times in the system measured in the two simulation runs are significantly different. If the test shows that the two results are not significantly different, the two number sequences are interchangeable for the purposes of the simulation.

It is easy to see that an algorithm can generate a sequence of numbers that has precisely the same statistics as a sequence of numbers generated by a natural random process, because in principle an algorithm can be made to produce any given number sequence of finite length. This sequence could be identical to the numbers on the tape previously mentioned, so it would obviously be statistically indistinguishable from that sequence, which was generated by a natural random process. Of course, it is not necessary to replicate the exact numbers appearing on the tape, because the statistical properties can be the same even if the sequences of numbers are not, and the sequences can therefore be interchangeable in the sense defined above.

A sequence of numbers that is generated by an algorithm but is interchangeable with a truly random sequence is called pseudorandom.

> A sequence of numbers is **pseudorandom** if every sufficiently short subsequence is interchangeable with a comparable sequence that is truly random.

The only unavoidable difference between a truly random sequence and a pseudorandom sequence is that the sequence generated by an algorithm might repeat, and this accounts for the phrase "sufficiently short" in the definition. Just as rewinding a tape that contains a truly random sequence and starting over yields a longer sequence that is no longer random, permitting the pseudorandom sequence generated by an algorithm to repeat might result in a longer sequence that is no longer interchangeable with a truly random sequence. To ensure this does not happen, the period of repetition of the algorithm must be longer than the sequence of numbers required by the simulation.

Random-Number Generators

After carefully distinguishing between the random sequences produced by natural random processes and the pseudorandom sequences produced by algorithms, it is conventional to use terminology that ignores the distinction. This is because in practice one never uses truly random sequences; it is always pseudorandom sequences that are under discussion, so no confusion can arise. Thus, an algorithm for generating a pseudorandom sequence of numbers is commonly called simply a **random-number generator.** It is also common to speak of **random numbers** with the understanding that a pseudorandom sequence of numbers is actually being referred to, and to speak of a single number in a pseudorandom sequence as *a* random number even though randomness is a concept that really applies only to the sequence as a whole.

There are many algorithms for generating random numbers, differing from one another in the extent to which they exhibit the following desirable properties:

- Long period of repetition

As already mentioned, it is desirable to be able to complete a simulation without having the random numbers begin to repeat. In practice, if the sequence is long

enough, then even if it does repeat that might have a small enough effect so the simulation results remain valid.

- Apparent statistical independence of successive numbers

This is necessary to ensure that the pseudorandom sequence is interchangeable with a truly random one. Knowing any subsequence should not help to determine the number that comes next.

- Uniform distribution

Sometimes a simulation requires random interevent times that are drawn from a uniform probability distribution, that is, a distribution whose values are equally probable within a given range. Even when that is not true, however, it is convenient to have a random-number generator produce uniformly distributed values. This is because, as we shall see, values from a uniform distribution are easy to transform into values from any arbitrary probability distribution that might be desired.

- Speed

Real simulations typically require a great many random numbers, so the speed with which an algorithm can be executed might have a big influence on the computer time consumed. Usually, simplicity is the key to high speed.

- Repeatability

Although it is important for the sequence to be pseudorandom in the sense already defined, it should be possible to use the *same* pseudorandom sequence from one run of a simulation to the next. The main reason for this is so that it is possible to debug the computer program that performs the simulation. Debugging is always necessary in practice and is extremely difficult unless the same number stream can be used repeatedly.

It is easy to design a random-number generator so that its output is repeatable. However, it is always necessary to make trade-offs between period of repetition, independence of successive numbers, uniformity of distribution, and speed. In order to produce pseudorandom sequences having good statistical properties, it is inevitably necessary to sacrifice speed. Thus, the selection of a suitable random-number generator for a given simulation can be a nontrivial task.

The Multiplicative Congruential Algorithm

The simplest random-number generator is the multiplicative congruential algorithm.

