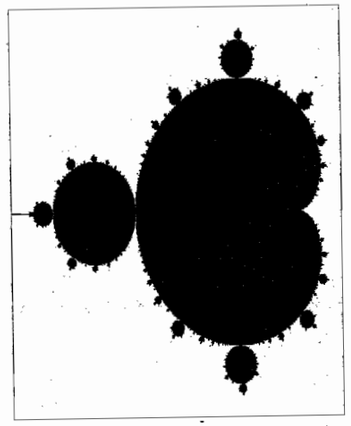

Figure 9.10 The Mandelbrot set is an example of a fractal. In this figure the lower left corner of the box represents the complex number $-1.5 - i$. The upper right corner of the box represents the complex number $0.5 + i$. Black points are in the set.

The magnitude of z is its distance from the origin; i.e., the length of the vector formed by its real and imaginary parts. If $z = a + bi$, the magnitude of z is $\sqrt{a^2 + b^2}$. If the magnitude of z ever becomes greater than or equal to 2, its subsequent values will grow without bound, and we know that c is not a point in the Mandelbrot set. If we iterate n times and find that the magnitude of z_n is still less than 2, we can conclude c is in the Mandelbrot set.

Your program should compute a Mandelbrot set for 600×600 evenly spaced points in a square region of the complex plane bounded by $-1.5 - i$ and $1 + i$. Let $n = 1{,}000$. If $z_{1000} < 2$, you should display point c as a member of the Mandelbrot set.

9.10 A **perfect number** is a positive integer whose value is equal to the sum of all its positive factors, excluding itself. The first two perfect numbers are 6 and 28:

$$6 = 1 + 2 + 3$$

$$28 = 1 + 2 + 4 + 7 + 14$$

The Greek mathematician Euclid (c. 300 BCE) showed that if $2^n - 1$ is prime, then $(2^n - 1)2^{n-1}$ is a perfect number. For example, $2^2 - 1 = 3$ is prime, so $(2^2 - 1)2^1 = 6$ is a perfect number. Write a parallel program to find the first eight perfect numbers.

CHAPTER

10

Monte Carlo Methods

O! many a shaft at random sent
Finds mark the archer little meant!
And many a word, at random spoken,
May soothe or wound a heart that's broken!
Sir Walter Scott, *The Lord of the Isles*

10.1 INTRODUCTION

A **Monte Carlo method** is an algorithm that solves a problem through the use of statistical sampling. The name is derived from the resort city in Monaco, famous for its games of chance. While early work in this field began in the nineteenth century, the first important use of Monte Carlo methods was for the development of the atomic bomb during World War II.

The Monte Carlo method is the only practical way to evaluate integrals of arbitrary functions in six or more dimensions. It has many other uses, including predicting the future level of the Dow Jones Industrial Average, solving partial differential equations, sharpening satellite images, modeling cell populations, and finding approximate solutions to NP-hard problems in polynomial time.

To illustrate the Monte Carlo method, let's begin with a physical analogy. Suppose we want to compute the value of π. We know that the area of a circle with diameter D is $\pi D^2/4$. We also know that the area of a square having sides of length D is D^2. Imagine slipping a round cake pan with diameter D inside a $D \times D$ cake pan and putting the pans out in the rain. After a few hours, we retrieve the pans and measure the amount of water in each. The ratio of the amount of water collected in the round pan to the total amount of water collected in both pans should be about $\pi/4$:

$$\frac{\pi D^2/4}{D^2} = \frac{\pi}{4}$$

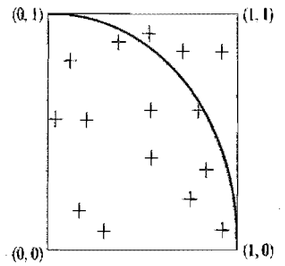

Figure 10.1 Using the Monte Carlo method to estimate the value of π. The area inside the quarter circle is $pi/4$. In this illustration, 12 of 15 points randomly chosen from the unit square are inside the circle, resulting in an estimate of 0.8 for $\pi/4$ or 3.2 for π.

We can use random numbers to perform a similar estimation. (This example shows the methodology, but keep in mind numerical integration is a better strategy when the number of dimensions is small.) Figure 10.1 illustrates a quarter circle with radius 1 embedded in a unit square. A complete circle with radius 1 has area π; hence the area of the quarter circle is $\pi/4$. We will generate a series of pairs (x, y), where both x and y are taken from a uniform random distribution between 0 and 1. Each pair represents a point inside the unit square. We keep track of the fraction f of points falling inside the quarter circle; that is, the points for which $x^2 + y^2 \leq 1$. Since $f \approx \pi/4$, we know $4f \approx \pi$.

We have implemented a C program to compute π using this methodology (Figure 10.2). Table 10.1 shows how the absolute error between the computed value of π and the actual value slowly decreases as the sample size n increases. (Given estimated value e and correct value a, the absolute error is $|e - a|/a$.) The function $1/(2\sqrt{n})$ closely approximates the absolute error of this Monte Carlo method.

10.1.1 Why Monte Carlo Works

The mean value theorem states that

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

where $\bar{f}$ represents the mean (average) value of $f(x)$ in the interval $[a, b]$. (See Figure 10.3.)

The Monte Carlo method estimates the value of I by evaluating $f(x_i)$ at n points selected from a uniform random distribution over $[a, b]$. The expected

Table 10.1 As the sample size increases, so does the accuracy of the estimated solution.

Sample size n	Estimate of π	Error	$1/(2\sqrt{n})$
10	2.40000	0.23606	0.15811
100	3.36000	0.06952	0.05000
1,000	3.14400	0.00077	0.01581
10,000	3.13920	0.00076	0.00500
100,000	3.14132	0.00009	0.00158
1,000,000	3.14006	0.00049	0.00050
10,000,000	3.14136	0.00007	0.00016
100,000,000	3.14154	0.00002	0.00005
1,000,000,000	3.14155	0.00001	0.00002

```
/*
 *   This C program uses the Monte Carlo method to
 *   compute the value of pi.
 */

#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
   int    count;          /* Points inside circle */
   int    i;
   int    n;              /* Number of samples */
   double pi;             /* Estimate of pi */
   unsigned short xi[3]; /* Random number seed */
   double x, y;           /* Point's coordinates */

   if (argc != 5) {
      printf ("Correct command line: ");
      printf ("%s <# samples> <seed0> <seed1> <seed2>\n",
         argv[0]);
      return -1;
   }
   n = atoi(argv[1]);
   for (i = 0; i < 3; i++)
      xi[i] = atoi(argv[i+2]);

   count = 0;
   for (i = 0; i < n; i++) {
      x = erand48(xi);
      y = erand48(xi);
      if (x*x+y*y <= 1.0) count++;
   }
   pi = 4.0 * (double) count / (double) n;
   printf ("Samples: %d   Estimate of pi: %7.5f\n", n, pi);
}
```

Figure 10.2 A C program computing π using the Monte Carlo method. The precision of the answer is related to the number of samples and the quality of the pseudo-random number generator.

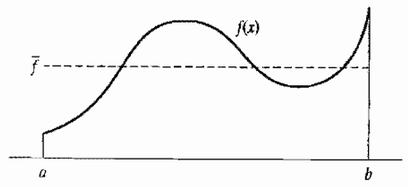

Figure 10.3 By the mean value theorem we know that the area under curve $f(x)$ is identical to the area under $\bar{f}$, where $\bar{f}$ is the mean value of $f(x)$ in the interval $[a, b]$.

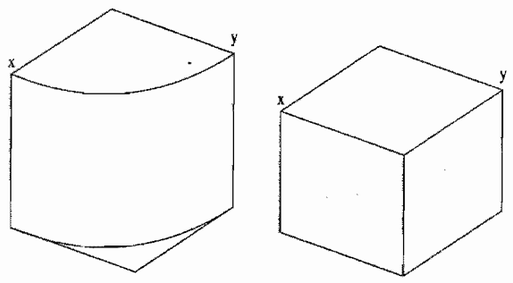

Figure 10.4 By the mean value theorem we know the volumes beneath both these surfaces are identical. (a) Within the square bounded by $0 \le x, y \le 1$, the height of the surface is 1 where $x^2 + y^2 \le 1$ and 0 otherwise. (b) Within the square bounded by $0 \le x, y \le 1$, the height of the surface is $\pi/4$.

value of $\frac{1}{n}\sum_{i=0}^{n-1} f(x_i)$ is $\bar{f}$. Hence

$$I = \int_a^b f(x)\,dx = (b-a)\bar{f} \approx (b-a)\frac{1}{n}\sum_{i=0}^{n-1} f(x_i)$$

Let's see how this applies to the π estimation algorithm we've already described. We know that the ratio in area between a quarter circle of radius 1 and a square having sides of length 1 is $\pi/4$. Consider the surface illustrated in Figure 10.4a. This surface has height 1 if $x^2 + y^2 \le 1$ and 0 otherwise. If we set an accumulator to 0, randomly generate pairs of points x, y from the unit square, add 1 to the accumulator if $x^2 + y^2 \le 1$ and add nothing to the accumulator if $x^2 + y^2 > 1$, we are sampling from this surface.

