Chapter 8 Monte Carlo Methods

"There is no such thing as a perfectly random number." - Harold Bailey, my 8th grade math teacher

Preface

When I was a youngster, I was the captain of my junior high school math team. One day, one of my teammates posed a problem that required Mr. Bailey, our coach, to pick a random number.

"Pick a perfectly random number, " he was challenged.

"What do you mean?" he replied.

"Pick a number that is as random as it can be," said my teammate, to which Mr. Bailey replied,

"There is no such thing as a perfectly random number."

And thus came to an end an idea that my friend once thought was pretty obvious, and one about which I had never given much thought. It is several decades later, and I will spend a few pages here on the subject of random numbers, their creation, and use.

8.1 Introduction

This chapter introduces a very important class of problem solving techniques known as Monte Carlo methods, which are based on the idea of randomization. A Monte Carlo method is an algorithm that uses streams of random numbers to solve a problem. Its name comes from the fact that the algorithm resembles gambling, because the solution is a guess based on probabilistic techniques, with a nonzero chance that it is incorrect. The Monte Carlo method arose out of discussions among John von Neumann, Robert Richtmyer, Stanislaw Ulam, and Nicholas Metropolis in the course of their research on nuclear radiation while working at Los Alamos Scientific Laboratory in the period from 1945 through 1947[9, 3], and named by John von Neumann, after the famous casino in Monaco.

8.2 Applications of Monte Carlo Methods

Many problems that arise in the physical and social sciences, pure mathematics, and many other disciplines, cannot be solved by ordinary deterministic methods. Many of these problems ultimately require the evaluation of multidimensional definite integrals. A multidimensional integral is of the form

$$\int\int\int \ldots \int f(x_1, x_2, x_3, \ldots, x_n) dx_1 dx_2 dx_3 \cdots dx_n$$

where the bounds of integration for each dimension are independent of each other. Evaluating an integral of this form using a deterministic method such as Simpson's Rule or the trapezoid or rectangular method does not work well because each dimension requires a large number of independent intermediate points. If, for example, we need 100 points in each of 30 dimensions, then we would need to evaluate the function 100^{30} times. (If this is not obvious, change the integrals to a discrete sum of the form $\sum_{x_2=1}^{100} \sum_{x_1=1}^{100} f(x_1)f(x_2)...$ to see the number of evaluations of the product.) This is often called the *curse of dimensionality.* Such high-dimensionality integrals arise in quantum physics and quantum mechanics with regularity. To compute the energy of a molecule that has n electrons, each of which has three spatial coordinates and one spin coordinate, requires solving a complex integral of dimension $4n$ many times over.

In economics, calculating risk in business and the future value of the Dow Jones Industrial average are problems that can only be solved using Monte Carlo methods. Monte Carlo methods are used in space exploration, image processing, artificial intelligence, and applied statistics. In artificial intelligence, Monte Carlo methods are used for searching large state spaces for optimal moves in games.

8.2.1 Example: Calculating PI (again)

If you are wondering why it is that calculating an estimate of π arises so often as an example of a method of computing, it is simply that everyone knows what π is, and that there are so many different and interesting ways to approximate its value. In this chapter estimating π provides a simple example of one use of the Monte Carlo method, albeit not the best example to illustrate its power.

A circle of unit radius has area $\pi r^2 = \pi$. Imagine a circle of radius 1.0 contained in a square whose sides have length equal to the circle's diameter, as shown in Figure 8.1. Since the radius is 1.0, the area of the circle is π and the area of the square is $(2r)^2 = 4.0$. Hence the ratio of the area of the circle to that of the square is $\pi/4$. Imagine now that the circle and the square are both shallow pans that can hold water, for instance, and that their sides are of infinitesimal thickness. We can conduct a thought experiment as follows. We place the two pans, one inside the other, on a flat surface outdoors during a rain storm and then measure the volume of water in the circular pan and the volume of water in the square pan. Suppose that the volume of water in the circular pan is V_c and the volume in the square pan is V_s. We ***expect*** that the ratio $V_c/(V_c + V_s)$ should be a good approximation of the area of the circle to the area of the square, assuming that the rain fell according to a ***uniform distribution*** from above, and this ratio should be $\pi/4$. (We will define a uniform distribution formally below. For now, if you are unfamiliar with the term, it is a probability distribution in which all outcomes of the random variable are equally probable.)