MULTIPLICATIVE CONGRUENTIAL ALGORITHM

0. Fix parameters p and m
1. u_0 = starting number, nonzero and odd
   $k \leftarrow 0$
2. $u_{k+1} = (mu_k) \bmod p$
3. $k \leftarrow k + 1$
   Go to 2

All the numbers used in the algorithm are nonnegative integers. The parameter m is called the **multiplier,** the parameter p is called the **modulus,** and the starting number u_0 is called the **seed.** The heart of the algorithm is the **recursion formula**

$$u_{k+1} = (mu_k) \bmod p$$

telling how each number in the sequence is generated from the previous one. The notation "mod p" in this formula refers to the result of an integer division.

$$x \bmod p = \text{remainder left over after dividing } x \text{ by } p$$

If no u_k is zero, the integers that can be generated range from 1 through $p - 1$, so $(u_k - 1)$ ranges from 0 through $p - 2$ and corresponding real numbers r_k in the range [0, 1] can be obtained as

$$r_k = \frac{u_k - 1}{p - 2}$$

To see how the multiplicative congruential algorithm works, consider the following example:

$$p = 17$$
$$m = 5$$
$$u_0 = 7$$
$$u_1 = (5 \cdot 7) \bmod 17 = \text{remainder of integer division } \tfrac{35}{17} = 1$$

$$\begin{array}{r} 2 \\ 17\overline{)35} \\ \underline{34} \\ 1 \end{array} = 35 \bmod 17$$

$$u_2 = (5 \cdot 1) \bmod 17 = 5$$
$$u_3 = (5 \cdot 5) \bmod 17 = 8$$

Continuing the process yields the sequence

7, 1, 5, 8, 6, 13, 14, 2, 10, 16, 12, 9, 11, 4, 3, 15, 7, 1, 5, 8, . . .

└──────── period of repetition ────────┘

The repetition is obvious in this example, with a period of 16 numbers. The period is always less than the parameter p, so p should be chosen as the largest value that can conveniently be used. Usually, because of efficiency considerations, this choice is influenced by the length of the words in the computer on which the algorithm is to be run. In particular, if the computer word length is w bits and we chose $p = 2^w$, the multiplication in the recursion formula yields a two-word result, the low-order word of which is the product modulo 2^w. The choice of $p = 2^{w-1}$ permits the desired value to be found almost as easily, and it is therefore also frequently used. In either case the fact that no division is required to perform the modulus operation provides a considerable speed advantage over algorithms using other values of p.

It is possible for the period of repetition to be much less than p if the multiplier m is inauspiciously chosen, and to get the longest possible period, $p - 1$, m should be chosen in such a way that $m - 1$ is a multiple of every prime number that divides p without a remainder.

It is also essential that no integer multiple of m be equal to p, as shown by the following example:

$$p = 10$$
$$m = 5$$
$$u_0 = 5$$
$$u_1 = (5 \cdot 5) \bmod 10 = 5$$
$$u_2 = (5 \cdot 5) \bmod 10 = 5$$
$$u_3 = (5 \cdot 5) \bmod 10 = 5$$

In such cases the period of repetition is only one number long.

Finally, for the sequence generated to have good statistical properties, m should be close to the square root of p, and the sequence used in a simulation should be no longer than the square root of the period of repetition.

Even if these rules are followed in selecting p and m, it is quite possible for the sequence generated by the multiplicative congruential algorithm to have significant **serial correlations** (that is, poor apparent independence of the successive numbers) or to depart significantly from a uniform distribution. In practice, therefore, the choice of values for p and m must be guided by statistical analysis of the sequences actually produced. The most commonly used multiplicative congruential generator, which is available on many computer systems by the subroutine name RANDU, has $p = 2^{31} = 2147483648$ and $m = 2^{16} + 3 = 65539$. These choices yield an algorithm that has a period of 2^{29} numbers and runs very fast on computers having a word length of 32 bits. However, the sequence it produces has the property that each number is related to the two previous ones by a simple formula, and this strong serial correlation makes the sequence generated by RANDU unsuitable for many simulations.