If, after n samples, we divide the contents of the accumulator by n, we produce a mean. The expected value of the calculated mean is $\pi/4$, as illustrated in Figure 10.4b. Hence multiplying the calculated mean by 4 yields a Monte Carlo estimate of π.

Importantly, the error in the Monte Carlo estimate of I decreases by a factor of $1/\sqrt{n}$. *The rate of convergence is independent of the dimension of the integrand.* This is in sharp contrast to deterministic numerical integration methods, such as Simpson's rule, which have a rate of convergence that decreases as the dimension increases. It explains why Monte Carlo techniques are superior to deterministic numerical integration methods when the integrand has more than about six dimensions.

10.1.2 Monte Carlo and Parallel Computing

Monte Carlo algorithms often migrate easily onto parallel systems. Many parallel Monte Carlo programs have a negligible amount of interprocessor communications. When this is the case, p processors can be used either to find an estimate about p times faster or to reduce the error of the estimate by a factor of $\sqrt{p}$. Another way of expressing the second point is to say that p processes can reduce the variance of the answer by a factor of p.

Of course, these levels of improvement are based on the assumption that the random numbers are statistically independent. A principal challenge in the development of parallel Monte Carlo methods has been the development of good parallel random number generators. It is widely claimed that half of all supercomputer cycles are dedicated to Monte Carlo calculations. For that reason it's important to understand what makes a good parallel random number generator. We will start with a quick overview of sequential random number generators.

10.2 SEQUENTIAL RANDOM NUMBER GENERATORS

Technically, the random number generators you'll find on today's computers are **pseudo-random number generators,** because their operation is deterministic, and hence the sequences they produce are predictable. In the best case these sequences are a reasonable approximation of a truly random sequence. However, since "pseudo-random number generator" is a mouthful, we'll stick with the simpler phrase. In the remainder of this chapter, when you see the phrase "random number generator," understand we're talking about a pseudo-random number generator.

Coddington [17] has identified ten properties of the sequence of numbers produced by an ideal random number generator:

- It is **uniformly distributed,** meaning each possible number is equally probable.
- The numbers are uncorrelated.

- It never cycles; that is, the numbers do not repeat themselves.
- It satisfies any statistical test for randomness.
- It is reproducible.
- It is machine-independent; that is, the generator produces the same sequence of numbers on any computer.
- It can be changed by modifying an initial "seed" value.
- It is easily split into many independent subsequences.
- It can be generated rapidly.
- It requires limited computer memory for the generator.

There are no random number generators that meet all of these requirements. For example, computers rely upon finite precision arithmetic. Hence the random number generator may take on only a finite number of states. Eventually it must enter a state it has been in previously, at which point it has completed a cycle and the numbers it produces will begin to repeat themselves. The **period** of a random number generator is the length of this cycle.

Similarly, since we demand that the sequence of numbers be reproducible, the numbers cannot be completely uncorrelated. The best we can hope for is that the correlations be so small that they have no appreciable impact on the results of the computation.

There is often a trade-off between the speed of a random number generator and the quality of the numbers it produces. Since the time needed to generate a random number is typically a small part of the overall computation time of a program, speed is much less important than quality.

In the following sections we consider two important classes of random number generators: linear congruential and lagged Fibonacci.

10.2.1 Linear Congruential

The linear congruential method is more than 50 years old, and it is still the most popular. **Linear congruential generators** produce a sequence X_i of random integers using this formula:

$$X_i = (a \times X_{i-1} + c) \bmod M$$

where a is called the multiplier, c is called the additive constant, and M is called the modulus. In some implementations $c = 0$. When $c = 0$, it is called **a multiplicative congruential generator.** All three values must be carefully chosen in order to ensure that the sequence has a long period and good randomness properties. The maximum period is M. For 32-bit integers the maximum period is 2^{32}, or about 4 billion. This is too small a period for modern computers that execute billions of instructions per second. A quality generator has 48 bits of precision or more.

The particular sequence of integer values produced by the generator depends on the initial value X_0, which is called the **seed.** Typically the user provides the seed value.

Linear congruential methods may also be used to generate floating-point numbers. Since the generator produces integers between 0 and $M - 1$, dividing X_i by M produces a floating-point number x_i in the interval $[0, 1)$.

The defects of linear congruential generators are well documented. The least significant bits of the numbers produced are correlated. (This is particularly true when the modulus M is a power of 2.) If you produce a scatter plot of ordered tuples $(x_i, x_{i+1}, \ldots, x_{i+k-1})$ in a k-dimensional unit hypercube, you'll see a lattice structure [79]. Since this problem becomes more pronounced as the number of dimensions increases, it can affect the quality of high-dimensional simulations relying on a linear congruential random number generator.

Despite these flaws, linear congruential generators with 48 or more bits of precision and carefully chosen parameters "work very well for all known applications, at least on sequential computers" [17].

10.2.2 Lagged Fibonacci

The popularity of lagged Fibonacci generators is rising, because they are capable of producing random number sequences with astonishingly long periods, while also being fast. The method produces a sequence of X_is. Each element is defined as follows:

$$X_i = X_{i-p} \star X_{i-q}$$

where p and q are the lags, $p > q$, and $\star$ is any binary arithmetic operation. Examples of suitable $\star$ operations are addition modulo M, subtraction modulo M, multiplication modulo M, and bitwise exclusive or. In the case of addition and subtraction, the X_is may be either integers or floating-point numbers. If the sequence contains floating-point numbers, $M = 1$. If $\star$ is multiplication, the sequence must consist solely of odd integers.

Note that unlike linear congruential generators, which require only a single seed value, lagged Fibonacci generators require p seed values $X_0, X_1, \ldots, X_{p-1}$. Careful selection of p, q, and M, as well as $X_0, \ldots, X_{p-1}$ results in sequences with very long periods and good randomness. If the X_is have b bits, the maximum periods attainable are $2^p - 1$ for exclusive or, $(2^p - 1)2^{b-1}$ for addition and subtraction, and $(2^p - 1)2^{b-3}$ for multiplication. Notice that increasing the maximum lag p increases the storage requirements but also increases the maximum period.

Function `random`, callable from C, is an additive lagged Fibonacci generator with a default lag of 31. Coddington reports this lag is much too small. He recommends setting (p, q) to at least (1279, 1063).

10.3 PARALLEL RANDOM NUMBER GENERATORS

Parallel Monte Carlo methods depend upon our ability to generate a large number of high-quality random number sequences. In addition to the properties mentioned in the previous section for sequential random number generators, an ideal parallel

random number generator would have these properties:

- No correlations among the numbers in different sequences.
- Scalability; that is, it should be possible to accommodate a large number of processes, each with its own stream(s).
- Locality; that is, a process should be able to spawn a new sequence of random numbers without interprocess communication.

In this section we discuss four different techniques for transforming a sequential random number generator into a parallel random number generator.

10.3.1 Manager-Worker Method

Gropp et al. [45] have described a manager-worker approach to parallel random number generation. A "manager" process has the task of generating random numbers and distributing them to "worker" processes that consume them. Here are two disadvantages of the manager-worker approach.

Some random number generators produce sequences with long-range correlations. Because each process is sampling from the same sequence, there is a possibility that long-range correlations in the original sequence may become short-range correlations in the parallel sequences.

The manager-worker method is not scalable to an arbitrary number of processes. It may be difficult to balance the speed of the random number producer with the speed of the consumers of these numbers. It clearly does not exhibit locality. On the contrary, it is communication-intensive.

These disadvantages are significant, and this method is no longer popular. Let's consider methods in which each process generates its own random number sequence.

10.3.2 Leapfrog Method

The leapfrog method is analogous to a cyclic allocation of data to tasks. Suppose our parallel Monte Carlo method is executing on p processes. All processes use the same sequential random number generator. The process with rank r takes every pth element of the sequence, beginning with X_r:

$$X_r, X_{r+p}, X_{r+2p}, \ldots$$

Figure 10.5 illustrates the elements used by the process with rank 2 in a seven-process parallel execution in which each process generates its own random number sequence.

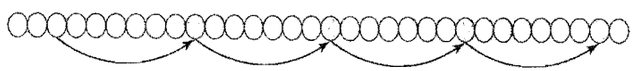

Figure 10.5 Process 2 (of 7) generates random numbers using the leapfrog technique.

It is easy to modify a linear congruential generator to incorporate leapfrogging. A jump of p elements is accomplished by replacing a with $a^p \bmod M$ and c with $c(a^p - 1)/(a - 1) \bmod M$. Makino has demonstrated how to modify lagged-Fibonacci generators to use leapfrogging [77].