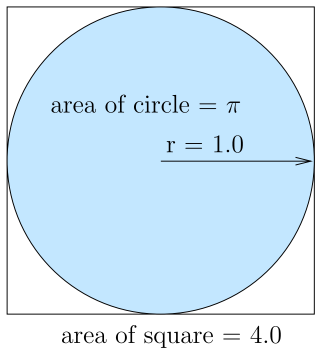

Figure 8.1: A circle in a square whose side equals its diameter. If the diameter is 2, then the square has area 4 and the circle has area π. The ratio of the circle's area to the square's is $\pi/4$.

Imagine now that we simulate the falling rain by generating a sequence of "random" locations within the square pan at which raindrops would fall. This is, in essence, a method of computing an approximation to π by generating random numbers, which is an application of the Monte Carlo method. Suppose that the pans are centered on the origin of the two-dimensional plane. Then a location within a square of side 2 centered on the origin is a point (x, y) such that $-1 \le x \le 1$ and $-1 \le y \le 1$. Some of locations will fall within the circle and some will not. To be precise, it is exactly those points (x, y) such that $x^2 + y^2 \le 1$ that lie within the circle; the others do not. If we count the number of randomly generated points within the circle, say c, and divide by the total number of points, n, the ratio c/n is an estimate of $\pi/4$, assuming that our random number generator really does generate the coordinates in some uniform, uncorrelated way.

Naturally, we can perform this same experiment by using just the part of the circle in the first quadrant, as shown in Figure 8.2. The ratio of the quarter circle to the square will still be $\pi/4$ and we can generate values of x and y that lie in the interval $[0, 1]$.

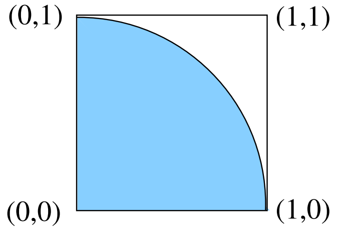

Figure 8.2: A quarter circle of unit radius in a square of unit width.

The ratio c/n is a statistical average. If we repeat the experiment with the same number of points we will get a different ratio because of the randomness of the numbers. However, as the number of sample points is increased, we should expect the estimate c/n to become more accurate, meaning closer in value to $\pi/4$, or stated another way, the ***absolute error*** to become smaller. The ***absolute error*** of an estimate is the absolute difference $|a - e|$ where a is the correct value of the quantity being approximated, and e is the computed estimate. In this particular example, we can obtain the actual value of π to any degree of precision that we choose, because it has been computed to millions of decimal digits, so we can compute the absolute error of the estimate to as much decimal precision as we choose. Using the C library's `random` function, we can write a simple sequential program to implement this algorithm. We choose `random` rather than `rand` because it is known that some implementations of the `rand` function may not produce sufficiently random-looking numbers. By running this program with different numbers of sample points, we can examine how the absolute error changes as the number of sample points is increased. We will do that shortly.

8.3 The Theory

In almost all applications of interest, the correct answer to the problem is not known in advance, otherwise we would not be seeking it in the first place! Therefore, when a Monte Carlo method is used to obtain an estimate, it is important to have a method of predicting the absolute error in advance. In order to predict the possible error, there needs to be an underlying theoretical model or foundation. The Monte Carlo method has such a foundation, and although it is beyond the scope of these notes to explore the theory in depth, it is necessary to state some basic results so that we have a way to understand its applicability and limitations.

The basic problem that Monte Carlo methods solve is the computation of ***expected values***. As we do not assume you know what an expected value is, we define it here. Let X be a discrete random variable whose values are $x_1, x_2, x_3, \ldots$. Then the expected value of X, denoted $E[X]$ (and sometimes denoted μ_x or μ when the meaning is clear) is defined by

$$E[X] = \sum_{i=1}^{\infty} x_i P\{X = x_i\} = lim_{N\to\infty} \sum_{i=1}^{N} x_i P\{X = x_i\} \tag{8.1}$$

If X can take on only finitely many values, then the summation in 8.1 would be finite. For example, suppose we have a fair six-sided die. The outcome of a roll of this die is one of the numbers from 1 to 6. Because it is fair, each has probability 1/6. Therefore, the expected value of a roll of this die is $1 \cdot (1/6) + 2 \cdot (1/6) + 3 \cdot (1/6) + 4 \cdot (1/6) + 5 \cdot (1/6) + 6 \cdot (1/6) = 3.5$. The expected value of a function of a random variable is similarly defined. To be clear, if $g(X)$ is some function of a random variable X, we define the expected value of $g(X)$ to be

$$E[g(X)] = \sum_{i=1}^{\infty} g(x_i) P\{X = x_i\}$$

For example, if there were a payoff associated with each value of a toss of the die, equal to the square of the face value of the roll, the function $g(x) = x^2$ would be the payoff and the expected value of the payoff would be

$$\sum_{x=1}^{6} x^2 \cdot (\frac{1}{6}) = \left(\frac{1+4+9+16+25+36}{6}\right) \approx 15.17$$

