarly, we can compute the "sample" variance of the U_i's over a full cycle and get $\frac{1}{12} - 1/(12m^2)$, which is close to $\frac{1}{12}$, the variance of the U(0, 1) distribution. Kennedy and Gentle also discuss "sample" correlations for LCGs. Although such formulas may seem comforting, they can be misleading; e.g., the result for the full-period LCG sample lag 1 correlation suggests that, to minimize this value, a be chosen close to $\sqrt{m}$, which turns out to be a poor choice from the standpoint of other important statistical considerations.

The best-known theoretical tests are based on the rather upsetting observation by Marsaglia (1968) that "random numbers fall mainly in the planes." That is, if U_1, $U_2, \ldots$ is a sequence of random numbers generated by a LCG, the overlapping d-tuples $(U_1, U_2, \ldots, U_d), (U_2, U_3, \ldots, U_{d+1}), \ldots$ will all fall on a relatively small number of $(d-1)$-dimensional hyperplanes passing through the d-dimensional unit hypercube $[0, 1]^d$. For example, if $d = 2$, the pairs (U_1, U_2), (U_2, U_3), $\ldots$ will be arranged in a "lattice" fashion along several different families of parallel lines going through the unit square. The lines within a family are parallel to each other, but lines from different families are not parallel.

In Figs. 7.2 and 7.3, we display all possible pairs (U_i, U_{i+1}) for the full-period multiplicative LCGs $Z_i = 18Z_{i-1} \pmod{101}$ and $Z_i = 2Z_{i-1} \pmod{101}$, respectively. (The period is 100 in each case, since the modulus $m = 101$ is prime and each value of the multiplier a is a primitive element modulo m.) While the apparent regularity in Fig. 7.2 certainly does not seem very "random," it may not be too disturbing since the pairs seem to fill up the unit square fairly well, or at least as well as could be expected with such a small modulus. On the other hand, all 100 pairs in Fig. 7.3 fall on just two parallel lines, which is definitely anxiety-provoking. Since there are large areas of the unit square where we can never realize a pair of U_i's, a simulation using such a generator would almost certainly produce invalid results.

The same difficulties occur in three dimensions. In Fig. 7.4, 2000 triples (U_i, U_{i+1}, U_{i+2}) produced by the infamous multiplicative LCG generator RANDU ($m = 2^{31}$ and $a = 65{,}539$) are displayed, viewed from a particular point outside of the unit cube. Note that all triples of U_i's fall on *only* 15 parallel planes passing through the unit cube. (In general, all the roughly half-billion triples across the period fall on these same planes.) This explains the horrific performance of RANDU on the three-dimensional serial test noted at the end of Sec. 7.4.1.

Among all families of parallel hyperplanes that cover all overlapping d-tuples $(U_i, U_{i+1}, \ldots, U_{i+d-1})$, take the one for which adjacent hyperplanes are farthest

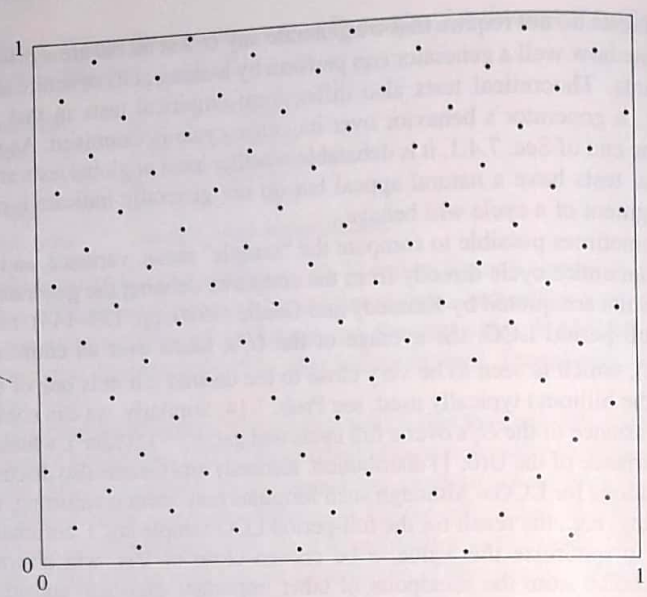

FIGURE 7.2
Two-dimensional lattice structure for the full-period LCG with $m = 101$ and $a = 18$.