Monte Carlo algorithms often require the generation of multidimensional random values. For instance, in the π estimation example we gave in Section 10.1, we generated coordinate pairs. If we want the parallel program to generate the same pairs as the sequential algorithm, the leapfrog method must be modified: we need to generate $(X_{2r}, X_{2r+1}, X_{2r+2p}, X_{2r+2p+1}, \ldots)$, not $(X_r, X_{r+p}, X_{r+2p}, X_{r+3p}, \ldots)$. This is a straightforward modification of the leapfrog method (Figure 10.6).

A disadvantage of the leapfrog method is that even if the elements of the original random number sequence have low correlation, the elements of the leapfrog subsequence may be correlated for certain values of p. This is especially likely to happen if p is a power of 2, a linear congruential generator is being used, and the modulus M is a power of 2. Even if this is not the case, leapfrogging can turn long-range correlations in the original sequence into short-range correlations in the parallel sequences.

Another disadvantage of the leapfrog method is that it does not support the dynamic creation of new random number streams.

10.3.3 Sequence Splitting

Sequence splitting is analogous to a block allocation of data to tasks. Suppose a random number generator has period P. The first P numbers emitted by the generator is divided into equal-sized pieces, one per process (Figure 10.7).

This method has the disadvantage of forcing each process to move ahead to its starting point in the sequence. This may take a long time. On the other hand, this only needs to be done at the initialization of the algorithm. After that, each process generates the elements in order.

Linear congruential generators with a power of 2 modulus have long-range correlations. Since the sequences produced by different processes represent elements far apart in the cycle, there may be correlations between the sequences produced by difference processes.

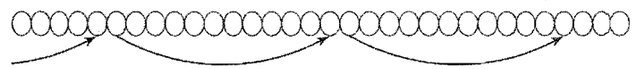

Figure 10.6 Process 2 (of 6) generates random number pairs in a modified leapfrog scheme.

Figure 10.7 In sequence splitting, each process is allocated a contiguous group of random numbers.

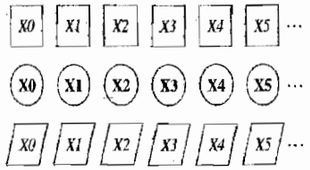

Figure 10.8 By using different parameters to initialize a sequential random number generator, it is often possible for each process to produce its own sequence.

Sequence splitting could be modified to support the dynamic creation of new sequences. For example, a process creating a new stream could give up half of its section to the new stream.

10.3.4 Parameterization

A fourth way to implement a parallel random number generator is to run a sequential random number generator on each process, but to ensure that each generator produces a different random number sequence by initializing it with different parameters (Figure 10.8).

Linear congruential generators with different additive constants produce different streams. Percus and Kalos have published a methodology for choosing the additive constant that works well for up to 100 streams [91].

Lagged Fibonacci generators are especially well suited for this approach. Providing each process with a different initial table of lag values allows each process to generate a different random number sequence. Obviously, correlations within lag tables or between lag tables would be fatal. One way to initialize the tables is to use a *different* lagged Fibonacci generator to generate the needed seed values. The processes could use the leapfrog technique or sequence splitting to ensure that they filled their tables with different values.

The number of distinct streams a lagged Fibonacci generator can produce is truly awesome [82]. For example, the default multiplicative lagged Fibonacci generator provided by the SPRNG library has around 2^{1008} distinct streams, allowing plenty of opportunities for creating new streams during the execution of the parallel program [83].

10.4 OTHER RANDOM NUMBER DISTRIBUTIONS

Our discussion to this point has focused on the problem of generating random numbers from a uniform probability density function. Sometimes we need to generate random numbers from other distributions.

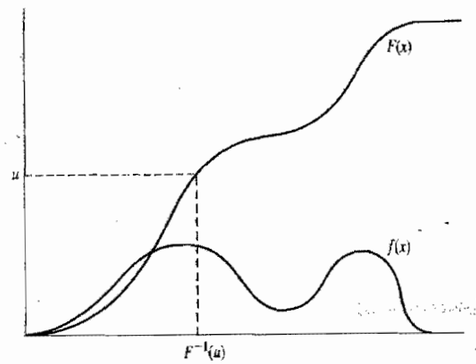

Figure 10.9 Given a probability density function $f(x)$, its cumulative distribution function $F(x)$, and u, a sample from a uniform distribution, then $F^{-1}(u)$ is a sample from $f(x)$.

10.4.1 Inverse Cumulative Distribution Function Transformation

Let u represent a sample from the uniform distribution [0, 1).

Suppose we want to produce random variables from a probability density function $f(x)$. If we can determine the cumulative distribution function $F(x)$ and invert it, then $F^{-1}(u)$ is a random variable from the probability density function $f(x)$ (see Figure 10.9).

As an example of this transformation, we will derive a formula that yields a sample from the exponential distribution.

Exponential Distribution The decay of radioactive atoms, the distance a neutron travels in a solid before interacting with an atom, and the time before the next customer arrives at a service center are examples of random variables that are often modeled by an exponential probability density function.

The exponential probability density function with expected value m is the function $f(x) = (1/m)e^{-x/m}$. We can integrate $f(x)$ to find the cumulative distribution function $F(x) = 1 - e^{-mx}$. Inverting $F(x)$, we find the inverse function to be $F^{-1}(u) = -m\ln(1-u)$. Since u is uniformly distributed between 0 and 1, there is no difference between u and $1-u$. Hence the function $F^{-1}(u) = -m\ln u$ is exponentially distributed with mean m.

EXAMPLE 1

Produce four samples from an exponential distribution with mean 3.

■ **Solution**

We start with four samples from a uniform distribution:

0.540 0.619 0.462 0.095

Taking the natural logarithm of each value and multiplying by −3:

$$-3\ln(0.540) \quad -3\ln(0.619) \quad -3\ln(0.462) \quad -3\ln(0.095)$$

yields four samples from an exponential distribution with mean 3:

$$1.850 \quad 1.440 \quad 2.317 \quad 7.072$$

EXAMPLE 2

A simulation advances in time steps of 1 second. The probability of a particular event occurring is from an exponential distribution with mean 5 seconds. What is the probability of the event occurring in the next time step? How do we determine if the event happens in the next time step?

■ Solution

The probability of an event occurring in the next time step is 1/5. To determine if the event happens in the next time step, we generate a random number from the uniform distribution between 0 and 1. If the random number is less than or equal to 1/5, the event has occurred.

10.4.2 Box-Muller Transformation

We cannot invert the cumulative distribution function to come up with a formula yielding random numbers from the normal (gaussian) distribution

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$$

Fortunately, the Box-Muller transformation allows us to produce a pair of standard deviates g_1 and g_2 from a pair of uniform deviates u_1 and u_2 [65]:

repeat
  $v_1 \leftarrow 2u_1 - 1$
  $v_2 \leftarrow 2u_2 - 1$
  $r \leftarrow v_1^2 + v_2^2$
until $r > 0$ and $r < 1$
$f \leftarrow \sqrt{-2\ln r / r}$
$g_1 \leftarrow f v_1$
$g_2 \leftarrow f v_2$

EXAMPLE 1

Produce four samples from a normal distribution with mean 0 and standard deviation 1.

u_1	u_2	v_1	v_2	r	f	g_1	g_2
0.234	0.784	−0.532	0.568	0.605	1.290	−0.686	0.732
0.824	0.039	0.648	−0.921	1.269			
0.430	0.176	−0.140	−0.648	0.439	1.935	−0.271	−1.254

■ Solution

From the uniform random samples 0.234 and 0.784 we derive the two normal samples −0.686 and 0.732. The next pair of uniform random samples 0.824 and 0.039 results in a value of $r > 1$, so we must discard these samples and generate another pair. The uniform samples 0.430 and 0.176 result in the normal samples −0.271 and −1.254.

EXAMPLE 2

Produce four samples from a normal distribution with mean 8 and standard deviation 2.

■ Solution

We modify the Box-Muller transformation by replacing the assignment

$$g_1 \leftarrow f v_1$$

with

$$g_1 \leftarrow 2 f v_1 + 8$$

We do a similar replacement for the assignment to g_2.

u_1	u_2	v_1	v_2	r	f	g_1	g_2
0.017	0.262	−0.965	−0.475	1.158			
0.832	0.743	0.663	0.486	0.676	1.075	9.426	9.045
0.670	0.439	0.339	−0.122	0.130	5.602	11.800	6.630

The value of r resulting from the uniform samples 0.017 and 0.262 is too large, and we must reject these samples. However, the uniform samples 0.832 and 0.743 produce the normal samples 9.426 and 9.045. The uniform samples 0.670 and 0.439 produce the normal samples 11.800 and 6.630.