In Monte Carlo approximation, the goal of the computation is to compute an approximation of the expected value of some random variable. In the approximation of π described above, for example, we can define the random variable X to represent whether or not a random drop of rain falls within the circular pan, given that it falls within the square pan. Because either it does or it does not, an observation of X takes on one of two values: 1, if the raindrop is within the circle, and 0 if not. Assuming that every raindrop is equally likely, the expected value of a sequence of N raindrops would be

$$lim_{N\to\infty}\sum_{i=1}^{N} x_i \cdot \frac{1}{N}$$

where $x_i \in \{0,1\}$. The symbol S_N is often used to denote the value of a statistical guess based on N observations of the random variable. In this experiment,

$$S_N = \sum_{i=1}^{N} x_i \cdot \frac{1}{N}$$

The two important questions are

1. Will the statistical guess, $S_N = \sum_{i=1}^{N} x_i \cdot \frac{1}{N}$, of the expected value always converge to the real expected value as the sample size increases? In symbolic terms, does S_N converge to μ?
2. Can we establish the rate of convergence to the real expected value?

The answer to both of these questions is yes: the Monte Carlo estimate will converge to the expected value, and we can predict its rate of convergence. These questions are important because the Monte Carlo estimate would not be an indicator of the actual value if it did not converge to the expected value of the random variable, and being able to predict the rate of convergence allows us to predict the difference between the estimate and the real value in a probabilistic way (such as with a statement like, there is a probability greater than 0.95 that the estimate is within ϵ units of the actual value.) The positive answer to the first question is a result of the ***Weak Law of Large Numbers (WLLN)*** . The positive answer to the second question is a result of the ***Central Limit Theorem***. To explain these two important theorems, we need a definition.

Definition 1. A sequence of random variables is ***independently and identically distributed*** (***iid***) if each variable has the same probability distribution as the others and they are mutually independent of each other (the outcomes of each do not depend on the outcomes of the others.)

As an example, when you toss a fair coin ten times, the set of outcomes is iid because each new toss does not depend on previous tosses, and in each toss heads are as likely as tails each time. In other words, each toss is a separate trial and these trials are independent and have the same probability distributions.

The Weak Law of Large Numbers states that, if X_1,X_2,... is an independently and identically distributed sequence of random variables whose expected value is finite, then the sequence of sample means

$$S_n = \frac{1}{n}\sum_{k=1}^{n} x_k$$

converges in probability to the true expected value $E[X]$ as $n \to \infty$. This means that, as n gets larger, the probability that the value of S_n differs from the expected value $E[X]$ by a large amount approaches zero.

The Central Limit Theorem states that, if X_1,X_2,... is an independently and identically distributed sequence of random variables with positive, finite variance σ^2, then

$$\frac{X_1 + X_2 + \cdots + X_n - nE[X_1]}{\sqrt{n}\sigma} \tag{8.2}$$

converges[1] to the standard normal distribution, $N(0,1)$, as $n \to \infty$. (The standard normal distribution has a mean of 0 and a variance of 1.) Since

$$\frac{X_1 + X_2 + \cdots + X_n - nE[X_1]}{\sqrt{n}\sigma} = \frac{\frac{1}{n}\sum_{k=1}^{n} X_k - E[X_1]}{\sigma/\sqrt{n}} = \frac{\sqrt{n}}{\sigma}\left(\frac{1}{n}\sum_{k=1}^{n} X_k - E[X_1]\right) = \frac{\sqrt{n}}{\sigma}\left(S_n - \mu\right)$$

[1]More accurately, it weakly converges.