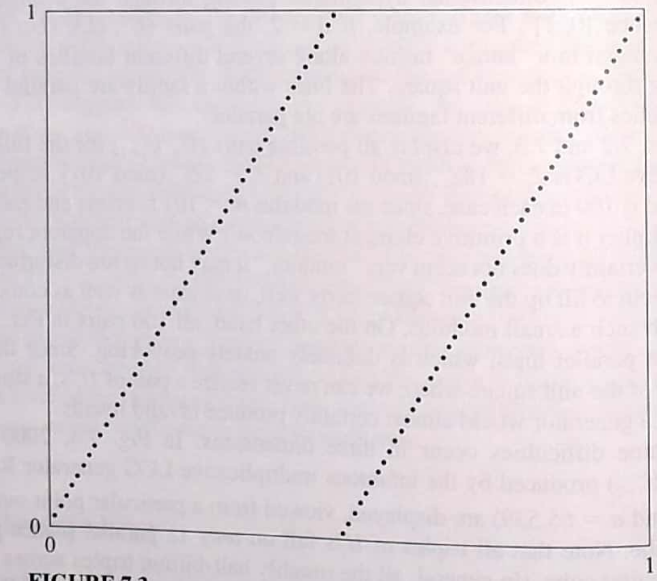

FIGURE 7.3
Two-dimensional lattice structure for the full-period LCG with $m = 101$ and $a = 2$.

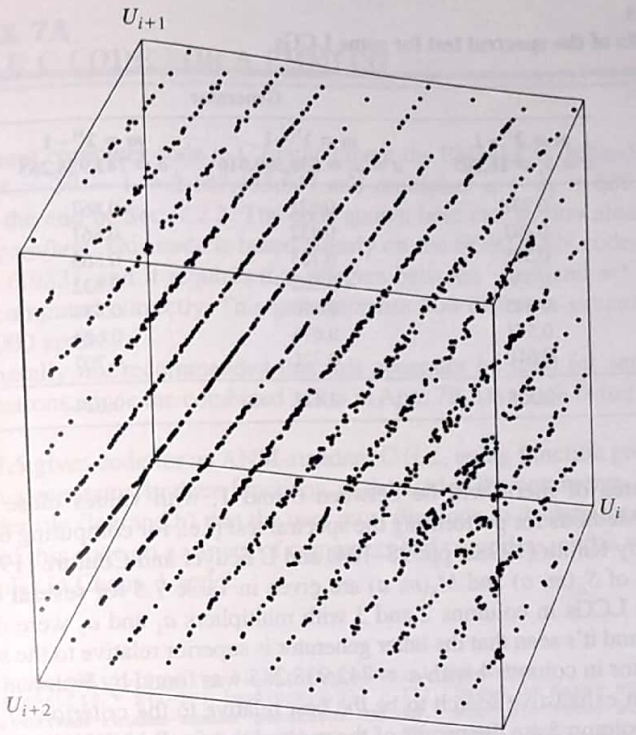

FIGURE 7.4
Three-dimensional lattice structure for 2000 triples from the multiplicative LCG RANDU with $m = 2^{31}$ and $a = 65{,}539$.

apart and denote this distance by $\delta_d(m, a)$. The idea of computing $\delta_d(m, a)$ was suggested by Coveyou and MacPherson (1967) and is typically called the *spectral test*. If $\delta_d(m, a)$ is small, then we would expect that the corresponding generator would be able to uniformly fill up the d-dimensional unit hypercube $[0, 1]^d$.

For LCGs it is possible to compute a theoretical lower bound $\delta_d^*(m)$ on $\delta_d(m, a)$, which is given by the following:

$$\delta_d(m, a) \geq \delta_d^*(m) = \frac{1}{\gamma_d m^{1/d}} \quad \text{for all } a$$

where γ_d is a constant whose exact value is known only for $d \leq 8$. We can then define the following figures of merit for a LCG:

$$S_d(m, a) = \frac{\delta_d^*(m)}{\delta_d(m, a)}$$

and

$$M_8(m, a) = \min_{2 \leq d \leq 8} S_d(m, a)$$

TABLE 7.5
The results of the spectral test for some LCGs

Figure of merit	Generator $m = 2^{31}-1$ $a = a_1 = 16{,}807$	$m = 2^{31}-1$ $a = a_2 = 630{,}360{,}016$	$m = 2^{31}-1$ $a = 742{,}938{,}285$	$m = 2^{31}$ $a = 65{,}539$
$S_2(m, a)$	0.338	0.821	0.867	0.931
$S_3(m, a)$	0.441	0.432	0.861	0.012
$S_4(m, a)$	0.575	0.783	0.863	0.060
$S_5(m, a)$	0.736	0.802	0.832	0.157
$S_6(m, a)$	0.645	0.570	0.834	0.293
$S_7(m, a)$	0.571	0.676	0.624	0.453
$S_8(m, a)$	0.610	0.721	0.707	0.617
$M_8(m, a)$	0.338	0.432	0.624	0.012

These figures of merit will be between 0 and 1, with values close to 1 being desirable. Methods for performing the spectral test [i.e., for computing $\delta_d(m, a)$] are discussed by Knuth (1998a, pp. 98–104) and L'Ecuyer and Couture (1997).