You can use the Box-Muller transformation to create a function that returns a single standard deviate. On the first, third, fifth, etc. invocations of this function, it performs the Box-Muller transformation, stores g_2, and returns g_1. On the second, fourth, sixth, etc. invocations of this function, the function returns the value of g_2 produced in the previous invocation.

10.4.3 The Rejection Method

The rejection method, first proposed by John von Neumann, allows us to produce samples from a probability density function $f(x)$ that we cannot integrate and/or invert analytically. Suppose we can generate samples for another probability density function $h(x)$, and we can find a constant δ such that $f(x) \le \delta h(x)$ for all x (Figure 10.10). We produce samples from f in the following way: We generate a sample x_i from h and another sample u_i from the uniform distribution. If $u_i \delta h(x_i) \le f(x_i)$ we accept x_i as a sample from $f(x)$ and return it. Otherwise, we repeat the test with another x_i and another u.

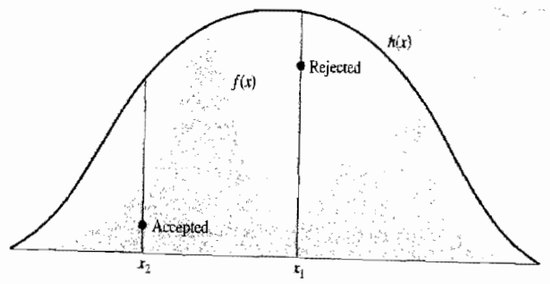

Figure 10.10 The rejection method allows us to generate samples from a probability density function $f(x)$. We produce x_i from probability density function $h(x)$ and u_i from the uniform probability density function over (0, 1). In this figure $u_1 = 0.8$, $u_1\delta h(x_1) > f(x_1)$, and we reject sample x_1. On the other hand, $u_2 = 0.15$, $u_2\delta h(x_2) < f(x_2)$, and we accept sample x_2.

The points $(x_i, u_i\delta h(x_i))$ uniformly sample the area under the curve $\delta h(x)$. Since we only accept those points under the curve $f(x)$, the resulting sequence of x_is reflects the probability density function $f(x)$.

The rejection method works best when there is a relatively small amount of error between $f(x)$ and $\delta h(x)$. The larger this area, the greater the frequency at which candidate random numbers will be rejected, slowing the process. The efficiency of the rejection method can decrease sharply as the number of dimensions increases. For example, suppose that 75 percent of the random numbers are accepted for a one-dimensional integral. If the same efficiency holds true as the number of dimensions increases, the efficiency for a six-dimensional integral would be $(0.75)^6$, or about 18 percent.

EXAMPLE

A random variable has the probability density function

$$f(x) = \begin{cases} \sin x, & \text{if } 0 \le x \le \pi/4; \\ (-4x + \pi + 8)/(8\sqrt{2}), & \text{if } \pi/4 < x \le 2 + \pi/4; \\ 0, & \text{otherwise} \end{cases}$$

This probability density function is illustrated in Figure 10.11.

■ Solution

We can use the rejection method to generate random variables from this distribution. We need to find δ and $h(x)$ such that $f(x) \le \delta h(x)$ for all x. We note that the probability density function is greater than 0 for the values of x between 0 and $2 + \pi/4$, and it has a maximum value of $\sqrt{2}/2$. We choose to use a uniform probability density function as our $h(x)$:

$$h(x) = \begin{cases} 1/(2 + \pi/4), & \text{if } 0 \le x \le 2 + \pi/4; \\ 0, & \text{otherwise} \end{cases}$$

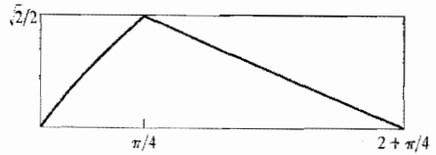

Figure 10.11 Using a uniform random variable and the rejection method to produce variables from a two-part probability density function.

If we multiply $h(x)$ by $\delta = (2 + \pi/4)(\sqrt{2}/2)$, then $\delta h(x) \ge f(x)$ for all x. Simplifying the terms, we see that

$$\delta h(x) = \begin{cases} \sqrt{2}/2, & \text{if } 0 \le x \le 2 + \pi/4; \\ 0, & \text{otherwise} \end{cases}$$

We generate a random number from the uniform distribution between 0 and 1 and multiply it by $2 + \pi/4$, giving us a random variable x_i from the uniform distribution between 0 and $2 + \pi/4$. Next we generate a random number u_i from the uniform distribution between 0 and 1. If $u_i\delta h(x_i) \le f(x_i)$, then we accept x_i as a sample from $f(x)$ and return it. Otherwise we generate another pair (x_i, u_i) and repeat the test.

x_i	u_i	$u_i\delta h(x_i)$	$f(x_i)$	*Outcome*
0.860	0.975	0.689	0.681	Reject
1.518	0.357	0.252	0.448	Accept
0.357	0.920	0.650	0.349	Reject
1.306	0.272	0.192	0.523	Accept

10.5 CASE STUDIES

The five case studies in this section provide a glimpse into a few of the many domains in which Monte Carlo methods are useful.

10.5.1 Neutron Transport

We consider a simplified model of neutron transport in two dimensions (see Figure 10.12). A source emits neutrons against a homogeneous plate having thickness H and infinite height. A neutron may be reflected by the plate, absorbed by the plate, or it may pass through the plate. We wish to compute the frequency at which each of these events occurs as a function of plate thickness H.

Two constants that describe the interaction of the neutrons in the plate are the cross section of the capture C_c and the cross section of the scattering C_s. The total cross section $C = C_c + C_s$.

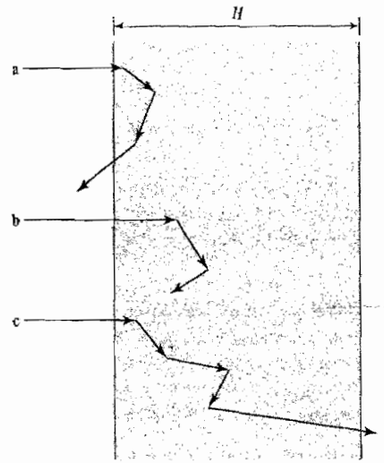

Figure 10.12 A neutron encountering a homogeneous medium may be (a) reflected, (b) absorbed, or (c) transmitted.

The distance L a neutron travels in the plate before interacting with an atom is modeled by an exponential distribution with mean $1/C$. As we saw in the previous section, if u is a random number from the uniform distribution $[0, 1)$, the formula

$$L = -\frac{1}{C} \ln u$$

is a random number from the appropriate exponential probability density function.

When a neutron interacts with an atom in the plate, the probability of bouncing off the atom is C_s/C, while the probability of being absorbed by the atom is C_c/C. We may use a random number from the uniform distribution $[0, 1)$ to determine the outcome of a neutron-atom interaction.

If a neutron scatters, it has an equal probability of moving in any direction. Hence its new direction D (measured in radians) can be modeled by a random variable uniformly distributed between 0 and π. (Since the plate has infinite height, we do not need to distinguish between bouncing upward and bouncing downward.) Given direction D, the actual distance in the x direction the neutron travels in the plate between collisions is $L \cos D$.

The simulation of a neutron continues until one of the following events occurs:

1. The neutron is absorbed by an atom.
2. The x position of the neutron is less than 0, meaning the neutron has been reflected by the plate.
3. The x position of the neutron is greater than H, meaning the neutron has been transmitted through the plate.

Neutron Transport Simulation:
C — Mean distance between neutron/atom interactions is $1/C$
C_s — Scattering component of C
C_c — Absorbing component of C
H — Thickness of plate
L — Distance neutron travels before collision
d — Direction of neutron (measured in radians between 0 and π)
u — Uniform random number
x — Position of particle in plate ($0 \le x < H$)
n — Number of samples
a — True while particle still bouncing
r, b, t — Counts of reflected, absorbed, transmitted neutrons

```
begin
  r, b, t ← 0
  for i ← 1 to n do
    d ← 0
    x ← 0
    a ← true
    while a do
      L ← −(1/C) × ln u
      x ← x + L × cos(d)
      if x < 0 then { Reflected }
        r ← r + 1
        a ← false
      else if x ≥ H then { Transmitted }
        t ← t + 1
        a ← false
      else if u < C_c/C then { Absorbed }
        b ← b + 1
        a ← false
      else
        d ← u × π
      endif
    endwhile
  endfor
  print r/n, a/n, t/n
end
```

Figure 10.13 Pseudocode for a neutron transport simulation using the Monte Carlo method.

Pseudocode for the neutron transport simulation appears in Figure 10.13. Note that time does not advance by the same amount in each iteration of the while loop. Instead, the simulation advances from one event (one interaction) to the next. This pseudo-time progression is called **Monte Carlo time.**

10.5.2 Temperature at a Point Inside a 2-D Plate

Imagine a very thin plate of homogeneous material. We wish to compute the steady-state temperature at a particular point in the plate. The top and the bottom of the plate are insulated, and the temperature at any point is solely determined by the temperatures surrounding it, except for the temperatures at the edges of the plate, which are fixed.