Eq. 8.2 is equivalent to the statement that as $n \to \infty$,

$$\frac{\sqrt{n}}{\sigma}(S_n - \mu) \Rightarrow N(0,1)$$

where $\Rightarrow N(0,1)$ means that the left-hand side is approximately normally distributed with mean 0 and variance 1. This in turn implies

$$S_n - \mu \approx \frac{\sigma}{\sqrt{n}} N(0,1) \tag{8.3}$$

The notation $X_n \Rightarrow N(0,1)$ means that X_n converges in distribution to $N(0,1)$. Intuitively, the limit of the distribution of the random variables X_n as n approaches infinity is the normal distribution $N(0,1)$. In effect, Eq. 8.3 can be interpreted as follows: if σ^2 is the variance of the random variables $X_1, X_2, \ldots X_n$, and μ is their true mean, and we define

$$\epsilon_n = S_n - \mu$$

then for n sufficiently large, ϵ_n is a normally distributed random variable with mean 0 and standard deviation $\sigma/\sqrt{(n)}$. As a consequence, the error tends to decrease in proportion to $1/\sqrt{(n)}$. Therefore to add a significant digit of accuracy requires increasing the number of samples by a factor of 100. This is not to say that it will always decrease at this rate or be bounded by any particular value. It means that there is a high probability that it will be bounded by a particular value. For example, for n sufficiently large, there is a 0.95 probability that

$$|\epsilon_n| \le 1.96 \frac{\sigma}{\sqrt{(n)}}$$

where σ is the standard deviation of the random variables.

Definition 2. A continuous random variable X is ***uniformly distributed*** with parameters (a, b) if all intervals of the same length within the interval $[a, b]$ are equally likely, and all intervals in $(-\infty, a)$ and (b, ∞) have zero probability. A continuous uniform distribution with parameters (a, b) is denoted $U(a, b)$. A ***discrete uniform distribution*** is a probability distribution in which each value of a discrete random variable X has equal probability.

A sequence of n random numbers $x_1, x_2, \ldots x_n$ generated by a random number generator can be viewed as a statistical sample of size n. If we can trust the random number generator, then this sequence behaves like a set of n independent and identically uniformly distributed random variables with finite variance. Hence, the Central Limit Theorem tells us that the value computed by a Monte Carlo approximation in which the random variables are obtained from a random number generator has a rate of convergence towards the true value that is proportional to $1/\sqrt{n}$.

8.4 Monte Carlo Integration

Suppose that f is a function of a single real variable. The ***Mean Value Theorem*** (for functions of a single variable) states that if f is an integrable function on the interval $[a, b]$ then

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

where $\bar{f}$ is the mean value of f on the interval. Another way to say this is that the area under the graph of $f(x)$ on the interval $[a, b]$ is equal to the area of a rectangle of height $\bar{f}$ and width $b - a$. From the results of Section 8.3, if the n random numbers $x_1, x_2, \ldots x_n$ are a statistical sample chosen uniformly from the interval $[a, b]$, then so are the values $f(x_1), f(x_2), \ldots f(x_n)$, and the statistic

$$\hat{f}_n = \frac{1}{n}\sum_{k=1}^{n} f(x_k)$$

converges to the mean value $\bar{f}$ of the function $f(x)$ over $[a, b]$. Therefore, by sampling the interval $[a, b]$ at n points, we can obtain n values of $f(x)$ and use the statistic

$$I \approx (b-a)\frac{1}{n}\sum_{k=1}^{n} f(x_k)$$

as an estimate of the area under the curve of the function f. The problem of estimating the value of π can be seen as an instance of this type of problem. Let $\phi(x, y)$ be the function defined on the unit square $0 \le x, y \le 1$ by

$$\phi(x,y) = \begin{cases} 1 & x^2+y^2 \le 1 \\ 0 & x^2+y^2 > 1 \end{cases}$$

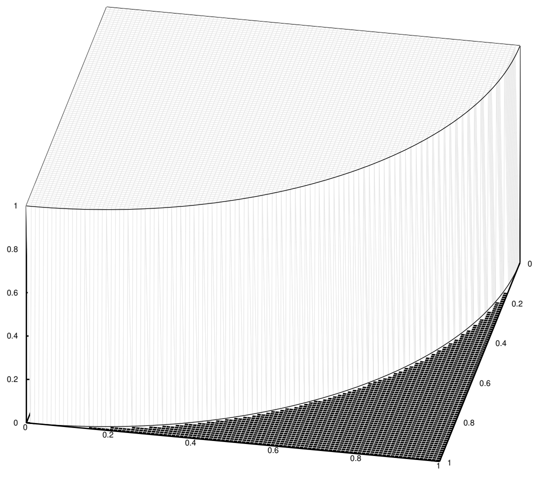

Figure 8.3: Graph of $\phi(x, y)$ on the unit square. The graph is a surface that has the value 1 within the quarter-circle and the value 0 outside of it. It makes a step along the quarter-arc of the circle.