Some other multiplicative congruential generators in common use are reported to have statistical properties much better than those of RANDU. Two that are available in widely used commercial scientific subroutine packages have the following parameters:

$p = 2147483648 = 2^{31}$		$p = 2147483647 = 2^{31} - 1$
	and	
$m = 302875106592253 = 13^{13}$		$m = 16807 = 7^5$

The generator on the right runs slower than RANDU because the modulus operation is much easier to perform using $p = 2^{31}$ than with $p = 2^{31} - 1$.

Other Uniform Random-Number Generators

Improved statistical properties can be obtained by adding a constant term to the multiplicative congruential recursion formula, as follows:

$$u_{k+1} = (a + mu_k) \bmod p$$

The number a is called the **increment,** and an algorithm that uses this recursion formula is called a **mixed** or **linear congruential** generator. As in the case of the

multiplicative congruential generator, p should be the largest value that can conveniently be used. The increment a should obviously be nonzero, but the other considerations involved in picking good values for a and m are rather complicated. One published algorithm uses the following parameter values:

$$p = 2147483648 = 2^{31}$$

$$m = 843314861 \approx p(\pi/8) + 5$$

$$a = 453816693 \approx p(3 - \sqrt{3})/6$$

This generator has excellent statistical properties, and although it runs slower than RANDU because of the extra addition in the recursion formula, it is faster on most machines with 32-bit words than multiplicative congruential generators having $p > 2^{32}$ or p not a power of 2.

Another commonly used random-number generator having good statistical properties is the **GPSS algorithm.** The idea of this algorithm can best be illustrated by an example using decimal integers, although it is actually carried out using binary numbers on a computer. Suppose for the sake of the example that it is desired to generate random numbers r_k from .0000 through .9999. We begin with a multiplier m, which is fixed, and a seed u_0.

$$m = 5167$$

$$u_0 = 3729$$

These are multiplied together to produce a product having eight digits.

$$mu_0 = 19267743$$

The middle four digits of this product are used to form the first random number r_1, and the rightmost four digits are used as u_1. Thus,

$$r_1 = .2677$$

$$u_1 = 7743$$

Next the product mu_1 is formed, its middle four digits are used as the digits of r_2, its rightmost four digits become u_2, and so forth. The GPSS generator gets its name from General Purpose System Simulator, a next-event simulation programming language discussed further in Section 13.5. The GPSS language is widely used, so many simulations generate random numbers this way.

The final category of random-number generators in wide use is the **generalized shift register** algorithms. These algorithms were developed by computational physicists for use in Monte Carlo simulations of the kind discussed in Section 13.4. In many such applications successive numbers are used as random coordinates in spaces of high dimension, and the sequences produced by otherwise acceptable multiplicative congruential generators were found to exhibit undesirable patterns when used in that way. Also, the periods of repetition of multiplicative congruential generators were found to be far too short for many Monte Carlo simulations. The generalized shift register approach avoids both shortcomings by using recursions in which u_k is determined from several previous values, extending back to u_{k-q}, rather than from only u_{k-1}. Results from number theory are used to select recursions having the maximum possible period of repetition, which is $2^q - 1$. One popular variant, called the R250 algorithm,

has $q = 250$ and uses the recursion

$$u_k = u_{k-147} \oplus u_{k-250} \qquad k = 251 \ldots 250 + K$$

to generate K random numbers.