Values of $S_d(m, a)$ and $M_8(m, a)$ are given in Table 7.5 for several well-known LCGs. The LCGs in columns 2 and 3 with multipliers a_1 and a_2 were discussed in Sec. 7.2.2, and it's seen that the latter generator is superior relative to the spectral test. The generator in column 4 with $a = 742{,}938{,}285$ was found by Fishman and Moore (1986) in an exhaustive search to be the best relative to the criterion $M_6(2^{31}-1, a)$. Finally, in column 5 are the results of the spectral test for RANDU; its poor behavior in $d = 3$ dimensions is clearly reflected in the value $S_3(2^{31}, 65{,}539) = 0.012$.

Other figures of merit have been suggested to measure the quality of a random-number generator in terms of its lattice structure. These include the lattice test [see Beyer et al. (1971), L'Ecuyer and Couture (1997), and Marsaglia (1972)] and computing the minimum number of hyperplanes that contain all d-tuples $(U_i, U_{i+1}, \ldots, U_{i+d-1})$ [see Fishman (1996, pp. 617–620)].

7.4.3 Some General Observations on Testing

The number, variety, and range of complexity of tests for random-number generators are truly bewildering. To make matters worse, there has been (and probably always will be) considerable controversy over which tests are best, whether theoretical tests are really more definitive than empirical tests, and so on. Indeed, no amount of testing can ever absolutely convince everyone that some particular generator is absolutely "the best." One piece of advice that is often offered, however, is that a random-number generator should be tested in a way that is consistent with its intended use. This would entail, for example, examining the behavior of pairs of U_i's (perhaps with the two-dimensional serial test) if random numbers are naturally used in pairs in the simulation itself. In a broader sense, this advice would imply that one should be more careful in choosing and testing a random-number generator if the simulation in which it will be used is very costly, requires high-precision results, or is a particularly critical component of a larger study.

APPENDIX 7A
PORTABLE C CODE FOR A PMMLCG

Here we present computer code in C to implement the PMMLCG defined by modulus $m = m^* = 2^{31} - 1 = 2{,}147{,}483{,}647$ and multiplier $a = a_2 = 630{,}360{,}016$, discussed at the end of Sec. 7.2.2. The code shown here can be downloaded from www.mhhe.com/law. This code is based closely on the FORTRAN code of Marse and Roberts (1983), and it requires that integers between $-m^*$ and m^* be represented and computed correctly. This generator has 100 different streams that are spaced 100,000 apart.

It is generally *not* recommended that this generator be used for serious real-world applications, since the combined MRG in App. 7B has much better statistical properties.

Figure 7.5 gives code for an ANSI-standard C (i.e., using function prototyping) version of this generator, in three functions, as detailed in the comments. Figure 7.6 gives a header file (lcgrand.h) that the user must #include to declare the functions. We have used this code on a variety of computers and compilers, and it was used in the C examples in Chaps. 1 and 2.

```
/* Prime modulus multiplicative linear congruential generator
   Z[i] = (630360016 * Z[i-1]) (mod(pow(2,31) - 1)), based on Marse and Roberts'
   portable FORTRAN random-number generator UNIRAN.  Multiple (100) streams are
   supported, with seeds spaced 100,000 apart.  Throughout, input argument
   "stream" must be an int giving the desired stream number.  The header file
   lcgrand.h must be included in the calling program (#include "lcgrand.h")
   before using these functions.

   Usage: (Three functions)

   1. To obtain the next U(0,1) random number from stream "stream," execute
         u = lcgrand(stream);
      where lcgrand is a float function.  The float variable u will contain the
      next random number.

   2. To set the seed for stream "stream" to a desired value zset, execute
         lcgrandst(zset, stream);
      where lcgrandst is a void function and zset must be a long set to the
      desired seed, a number between 1 and 2147483646 (inclusive).  Default
      seeds for all 100 streams are given in the code.

   3. To get the current (most recently used) integer in the sequence being
      generated for stream "stream" into the long variable zget, execute
         zget = lcgrandgt(stream);
      where lcgrandgt is a long function. */

/* Define the constants. */

#define MODLUS 2147483647
#define MULT1        24112
#define MULT2        26143

/* Set the default seeds for all 100 streams. */
```

FIGURE 7.5
C code for the PMMLCG with $m = 2^{31} - 1$ and $a = 630{,}360{,}016$ based on Marse and Roberts (1983).