The function $\phi(x, y)$ is a function of two variables, and its graph is not a curve, but a two-dimensional surface. Because $\phi(x, y)$ is piecewise-defined, and because it takes on only one of two values, 0 or 1, its graph looks like a curved step, as shown in Figure 8.3. Because the function is zero outside of the quarter-circle, its volume is zero there. Therefore, the volume under this surface is the volume of the one-quarter cylinder with radius 1 and height 1 and is therefore $\pi/4$. Although the Mean Value Theorem is defined only for functions of a single variable, it is possible to apply it twice to solve this problem. It turns out that because the function $\phi(x, y)$ is integrable over the unit square, it has a mean value $\bar{\phi}$ over the set of points within the unit square. Because the unit square has area of 1, applying the Mean Value Theorem twice we can show that the volume under the surface is

$$\int_0^1 \int_0^1 \phi(x,y)\,dx\,dy = \overline{\phi}$$

Therefore, a Monte Carlo estimate of $\overline{\phi}$ within this unit square is an estimate of the volume under the surface. Since a Monte Carlo estimate based on n randomly selected points (x_1, y_1) within the unit square is

$$S_n = \frac{1}{n}\sum_{k=1}^{n} \phi(x_1, y_1) \tag{8.4}$$

it follows that S_n is an estimate of $\pi/4$. This is what the algorithm we described above does; it computes an estimate of $\pi/4$ by sampling the function $\phi(x, y)$.

The sequential program in Listing 8.1 uses the algorithm we described to estimate the value of π, given the number of sample points as a command-line argument. Its output includes the actual estimate, the absolute error, and the 95% confidence interval limit on the absolute error for the given sample size and actual sample.

Listing 8.1: seq_monte_carlo_pi.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <errno.h>

#define  ZVAL_95  1.9599

int main ( int argc, char* argv[] )
{
    long    incircle = 0;    /* counts number of points in circle       */
    double  estimate;        /* estimated value of PI                   */
    double  x, y;            /* coordinates of random point             */
    long    counter = 0;     /* counts number of sample points          */
    long    num_samples;     /* number of sample points for estimate    */
    double  mean;            /* mean of computed function               */
    double  std_dev;         /* standard deviation of function          */
    double  error;           /* absolute error of estimate              */
    double  error_bound;     /* 95% 2-sided confidence interval         */


    if ( argc < 2 ) {
        printf("usage %s <number-of-sample-points>\n", argv[0]);
        exit(1);
    }

    /* Get number of samples from command line argument */
    errno = 0;
    num_samples = strtol(argv[1], '\0', 0);
    if (errno != 0 ) {
                    perror("strtol");
                    exit(EXIT_FAILURE);
    }
    /* Seed the random number generator with the current time */
    srandom(time(NULL));

    /* Generate the sequence, count the points in the circle */
    while ( counter < num_samples ) {
        x = (double) (random()) / RAND_MAX;
        y = (double) (random()) / RAND_MAX;
        if ( x*x + y*y  <= 1.0 )
            incircle++;
        counter++;
    }

    /* Compute the estimate, the error, and the statistics */
    mean        = incircle / (double) num_samples;
    std_dev     = sqrt( mean * (1.0 - mean) / (num_samples - 1 ) );
    error_bound =  4.0 * ZVAL_95 * std_dev;
    estimate    =  4.0 * mean;
    error       = fabs(estimate - M_PI);