The symbol $\oplus$ in the recursion formula denotes the **exclusive-or** logical operation, applied to the bits of u_{k-147} and u_{k-250} when they are written as binary numbers. The exclusive-or operation is defined by the following table in which the marginal entries are the possible operand bits and the entries inside the table are the result bits obtained for the various input combinations:

x \ y	0	1
0	0	1
1	1	0

$x \oplus y$

exclusive-or operation

The exclusive-or returns 1 if the input bits are different and 0 if they are the same. To see how the exclusive-or is used in a typical iteration of the R250 algorithm, suppose that at step k of the algorithm we have $u_{k-147} = 369$ and $u_{k-250} = 811$. To find u_k according to the recursion formula, we (or rather the computer) would proceed as follows:

$$\begin{aligned} u_{k-147} &= 369_{10} = 000101110001_2 \\ &\qquad\qquad \oplus \\ u_{k-250} &= 811_{10} = 001100101011_2 \\ &\qquad\qquad\qquad\qquad\qquad\qquad \overline{001001011010_2 = 602_{10} = u_k} \end{aligned}$$

Usually, of course, computers have words longer than the 12 bits used in this simplified example.

The 250 most recently generated values must be retained at each step of the R250 algorithm, so that subsequent steps will always have the values they require. This vector of the 250 previous u_k's can be thought of as a shift register whose contents are shifted left one element per iteration of the algorithm, and this accounts for the name. The first 250 values in the shift register may be thought of as seeds, and must be generated as part of initializing R250 by using another method such as a multiplicative congruential algorithm. The R250 algorithm achieves its maximum possible period of repetition of $2^{250} - 1$ ($\approx 2 \cdot 10^{75}$), and has been found empirically to have good statistical properties. The shifting of the shift register takes some computer time not required by the other algorithms we have discussed, and this algorithm also uses more storage, but the recursion itself runs very fast and the extremely long period makes R250 the algorithm of choice for some applications.

Nonuniform Distributions

Often it is necessary to provide interevent times that are randomly drawn from some probability distribution other than the uniform distribution. Various

ad hoc techniques can be used to obtain random numbers from nonuniform distributions, but only one method is both universally applicable and widely used. That is the **method of inverse transformation.** The idea is to generate a random number from a uniform distribution and then calculate, from a formula (or a graph or table), a corresponding value that appears to be drawn from the desired distribution. The process is shown graphically in Figure 13.5.

To use the method of inverse transformation, we generate values u of a random variable U that is uniformly distributed on the interval [0, 1]. For each value u, we read across on the graph of Figure 13.5 until we meet the curve, and then read down to obtain a corresponding value x for the random variable X, which has the distribution we want. In other words, if the curve is a graph of the function $U = F(X)$, we calculate x as $x = F^{-1}(u)$.

For a hypothetical distribution function F, if $x = F^{-1}(u)$, then

$$P[X \leq x] = P[F^{-1}(U) \leq x] = P[U \leq F(x)]$$

The last equality above holds because the two probabilities are both represented by the shaded area in Figure 13.5. Next we observe that, by the definition of a uniformly distributed random variable,

$$P[U \leq F(x)] = F(x)$$

Thus $P[X \leq x] = F(x)$, and we see that the distribution function of X is in fact equal to $F(x)$. That is,

> if U is a random variable uniformly distributed on [0, 1],
> then $X = F^{-1}(U)$ is a random variable having the distribution function $F(x)$.

As an example of the method of inverse transformation, consider the problem of transforming uniformly distributed random numbers into random numbers drawn from an exponential distribution with parameter c. This comes up all the time in the simulation of queueing systems because exponentially distributed interevent times are often observed (or at least assumed). If X is an exponentially distributed random variable, its distribution function is

$$F(x) = 1 - e^{-cx}$$

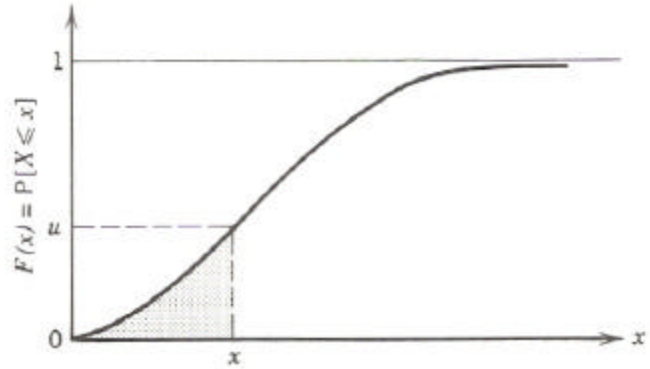

FIGURE 13.5 The method of inverse transformation.