The interior temperature distribution is described by Laplace's equation, $\nabla^2 T = 0$, which means the temperature at a point is the average of the temperatures around it.

One approach to solving Laplace's equation numerically is to make the problem discrete by overlaying the plate with a two-dimensional mesh of points. In this case the temperature at a point is the average of the temperatures of the points above it, below it, to its right, and to its left (which we can think of as "north," "south," "east," and "west").

We can use a Monte Carlo technique to find the temperature at a particular point S. We compute the temperature of S by randomly choosing one of the four neighbors and adding its temperature to an accumulator. After we have sampled a random neighbor's temperature n times, we divide the sum by n to yield the temperature of S. This average has an expected value of $(T_n + T_s + T_e + T_w)/4$.

Of course, we do not know the temperatures of the neighboring points, but we could use the same technique to find their temperatures, too. Applying this idea recursively, we end up doing a **random walk** on the plate. The recursion and the random walk do terminate, because the temperatures on the edges of the plate are known.

Following are the Monte Carlo algorithm results (see Figure 10.14). We start at intersection S and randomly choose which direction to move (north, south, east, or west). We continue to move in a random fashion until we hit one of the edges of the plate. At this point we add the temperature at the edge to our accumulator and repeat. At each iteration we can also determine the average edge temperature encountered over all random walks we have taken so far. We terminate the algorithm when the average temperature value converges.

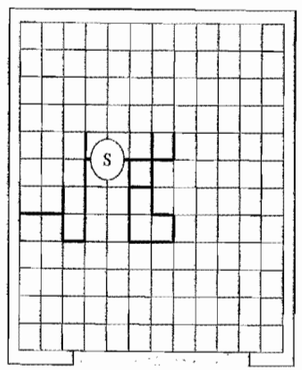

Figure 10.14 Use of a random walk to estimate the temperature of point S on a thin plate. The boundary temperatures are fixed. Edge points contacting the U-shaped white bar have temperature 0. Edge points contacting the gray bar have temperature 100. This random walk from S, illustrated by heavy lines, results in the temperature 0 being added to the sample.

10.5.3 Two-Dimensional Ising Model

The two-dimensional Ising model may be used to simulate the behavior of simple magnets as well as other phenomena (see Figure 10.15). The problem domain is a square lattice. Each intersection is called a **site.** Every site σ_k has an associated spin. Each spin can be in one of two states: up or down. We associate the value $1/2$ with up and the value $-1/2$ with down. The energy of the system is determined

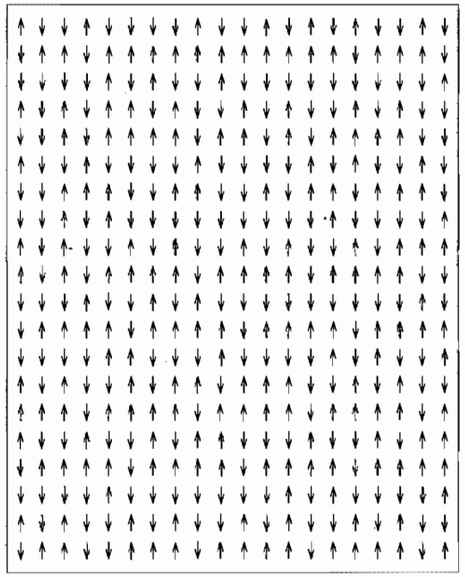

Figure 10.15 A 20 x 20 Ising model. Each of the 400 sites has an associated spin, either up or down. The energy of the system is a function of the spins. The model may take on any of 2^{400} different states. The probability of entering each of these states is influenced by both the current state of the system and its temperature.

by the function

$$E(\sigma) = -\sum_{i,j} J\sigma_i\sigma_j - B\sum_i \sigma_i$$

where the first sum is over nearest neighbors, J is a constant indicating the strength of the spin-spin interaction, and B is another constant having to do with the external magnetic field.

Our goal is to estimate the specific heat per particle, a problem analogous to performing integration over the possible configurations of the system. Given the temperature T and Boltzmann's constant k, the probability density function over all possible configurations is

$$\mu(\sigma) = \frac{e^{-E(\sigma)/kT}}{Z(T)}$$

where $Z(T)$ is a weighted sum over all states.

Unfortunately, it is difficult to sample from distribution μ. Here, a random sample x_i represents a configuration of spins. Note that the number of configurations is exceedingly large, even for small lattices. For example, our example 20×20 lattice has 400 sites. Each site has two possible values, meaning the number of configurations is 2^{400}. Because the probability density function is an inverse exponential function, the probabilities associated with most states are extremely small. If we try to take a uniform sample of the configurations, it is unlikely we will "hit" on enough of the higher-probability configurations to yield a good estimate of the integral. Instead, we need to find a sampling of the configurations that is biased toward those that have higher probability. The Metropolis algorithm generates such a sampling.

The **Metropolis algorithm** uses the current configuration x_i (current random sample) to generate the next configuration x_{i+1} (next random sample). Given x_i, the algorithm generates a neighboring configuration x'. If $E(x') < E(x_i)$, then $x_{i+1} = x'$. If $E(x') > E(x_i)$, then $x_{i+1} = x'$ with probability $e^{-[E(x')-E(x_i)]/kT}$; otherwise $x_{i+1} = x_i$. The series of random samples, called a **Markov chain**, represents a random walk through the possible configurations. When applied to the Ising model, the Metropolis algorithm takes the form shown in Figure 10.16.

While short series of random samples produced by the Metropolis algorithm are highly correlated, if the algorithm is allowed to produce enough samples, it can provide good coverage of an entire probability density function.

How can we be sure that the Markov chain of configurations visited by the Metropolis algorithm corresponds to the underlying probability density function? One way to be sure is to satisfy the detailed balance condition. Let $P(x_i)$ represent the probability of being in configurations x_i, and let $P(x_j \mid x_i)$ represent the probability of the random walk moving to configuration x_j from configuration x_i. The **detailed balance condition** holds if

$$P(x_1)P(x_2 \mid x_1) = P(x_2)P(x_1 \mid x_2)$$

Metropolis Algorithm:
k — Boltzmann's constant
T — Temperature
E — Energy function
Δ — Change in energy
ρ — Probability of changing to state x'
u — Uniform random variable

```
begin
  x_0 ← Initial state of model
  i ← 0
  repeat
    σ ← randomly selected site (from uniform distribution)
    x' ← Identical to x_i except spin at σ is reversed
    Δ ← E(x') − E(x_i)
    if Δ < 0 then
      ρ ← 1
    else
      ρ ← e^{−Δ/kT}
    endif
    if u < ρ then
      x_{i+1} ← x'
    else
      x_{i+1} ← x_i
    endif
    i ← i + 1
  forever
end
```

Figure 10.16 The Metropolis algorithm applied to the Ising model.

Suppose $E(x') > E(x_i)$. The Metropolis algorithm satisfies the detailed balance condition if

$$\begin{aligned} & P(x_i)P(x' \mid x_i) && = P(x')P(x_i \mid x') \\ \Rightarrow\ & \frac{e^{-E(x_i)/kT}}{Z(T)} \times e^{-[E(x')-E(x_i)]/kT} && = \frac{e^{-E(x')/kT}}{Z(T)} \times 1 \\ \Rightarrow\ & \frac{e^{-E(x')/kT}}{Z(T)} && = \frac{e^{-E(x')/kT}}{Z(T)} \end{aligned}$$

The equality also holds if $E(x') \leq E(x_i)$. Hence the Metropolis algorithm satisfies the detailed balance condition.

10.5.4 Room Assignment Problem

Given n, an even number of college freshmen, our goal is to assign them to $n/2$ rooms in a residence hall so that interpersonal conflicts are minimized. Every student has completed a survey, and a computer program has produced a table of "dislikes"—in other words, the value of entry (i, j) of the table indicates the extent to which students i and j are likely to get on each other's nerves. (The value

of entry $[i, j]$ equals the value of entry $[j, i]$.) We will solve this problem using a technique called simulated annealing.

Physical annealing is the process of heating a solid until it melts, then cooling it slowly. The purpose of physical annealing is to produce a strong, defect-free crystal with a regular structure. When the material is hot, the atoms are in a higher-energy state and more easily rearrange themselves. As the temperature drops, the atomic energies decrease, and the atoms do not rearrange themselves as easily. Slow cooling allows the material to reach a state of minimum energy, which is its crystalline form.

Simulated annealing makes an analogy between physical annealing and solving a combinatorial optimization problem. A solution to the optimization problem corresponds to a state of the material, the value of the objective function for a particular solution corresponds to the energy associated with a particular state, and the optimal solution to the problem corresponds to the minimum energy state.