    printf("num_samples = %ld\testimate = %12.8f\t", num_samples, estimate);
    printf("abs_error = %8.7f\t95%%_error_bound = %8.7f\n", error, error_bound);
    return 0;
}
```

We ran this program ten times at each sample size, increasing the sample size by a factor of ten, from $N = 10^3$ to $N = 10^9$, and we display the results in Table 8.1. Observe from the table that the actual error decreases roughly in proportion to the calculated confidence limit, and is smaller in every case.

Sample Size (N)	**Estimated Value** S_N **of** π	**Absolute Error** ($\epsilon_N = \lvert\pi - S_N\rvert$)	**95% Confidence Limit on Error** $z_{0.95}\sigma/\sqrt{N}$
1000	3.16400000	.02240730	.10085440
10000	3.12200000	.01959300	.03245200
100000	3.13968000	.00191300	.01018700
1000000	3.14332400	.00173100	.00321600
10000000	3.14163500	.00037340	.00101800
100000000	3.14156970	.00013840	.00032200
1000000000	3.14159880	.00003050	.00010200

Table 8.1: Progression of estimates of the value of π using Monte Carlo estimation.

A rate of convergence of $1/\sqrt{n}$ is not a very fast rate of convergence in general. There are methods of estimating π that converge much more rapidly than this. In general, Monte Carlo methods are not the best choice when the state space of the problem has low dimension, but when the dimension of the space, i.e., the number of integrands, is six or more, there are no deterministic methods that can be used with success, and Monte Carlo methods are the best option.

8.5 Parallel Algorithms and Monte Carlo Methods

Monte Carlo algorithms are usually easy to convert to parallel algorithms because there are natural data decompositions of the problems that lead to a set of processes that have little or no inter-process communication. When this can be done, a program with p processes can find the solution almost p times faster or to obtain an answer with $\Theta(\sqrt{p})$ greater accuracy in the same amount of time. However, both of these assertions depend on the *quality* of the random number sequence, and for that reason, before we begin a study of parallel Monte Carlo algorithms, we summarize the important aspects of random number sequences and their generation.

8.6 Sequential Random Number Generators

The success of any method of computing that relies on a stream of random numbers depends on the quality of the random number generator itself. Technically, what we usually call a random number generator is a ***pseudo-random number generator***, (***PRNG***) because a deterministic algorithm is usually used to generate the sequence, and therefore, by many people's definitions, the numbers are not truly random. On the other hand, as Knuth [7] said, "being 'apparently random' is perhaps all that can be said about any random sequence anyway" and so we will continue to use the term random number generator as it generates sequences that appear to be random.

There are generators that use physical, random events to generate numbers, such as recording charged particle emissions in a vacuum tube and using the times of emission to generate a random sequence[2]. In fact the earliest random number generators were physical machines, and one of these (named ERNIE) was used to pick winning lottery tickets in Great Britain[5]. However, most modern generators use various algorithms and are not physical.

[2]These times follow a Poisson distribution, but they can be converted to a uniform distribution, as we will see in a subsequent section.

What properties should a random number generator have? Although Coddington[2] identifies nine properties of an ideal random number generator, no generator can have them all. We list some realistic properties that a good generator should have. The numbers

1. are uniformly distributed,
2. are uncorrelated,
3. satisfy certain standard statistical tests for randomness, such as the χ^2 test and the spectral test[3],
4. have a very long period (they take a long time before they repeat),
5. are reproducible (so that debugging is easier), and
6. are easy to compute.

Condition (3) is not enough if the random numbers are to be used in a critical way. In this case, the numbers should be tested more thoroughly. For example, they can be supplied to a Monte Carlo simulation of a problem that has an exact solution and the results compared.

For parallel computing, we require one additional property:

7. The numbers can be split easily into independent subsequences.

Modern random number generators fall into several different categories, but the most common are

- linear congruential generators
- shift-register feedback
- lagged Fibonacci
- combined generators

Shift-register feedback generators are now known to have poor qualities and will not be discussed, nor will combined generators.

8.6.1 Linear Congruential Generators

Linear congruential generators (***LCG***), first introduced by D. H. Lehmer in 1949 [8], generate a sequence of numbers u_0, u_1, $u_2, \ldots$ using the formula

$$u_{i+1} = (au_i + c)\, mod\, m$$