Given a value u drawn at random from a distribution that is uniformly distributed on [0, 1], we can find the corresponding value of x by solving the equation $u = F(x)$ for x. If u is drawn from a distribution that is uniform on [0, 1], however, then we could just as well use $1 - u$ instead because it is also drawn from a distribution that is uniform on [0, 1]. Then it is easy to solve for x as follows:

$$\begin{aligned} 1 - u &= 1 - e^{-cx} \\ e^{-cx} &= u \\ -cx &= \ln(u) \\ x &= \frac{-\ln(u)}{c} = F^{-1}(1 - u) \end{aligned}$$

This result says that if U is distributed uniformly on [0, 1], then $X = -\ln(U)/c$ is distributed exponentially with parameter c. It also means that we can generate uniformly distributed random numbers and simply use the formula to get random numbers that are exponentially distributed with parameter c.

The method of inverse transformation can be used to transform uniformly distributed random numbers into random numbers from any distribution, so long as the inverse, F^{-1}, of the desired distribution can be found analytically or approximated numerically.

13.4 MONTE CARLO SIMULATIONS

In addition to next-event simulation, described in Section 13.2, there is another important technique, called **Monte Carlo simulation** or **static simulation,** in which random numbers and the laws of probability are used to approximate the solutions of problems that have nothing to do with queueing-type systems. Although this technique is referred to as simulation, it does not necessarily have any interpretation as the mechanical imitation of a real process. The name Monte Carlo derives from the fact that a city of that name is famous for its gambling casinos, and reflects the importance of random numbers in the method. Monte Carlo simulation, like dynamic programming, is a general approach rather than a specific algorithm. To give the flavor of this approach we shall discuss two of its standard applications, integration and optimization.

Evaluation of an Integral

Consider the problem of calculating the numerical value of the following definite integral:

$$I(a, b) = \int_a^b f(x)\, dx = \int_a^b \frac{\ln(x)}{(3 - x)}\, dx$$

This integral cannot be evaluated in closed form; that is, it cannot be expressed in terms of a finite number of elementary functions. One way of calculating $I(a, b)$ for given values of a and b would be to use a deterministic numerical procedure such as Simpson's rule. For $a = 1$ and $b = 2$, this yields $I(1, 2) \approx 0.294441$. Figure 13.6 shows a graph of the integrand function $f(x)$, with the area corresponding to $I(1, 2)$ shaded.

Another way of approximating the integral is to estimate what fraction of the rectangular box in Figure 13.6 is occupied by the shaded area, and then multiply that fraction by the known area of the box. The box can be chosen arbitrarily, so long as it contains the entire area corresponding to the value of the integral. For convenience the box in Figure 13.6 has for its base the interval on the x axis between the lower and upper limits of integration, and a height of 1, which makes the area of the box equal to 1.

To estimate the fraction of the box that is occupied by the shaded area, we can generate points (x, y) randomly located within the box, check each point to see whether it is above or below the graph of the function, and estimate the fraction of the box that is occupied by the shaded area from

$$\text{area fraction} \approx \frac{\text{number of random points } (x,\ y) \text{ having } y < f(x)}{\text{total number of random points tried}}$$

Each random point requires two random numbers, x and y. The x value needs to be uniformly distributed between 1 and 2, and the y value needs to be uniformly distributed between 0 and 1. We can generate a single stream of random numbers u that are uniformly distributed on [0, 1] and find the coordinates of the kth random point (x, y) from the following formulas:

$$x_k = 1 + u_{2k-1}$$

$$y_k = u_{2k}$$

Figure 13.6 shows the first 100 points obtained by using a multiplicative congruential algorithm to generate random numbers and the preceding formulas to find the point corresponding to each pair of random numbers. There are 32