Simulated annealing is an iterative algorithm. During each iteration the current solution is randomly changed to create an alternate solution in the neighborhood of the current solution. If the value of the objective function for the new solution is less than the value of the objective function for the current solution, then the new solution becomes the current solution. If the value of the objective function for the new solution is greater than the value of the objective function for the current solution, then the new solution becomes the current solution with probability $e^{-\Delta/T}$, where Δ is the difference between the values of the objective function and T is the current "temperature."

Why would we want to move to a solution that is inferior to one we have already found? The reason is that solution spaces usually have local minima. We do not want the algorithm to settle too quickly into a local minimum. When the temperature is higher, the algorithm can easily "climb out of" local minima (Figure 10.17a). When the temperature decreases, the probability of doing so is reduced (Figure 10.17b).

Note that simulated annealing and the Metropolis algorithm are closely related. Both use the same probability function to determine if a jump should be made to a higher-energy state. The difference is that in simulated annealing we are searching for the minimum value of the function, rather than computing an integral.

In order to solve a problem using simulated annealing, we must:

- decide how to represent solutions
- define the cost function
- define how to generate a new, neighboring solution from an existing solution
- design a cooling function

Let's go through each of these steps for the room assignment problem. We start with an incompatibility matrix D; entry $d_{i,j}$ is a floating-point value between 0 and 10 that indicates how much students i and j are going to dislike each other. Note that $d_{i,j} = d_{j,i}$.

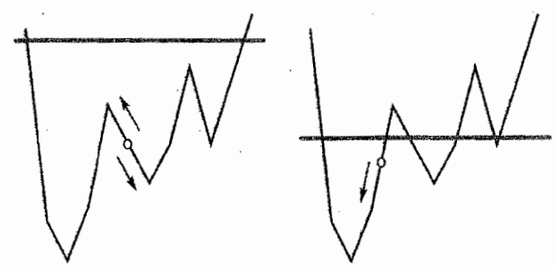

Figure 10.17 Simulated annealing always allows the search to move to a newly generated solution of lower cost. The probability of moving to a newly generated solution of higher cost shrinks as the temperature drops. (a) When the temperature is high, moving to a solution of higher cost is more probable. (b) When the temperature is low, moving to a solution of higher cost is less probable.

A solution is an assignment of the n students to $n/2$ rooms. We create array a to keep track of these assignments. Each entry a_i is an integer between 0 and $n/2 - 1$, representing the roomt person i is assigned to. Each value j in the range 0 through $n/2 - 1$ appears exactly twice in array a.

The cost function is simply the sums of the incompatibilities of the students in the rooms. Let r_i represent the roommate of student i. Then the cost function is defined to be

$$\sum_{i=0}^{n-1} d_{i,r_i}$$

We can generate a new solution near the current solution by choosing two students at random and switching their room assignments.

Finally, we need to choose the temperature function. The choice of temperature function can have a great effect on the performance of the algorithm. A poor function may cause a simulated annealing algorithm to find a poor solution, take too long to execute, or both.

For this problem we choose a simple geometric temperature function:

$$T_0 = 1$$

$$T_{i+1} = 0.999T_i$$

Figure 10.18 illustrates the convergence of the simulated annealing algorithm solving the room assignment problem using a geometric temperature function. Both algorithms find the same solution, but the algorithm starting with $T_0 = 10$ iterates twice as long as the algorithm starting with $T_0 = 1$.

Pseudocode for a simulated annealing solution to the room assignment problem appears in Figure 10.19.

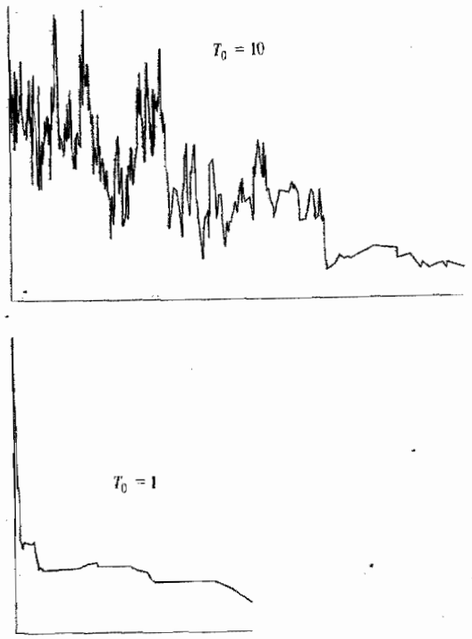

Figure 10.18 Convergence of simulated annealing algorithms solving the room assignment problem. In both cases the geometric temperature function $T_{i+1} = 0.999T_i$ is used. Both algorithms converge on the optimal solution. However, when the initial temperature is higher, the convergence is slower.

Simulated annealing is not guaranteed to find an optimal solution. In fact, the same algorithm using different streams of random numbers may converge on different solutions. Hence it makes sense to execute the same algorithm multiple times with different random number seeds. This is an obvious opportunity to use a parallel computer to speed overall execution time.

10.5.5 Parking Garage

A parking garage has S stalls. The length of time between successive arrivals of cars at the entrance to the garage is a random variable from a Poisson distribution with mean A minutes. If a car arrives at the garage and a stall is available, it occupies one of the stalls. The length of time a car stays in the garage is a random variable from a normal distribution with mean M minutes and standard deviation $M/4$ minutes. If a car arrives at the entrance and no stalls are available, the car is turned away. We wish to determine the steady-state characteristics of the parking

Room Assignment Problem:

```
a[0..n − 1] — n-element array containing room assignments
c1, c2 — two persons involved in possible room swap
d[0..n − 1, 0..n − 1] — n × n matrix containing roommate incompatibilities
sum — sum of dislikes of best solution found so far
new_sum — sum of dislikes of newly generated solution
t — temperature

begin
  Randomly assign students to rooms
  sum ← 0
  for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
      if a[i] = a[j] then
        sum ← sum + d[i][j]
    endfor
  endfor
  t ← 1
  i ← 0
  while i < 1000 do {Stop if no changes for 1000 iterations}
    repeat
      c1 ← {u × n}
      c2 ← {u × n}
    until a[c1] ≠ a[c2]
    Compute new_sum assuming c1 and c2 swap rooms
    if new_sum < sum or u ≤ e^((sum−new_sum)/t) then
      Swap room assignments for c1 and c2
      sum ← new_sum
      i ← 0
    else i ← i + 1
    endif
    t ← 0.999 × t
  endwhile
  print a and sum
end
```

Figure 10.19 Solving the room assignment problem using simulated annealing.

garage: the average number of stalls occupied by cars and the probability of a car being turned away because the garage is full.

We model time in minutes as a real variable t. When the simulation begins, $t = 0$.

We model the parking garage stalls as an array G with S elements. Element G_i contains the time that stall i is available. At the beginning of the simulation $G_i = 0$, for all i, $0 \leq i < S$.

We begin the simulation with the arrival of the first car; that is, it arrives at time 0.

Since car arrivals are characterized by a Poisson distribution, the time between car arrivals is an exponential distribution with mean A. As we saw in the previous section, we can use the expression $-A \ln u$ to determine the next car arrival time, where u is a random number uniformly distributed in [0, 1).

We increment t by this amount and look for an available stall; that is, a stall i such that $G_i \leq t$.

When we assign a car to a stall i, we must reset G_i to reflect the time the car leaves the parking garage. Since this is a normal distribution, we can use the Box-Muller transformation described in Section 10.4.

10.5.6 Traffic Circle

A traffic circle (also called a rotary or a roundabout) is a way of handling traffic at an intersection without using signal lights. Often seen in Europe and the northeastern United States, traffic circles support the concurrent movement of multiple cars in the same direction.

Figure 10.20 illustrates a simple traffic circle. Traffic feeds into the circle from four roads, labeled N, W, S, and E. Every vehicle moves around the circle in a counterclockwise direction.

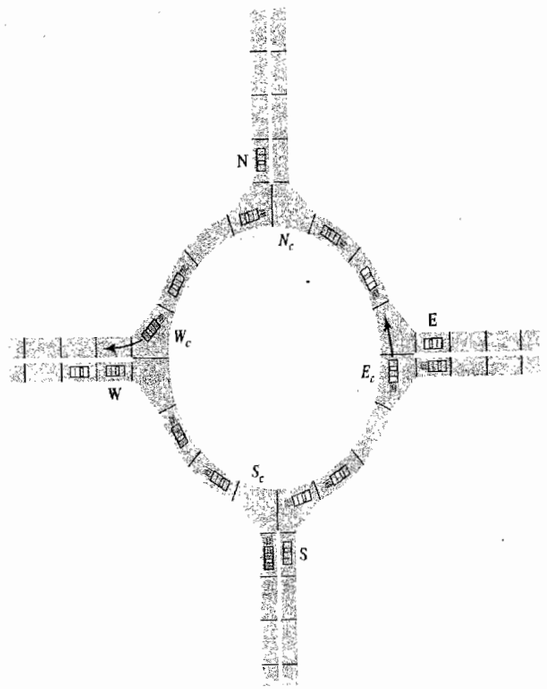

Figure 10.20 A model of a traffic circle. Cars within the traffic circle travel in a counterclockwise direction and take priority over cars trying to enter the circle.

	f
N	3
W	3
S	4
E	2

D	N	W	S	E
N	0.1	0.2	0.5	0.2
W	0.2	0.1	0.3	0.4
S	0.5	0.1	0.1	0.3
E	0.3	0.4	0.2	0.1

Figure 10.21 Probabilities associated with the traffic circle problem. Array element f_i is the mean time between vehicle arrivals at entrance i. Matrix element $d_{i,j}$ is the probability that a car entering the circle at i will exit at j.