where the initial value u_0 is called the ***seed***, a is called the ***multiplier***, c is called the ***increment***, and m is the ***modulus***. Of necessity, the sequence of numbers will eventually cycle, because m is finite, and there are at most m different moduli (0 to $m-1$); as soon as a number repeats, the entire sequence repeats, because if $u_j = u_k$, then clearly $(au_j + c)\, mod\, m = (au_k + c)\, mod\, m$. If the choice of a, c, and m is poor, the period will be small, and if it is good, it will be of length m, the maximum it can be. Obviously, the larger m is, the longer the period can be, not of necessity, but if the particular choice of m is a good one. (We can have a very large m but a very small period if the choice of multiplier and increment is bad.) We summarize the various choices of parameters, both the good ones and the ones to be avoided.

[3]The χ^2test is a standard statistical test that, roughly stated, measures the deviation of a sample from its expected deviation. The spectral test is a test of a sequence of random numbers that treats them as a sequence of vectors of a given length n. Many random number generators generate sequences such that, when the vectors are viewed as belonging to the n-dimensional hypercube, they form a lattice. The spectral test measures the maximum distance between the planes that they form in this lattice.

8.6.1.1 When $c = 0$ (Multiplicative Congruential Method)

When $c = 0$, the linear congruential method is called a ***multiplicative congruential method***, because it is purely a multiplication, as it is of the form

$$u_{i+1} = au_i \, mod \, m$$

It is computational slightly faster than when $c \neq 0$, as an operation is eliminated. Lehmer's original idea set $c = 0$, and it was only later realized that the cycle can be longer when $c \neq 0$. When $c = 0$, the maximum period possible is attained when

- u_0 is relatively prime to m, and
- a is a ***primitive element*** modulo m

We explain the idea of a primitive element. Pick any number u, $0 \leq u < m$ that is relatively prime to m. Consider the powers of u modulo m: $u^2 \, mod \, m$, $u^3 \, mod \, m$, ... and so on. Because there are at most m distinct values modulo m, by the pigeon-hole principle, for some power k of u, $u^k \, mod \, m$ must be a repeat of some $u^j \, mod \, m$, $j < k$. Therefore, we have $u^k \, mod \, m = u^j \, mod \, m$, so that

$$(u^k - u^j) \, mod \, m \quad = u^j(u^{k-j} - 1) \, mod \, m = \quad 0$$

which is another way of saying that $u^j(u^{k-j} - 1)$ is a multiple of m. Because u is relatively prime to m, so is u^j, and the only way that the product $u^j(u^{k-j} - 1)$ can be a multiple of m is if $(u^{k-j} - 1) \, mod \, m = 0$. This implies that $u^{k-j} \, mod \, m = 1$. For each u that is relatively prime to m there is a smallest number k such that $u^k \, mod \, m = 1$. This is called the ***order of** u **modulo*** m. For example, the order of 4 modulo 15 is 2 because $4^2 \, mod \, 15 = 16 \, mod \, 15 = 1$ and obviously this is the smallest power for which this is true. There will always be a number u that is relatively prime to m whose order is the largest among all possible values $0, 1, \ldots, m - 1$. Such an u is called a primitive element of m.

If m is a prime number, then the primitive elements will all have a period of length $m - 1$, so making m the largest prime that can fit in the machine word is a good choice. Not all numbers will have maximum period. The constraints are limiting when m is prime and there are various tables of values that can be used.

If m is a power of 2, then the period is maximal when a is a number for which $a \, mod \, m = 3$ or $a \, mod \, m = 5$, but this is a bad choice of m because there will be subsequences whose low-order bits cycle with very small period.

Summary. The best choice when $c = 0$ is to make $m = 2^w - 1$, where w is the word size of the computer, and if m is not prime (e.g., $m = 63$) then choose a suitable value for a to maximize the period.

8.6.1.2 When $c \neq 0$

When $c \neq 0$, the modulus m should be as large as possible to maximize the period. The LCG defined by the parameters m, a, and c will have a period of length m if and only if

1. c must be relatively prime to m
2. for every prime number p that divides m, $a - 1$ must be a multiple of p, and
3. if m is a multiple of 4, then $a - 1$ must be a multiple of 4.

This is stated without proof. See Knuth [7] for its proof. As an example, take $m = 8$. If $a = 3$ and $c = 3$, then the choice of a does not satisfy (3) because 8 is a multiple of 4 but $a - 1$ is not a multiple of 4. The sequence starting with $u_0 = 0$ would be $(0, 3, 4, 7, 0, 3, 4, 7, \ldots)$. On the other hand, if $a = 5$ and $c = 3$, then all conditions are satisfied. No matter what choice of u_0, the cycle length will be 8. Take $u_0 = 0$ for example. The sequence would be $(0, 3, 2, 5, 4, 7, 6, 1, 0, \ldots)$.

8.6.1.3 Synopsis

Linear congruential generators are found in many software libraries. The `rand` function from the *C* library uses an LCG. The problem with `rand` and with many LCGs is that they exhibit correlations of the low-order bits of the generated numbers. If you were to take the values produced by `rand`, for example, and divide by `RAND_MAX`, so that the resulting numbers were in the interval $[0, 1]$, and then form n-tuples of the form $(u_i, u_{i+2}, \ldots, u_{i+n})$ for many values of i, and plot these inside a unit hypercube of dimension n, you would see a crystalline lattice (a regular n-dimensional pattern) because of the repeated pattern of the low-order bits.