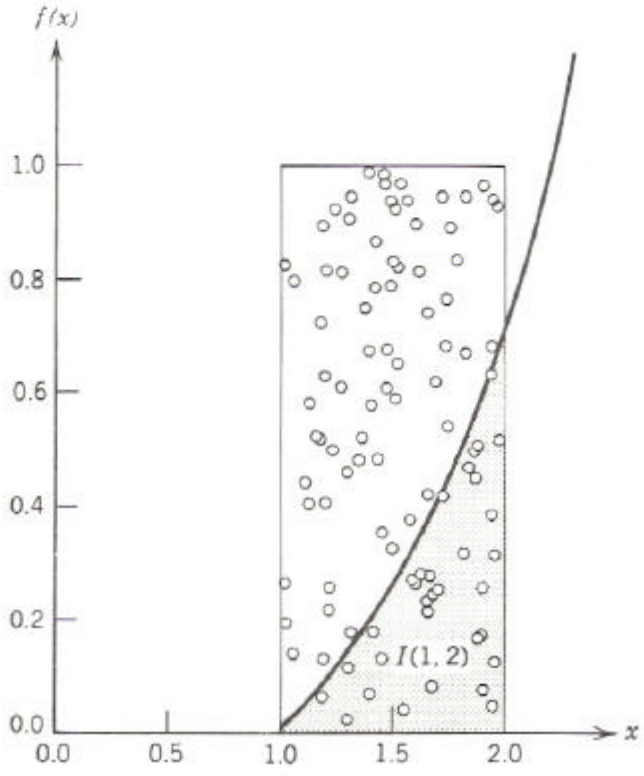

FIGURE 13.6 Monte Carlo integration.

points below the curve, so a crude estimate of the area fraction occupied by the shaded area is

$$\text{area fraction} = \tfrac{32}{100} = .32$$

Then the value of the integral is approximately

$$I(1, 2) = (\text{area fraction})\cdot(\text{area of box}) \approx (.32)\cdot(1) = .32$$

Using more random points yields the progressively better estimates given in the following table:

Total Number of Random Points (x, y)		Resulting Estimate of $I(1, 2)$
100		.320000
1000		.278000
10000		.299400
100000		.296040
1000000		.294538
True value	$\approx$	.294441

The estimate approaches (very slowly) the true value of the integral as the number of trial points increases.

One would never actually use Monte Carlo simulation to solve a simple problem like this example because a deterministic numerical method such as Simpson's rule gives a much more accurate result with much less computational work. For integrals in high dimensional spaces, however, the situation is often reversed. The integral value is then the volume of a region whose boundaries might be described by complicated formulas, so that it is difficult to determine what limits to use in a direct numerical integration. The need to integrate over many dimensions also greatly increases the computational work, so that a Monte Carlo estimate is often easier. Monte Carlo simulation for the evaluation of definite integrals (and for some closely related problems) is thus very important in fields such as nuclear physics and accounts for a large proportion of the supercomputer cycles devoted to scientific and engineering calculations today.

The Metropolis Algorithm

In 1953, N. Metropolis introduced a simple algorithm for simulating a collection of atoms in thermal equilibrium at a given temperature. Starting from some nonequilibrium initial arrangement of the atoms, the algorithm considers one atom at a time and computes the change in the energy of the system that would result from a small random displacement of that one atom. If the energy would decrease or stay the same, the displacement is accepted and the configuration with the displaced atom is used as the starting point for the next step. If the energy would increase, the displacement might still be accepted, with a probability that depends on the energy change proposed. The probability of accepting a higher-energy configuration decreases with decreasing temperature. For physical reasons Metropolis used the following probability formula:

$$\text{P(accept an energy increase of } \Delta E) = e^{-\Delta E/bT}$$