In our simulation of the circle we divide the circle into 16 sections. During a single time step all vehicles inside the circle move to the next section in the counterclockwise direction (or leave the circle at one of the four exits). Vehicles within the traffic circle take priority over vehicles trying to enter the circle. Hence vehicles inside the circle are never prevented from moving forward during a time step.

A vehicle wishing to enter the circle may do so if there is no vehicle already in the circle attempting to enter the same zone. In Figure 10.20, for example, the vehicles waiting at N and S may enter the circle at the next time step, since there are no vehicles in the potential conflict zones S_c and N_c. The vehicle waiting at W may also enter the circle, because the car at W_c is leaving the circle. However, the vehicle waiting at E may not enter the circle, because the car at E_c is staying in the circle and has precedence.

To complete our model of the traffic circle, we must know the frequency at which cars arrive at the four access points. We must also know the frequency at which cars entering at a certain point exit at each of the four points. See Figure 10.21. The probability of a car arriving at an entrance during a particular time step is a random variable from an exponential distribution with mean m. Array f provides the mean time between arrivals at each of the four entrances. Element $d_{i,j}$ of matrix D is the probability that a car entering at i will exit at j. For example, the probability that a car entering at E will exit at S is 0.20.

Our goal is to construct a simulation of the traffic circle in order to answer two questions:

1. For each of the four traffic circle entrances, what is the probability that a car will have to wait before entering the circle?
2. For each of the four traffic circle entrances, what is the average length of the queue of vehicles waiting to enter the traffic circle?

Eight principal arrays are sufficient to perform the simulation and store the information needed to answer these two questions. See Figure 10.22. The traffic

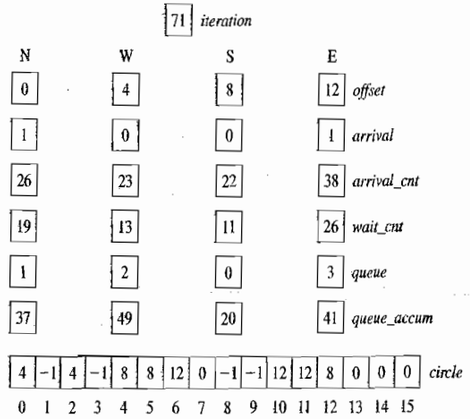

Figure 10.22 Data structures supporting the traffic circle simulation.

circle itself is represented by *circle,* a circular buffer implemented as an array of 16 integers. Array *offset* indicates the index in *circle* associated with each of the four entrances and exits. Index 0 represents the northern entrance/exit, index 4 is the location of the western entrance/exit, and so on. Each element of array *circle* represents a circle segment that is either empty or holds one car. If $circle[i] = -1$, the segment is empty. Otherwise, $circle[i]$ contains an integer that represents the car's exit (0, 4, 8, or 12).

When cars arrive at one of the entrances to the traffic circle, the appropriate element of array *arrival* is set to 1 at that time step. Array *arrival_cnt* contains the total number of arrivals at each entrance, and array *wait_cnt* is a count of the number of cars that could not enter the traffic circle immediately. Array *queue* keeps track of how many cars are waiting to enter the traffic circle at each entrance, and array *queue_accum* is a total, over all time steps of the simulation, of the values in *queue*.

Pseudocode for the traffic circle simulation appears in Figure 10.23. Each time step of the simulation is divided into three phases. First, new cars arrive at the traffic circle. Second, cars already inside the traffic circle move forward. (Array *new_circle*, not shown in Figure 10.22, facilitates this phase.) Cars that reach their destination exit are removed from the circle. Note that there is no need to simulate the lanes leading away from the circle. Third, cars enter the circle if there is room.

When a car does enter the traffic circle, the simulation must determine the desired exit of that car by generating a uniform random variable and referring to matrix *D*. In the pseudocode this step is represented by a call to function *ChooseExit*. We illustrate this process with an example. Suppose a car is entering

Traffic Circle Simulation:

```
Data Structures Representing the Traffic Circle
  circle[0..15] — Current state of traffic circle
  new_circle[0..15] — Next state of traffic circle

Data Structures Representing the Four Entrances
  offset[0..3] — Each entrance's location (index) in traffic circle
  arrival[0..3] — 1 if a car arrived this time step
  wait_cnt[0..3] — Number of cars that have had to wait
  arrival_cnt[0..3] — Total number of cars that have arrived
  queue[0..3] — Number of cars waiting to enter circle
  queue_accum[0..3] — Accumulated queue size over all time steps

begin
 for i ← 0 to 15 do
  circle[i] ← −1
 endfor
 for i ← 0 to 3 do
  arrival_cnt[i], wait_cnt[i], queue[i], queue_accum[i] ← 0
 endfor
 for iteration ← 0 to requested_iterations
  { New cars arrive at entrances }
  for i ← 0 to 3 do
   if u ≤ 1/f[i] then { u is a uniform random number }
    arrival[i] ← 1
    arrival_cnt[i] ← arrival_cnt[i] +1
   else arrival[i] ← 0
   endif
  endfor
  { Cars inside circle advance simultaneously }
  for i ← 0 to 15 do
   j ← (i +1) mod16
   if circle[i] = −1 or circle[i] = j then new_circle[j] ← −1
   else new_circle[j] ← circle[i]
   endif
  endfor
  circle ← new_circle
  { Cars enter circle }
  for i ← 0 to 3 do
   if circle[offset[i]] = −1 then
    {There is space for car to enter}
    if queue[i] > 0 then
     {Car waiting in queue enters circle}
     queue[i] ← queue[i] −1
     circle[offset[i]] ← Choose_Exit(i)
    else if arrival[i] > 0
     {Newly arrived car enters circle}
     arrival[i] ← 0
     circle[offset[i]] ← Choose_Exit(i)
    endif
   endif
```

Figure 10.23 Pseudocode for traffic circle simulation.

```
    if arrival[i] > 0 then
      {Newly arrived car queues up}
      wait_cnt[i] ← wait_cnt[i] + 1
      queue[i] ← queue[i] + 1
    endif
  endfor
  for i ← 0 to 15 do
    queue_accum[i] ← queue_accum[i] + queue[i]
  endfor
 endfor {iteration}
end
```

Figure 10.23 (contd.) Pseudocode for traffic circle simulation.

from the west, and we generate the random variable 0.55. We work through row W of matrix D until the total of the probabilities exceeds 0.55. The first entry is 0.2, which is not greater than 0.55. That means the destination is not the north exit. The second entry is 0.1. Adding this value to the first gives us 0.3. Since 0.3 is not greater than 0.55, the destination is not the west exit. Adding the third entry, 0.3, to the total gives us 0.6. Since 0.6 is greater than 0.55, the south exit is the destination. The car is entering at the west entrance (offset 4) and leaving at the south exit (offset 8). Hence we perform the assignment $circle[4] \leftarrow 8$.

When the traffic circle simulation begins, there are no cars inside the traffic circle, and delays are at a minimum. As the simulation progresses, traffic jams develop and then dissipate. The simulation should continue until the answers to the two questions have converged.

10.6 SUMMARY

Monte Carlo methods use statistical sampling to find approximate solutions to a wide variety of problems. Two important applications of Monte Carlo methods are numerical integration and simulation. Monte Carlo methods are superior to deterministic numerical algorithms for finding integrals when the number of dimensions is larger than about six. It is difficult to derive analytical answers to many questions arising from systems with stochastic behavior. Monte Carlo simulations of these systems can be good tools for generating approximate answers to these questions.

In order to produce reliable results, a Monte Carlo method must have access to a good stream of random numbers. The maximum period of a random number generator returning 32-bit integers is 2^{32}, or about four billion. This is too small a period for modern computers. Make sure you use a generator that has at least 48 bits of precision.

Sometimes a random number generator that is good in general may not work well for a particular application. If you have a critical application, it is a good idea to run it twice using two different random number generators to see if both runs produce similar results.

A variety of methods have been proposed for generating random numbers on a parallel computer, including the leapfrog method, sequence splitting, and maintaining independent sequences.

The most popular random number generators produce a pseudo-random sequence of values from a uniform distribution. Often a Monte Carlo method requires a random number from another distribution. Straightforward algorithms exist to transform samples from a uniform distribution into samples from an exponential distribution or a normal (gaussian) distribution. The rejection method allows us to produce numbers from other distributions.

We have considered six applications of the Monte Carlo method that demonstrated a variety of solution techniques. In the process of solving these problems we introduced two important algorithms. The Metropolis algorithm is a particularly good way to produce a sample from a high-dimensional space. Simulated annealing is an algorithm for finding approximate solutions to combinatorial optimization problems.

10.7 KEY TERMS

detailed balance condition
linear congruential generator
Markov chain
Metropolis algorithm
Monte Carlo method
Monte Carlo time
multiplicative congruential generator
period
pseudo-random number generator
random walk
seed
simulated annealing
site
uniform distribution

10.8 BIBLIOGRAPHIC NOTES

Easy-to-understand introductions to the Monte Carlo method and the Metropolis algorithm appear in *Computational Physics: Problem Solving with Computers* by R. Landau and Paez [65]. In contrast, *A Guide to Monte Carlo Simulations in Statistical Physics* by D. Landau and Binder provides a more rigorous presentation of the design and implementation of Monte Carlo simulations and the analysis of their results [64]. I first saw the "raindrops in cake pans" analogy in another introductory book, *Monte Carlo Methods*, written by Kalos and Whitlock [58].

Lehmer published the linear congruential method in 1951 [69]. For a time it was called "Lehmer's algorithm." Work on linear congruential generators with much longer periods continues. Wu gives a multiplicative congruential generator with the large prime-modulus $2^{61} - 1$ and four forms of multipliers [118]. However, L'Ecuyer and Simard warn that this generator fails a test of independence between the number of 1s in the binary representations of consecutive random numbers [68].

A variety of algorithms have been proposed for generating random variables from important nonuniform distributions. Wallace describes a fast way to generate normal and exponential random variables without relying on a source of uniform random variables [110]. Leva presents a fast algorithm for generating normal random variables that requires on average only 0.012 logarithm evaluations per standard deviate [72]. Marsaglia and Tsang describe a fast method for generating normal, exponential, and other random variables [80].

Mascagni surveys methods for generating parallel streams of random numbers via parameterization rather than sequence splitting [84]. His article contains a useful bibliography of earlier work.

The Scalable Parallel Random Number Generators (SPRNG) library, briefly documented by Mascagni and Srinivasan in *ACM Transactions on Mathematical Software* [83], is freely available from Florida State University. The URL is `http://sprng.cs.fsu.edu`.

The traffic circle problem is based on an example from Manno's *Introduction to the Monte-Carlo Method* [78].

10.9 EXERCISES

10.1 Suppose you are using the Monte Carlo method to compute an integral. The methodology is similar to the π-finding example in Section 10.1, except that the function to be integrated has 20 dimensions rather than 2. What sort of problem should you look out for if you are using a linear congruential random number generator?

10.2 An approach to parallel random number generation not discussed in the book is to assign each process the same linear congruential generator (with identical values for the multiplier, additive constant, and modulus). However, each process starts with a different seed value X_0. What is the principal risk associated with this approach?

10.3 Write a C function that uses the Box-Muller transformation to return a double-precision floating-point number representing a random value from the normal distribution.

10.4 A cylindrical hole with diameter d is drilled completely through a cube with edge length s so that the center of the cylindrical hole intersects two opposite corners of the cube. (See Figure 10.24.) Write a program to determine, with five digits of precision, the volume of the portion of the cube that remains when $s = 2$ and $d = 0.3$. Hint: The distance between the point (x_1, y_1, z_1) and the line $x = y = z$ is

$$\sin\left[\cos^{-1}\left(\frac{x_1 + y_1 + z_1}{\sqrt{3}\sqrt{x_1^2 + y_1^2 + z_1^2}}\right)\right]\sqrt{x_1^2 + y_1^2 + z_1^2}$$

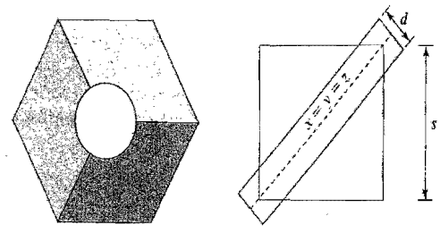

Figure 10.24 Two views of the object described in Exercise 10.4. (a) Looking down on one corner of the cube, we see that the hole goes completely through to the opposite corner. (b) Looking from the side of the cube, we see that all material within distance $d/2$ of the line $x = y = z$ is removed.

10.5 Write a program to evaluate the definite integral

$$\int_{x=0}^{4}\int_{y=0}^{3}\int_{z=0}^{2} 4x^3 + xy^2 + 5y + yz + 6z\,dz\,dy\,dx$$

to five digits of precision.

10.6 Write a program to evaluate the definite integral

$$\int_{x=0}^{4}\int_{y=0}^{3}\int_{z=0}^{x+y} 4x^3 + xy^2 + 5y + yz + 6z\,dz\,dy\,dx$$

to five digits of precision.

10.7 A radioactive atom has a mean lifetime of m time units. The probability that a radioactive atom will decay in any given time unit is $(1/m)e^{-1/m}$. Given an initial pool of 100,000 radioactive atoms, compute how many atoms decay at each time step in the first 1,000 time units, when $m = 250$.

10.8 Implement a parallel program solving the neutron transport problem described in Section 10.5.1. Let $C_c = 0.3$ and $C_s = 0.7$. Determine the probability of absorption, reflection, and transmission for $H = 1, 2, 3, \ldots, 10$. Base your results on 10 million tests (neutrons) for each value of H.

10.9 Implement a parallel program solving the steady-state temperature problem described in Section 10.5.2. Assume the square plate has been discretized into a 20×20 grid of smaller squares. Assume the temperature on three sides of the plate is 0° and the temperature on the fourth side is 100°. Compute the temperature at the middle of the plate to three digits of precision.

10.10 Write a parallel program implementing the Ising model described in Section 10.5.3. The objective is to find the energy level of a 100×100 system after 1,000,000 iterations. Let $J = 1$, $B = 0$, and $kT = 1$. Give the system a "cold start" by initializing every site σ_k to "up" in state x_0. Evaluate $\sum_{i,j} J\sigma_i\sigma_j$ for every pair of sites that are horizontally or vertically adjacent. Repeat the experiment 1,000 times.

10.11 Implement a parallel program solving the room assignment problem posed in Section 10.5.4. Assume $n = 20$ and $T = 1$. Use a random number generator to construct matrix D. Each entry should be a uniform random variable between 0 and 10. Each process should solve the problem for the same matrix D, but with different seeds for the random number generator.

10.12 Implement a parallel program solving the parking garage problem posed in Section 10.5.5. Assume $S = 80$, $A = 3$, and $M = 240$. Determine the average number of stalls occupied by cars, and the probability of a car being turned away because the garage is full, as $t \to \infty$, that is, in the steady state.

10.13 Implement a parallel program solving the traffic circle problem posed in Section 10.5.6. Use the program to answer these two questions:

a. For each of the four traffic circle entrances, what is the steady state probability that a car will have to wait before entering the circle?

b. For each of the four traffic circle entrances, what is the average length of the queue of vehicles waiting to enter the traffic circle, in the steady state?

CHAPTER

11

Matrix Multiplication

We go on multiplying our conveniences only to multiply our cares. We increase our possessions only to the enlargement of our anxieties.

Anna C. Brackett, *The Technique of Rest*

11.1 INTRODUCTION

Considering how often the matrix multiplication algorithm is presented in computer science classes, it's ironic that few scientific and engineering problems require the multiplication of large matrices. Here are two domains in which matrix multiplication is used. Computational chemists represent some problems in terms of states of a chemical system. Each index corresponds to a different basis state, and the matrix approximates the Hamiltonian of the system. A change of basis is accomplished through matrix multiplication. As another example, some transforms used in signal processing rely on the multiplication of large matrices.

This chapter presents two sequential matrix multiplication algorithms and then explores two different approaches to parallel matrix multiplication. In Section 11.2 we review the standard sequential matrix multiplication algorithm. Charting the algorithm's performance as matrix sizes increase, we see how performance drops dramatically once the second factor matrix no longer fits inside cache memory. We then show how a recursive implementation of matrix multiplication that multiplies blocks of the original matrices can maintain a high cache hit rate.

In Section 11.3 we design a parallel algorithm based upon a rowwise block-striped decomposition of the matrices. We derive an expression for the expected computation time of this algorithm, and we analyze its isoefficiency. In Section 11.4 we go through the same design and analysis methodology for a parallel algorithm based on a checkerboard block decomposition of the matrices.