Those LCGs that use a modulus that is a power of two, such as `drand48`, produce numbers with highly-correlated low-order bits and should be avoided. They also produce long-range correlations, which we will see can be a problem when used for parallel processes sharing the generator. Those that use prime moduli are safer in general.

A period of 2^{31}, which is roughly 2 billion, is not very large, given the demands of scientific problems to be solved, which need far many more random variables. Since many architectures are 64 bits, larger choices of modulus are better.

Summary. On a 64*-bit machine, the values* $a = 2862933555777941757$*,* $c = 3037000493$*, and* $m = 2^{64}$ *produce a maximal,* 2^{64}*-length period, although the low-order bit correlations will still be present. In general,* 32*-bit LCGs should be avoided when there are alternatives.*

8.6.2 Lagged Fibonacci Generators

A ***lagged Fibonacci generator*** (LFG) generates a sequence of numbers u_0, u_1, $u_2, \ldots$ using the formula

$$u_i = u_{i-p} \odot u_{i-q}$$

where p and q are called the ***lags***, $p > q$, and $\odot$ is any binary arithmetic operator, such as addition, subtraction, or multiplication, modulo m for some modulus m, or bit-wise exclusive-or, in which case the modulus is not required. If the operator is addition modulo 2^w, where w is the word size of the computer, you can see that this is a generalization of the Fibonacci sequence; it would be in effect $u_i = u_{i-p} + u_{i-q}$. This method requires storing the p previous values in an array called the ***lag table***. The larger p is, the more memory is needed for the lag table. It also requires creating p seed values rather than 1. An LFG is completely determined by the parameters p, q, m, and the operator $\odot$. For example the `random` function from the C library is an LFG with parameters $p = 31$, $q = 3$, $m = 2^{31} - 1$ and addition modulo m. Its lag table is by default of size 31[4].

Unlike LCGs, LFGs can have an extremely large period. With an LCG, as soon as one of the random numbers that was generated before is generated again, the sequence from that point will be repeated all over again. With an LFG, the next number depends on two previous values. The fact that $u_k = u_j$ for some $k > j$ does not imply that $u_{k+1} = u_{j+1}$ because $u_{k+1} = u_{k+1-p} \odot u_{k+1-q}$, which will not be of necessity equal to $u_{j+1-p} \odot u_{j+1-q}$. In fact, if $m = 2^b$, and p and q are suitably chosen[5], then the period is

- $2^p - 1$ if the operation is XOR
- $(2^p - 1)2^{b-1}$ if the operation is addition or subtraction, and
- $(2^p - 1)2^{b-3}$ for multiplication.

For example, the period for the *C* library `random` function on a 32-bit processor with the default lags is $2^{31} \cdot 2^{31} = 2^{62} \approx 4.6 \times 10^{18}$.

[4]The man page for `random` states that it is a non-linear additive generator; this is incorrect!
[5]p and q must be the exponents of a primitive polynomial.

8.6.2.1 Choosing LFG Parameters

Unfortunately, the length of the period alone is not the only factor in the quality of a random number generator. There may be unacceptable correlations even with LFGs with long periods. Coddington[2] has very specific recommendations for the choice of the LFG parameters, which we summarize here.

- Multiplicative LFGs have better randomness properties than additive LFGs in general and are preferred, especially for integer sequences, given the current speeds of multiplication circuits within modern processors. XORs should be avoided.
- Lags should be at least 1000 for additive LFGs; the choice $p = 1279$, $q = 1063$ is recommended, provided memory is not an issue.
- Lags for multiplicative LFGs should be at least 100; 127 is a recommended value for p.
- It is best if the seeds for the LFG are generated randomly by a different type of generator, such as an LCG. The *C* `random` function, for example, uses an LCG with parameters $u_0 = 1$, $a = 16807$, $c = 0$, and $m = 2^{31} - 1$ to generate the 31 seeds needed for the default initial state.

8.7 Parallel Random Number Generators

Parallel algorithms that use random numbers to solve a problem need a method of generating random numbers for multiple processes. This adds a new set of requirements to the method of random number generation. Ideally, the following conditions should be satisfied.

- The generator algorithm must work for any number of processes. In other words, it should not be designed so that there is a limit on how many processes can use the algorithm.
- The sequence seen by any individual process should satisfy the requirements of a good sequential random number generator for each process. This means, at the very least, that the sequence has a long period, that it has little, if any, correlations, and that the numbers are uniformly distributed.
- The sequences seen by different processes should not have any correlations between them. For example, if two processes had sequences such that the k^{th} number in one process's sequence was always double that of the second process's sequence, this would be a violation of this requirement.
- The sequence that is produced should be independent of the number of processes. There are a few reasons for this. One is that the solution obtained by a parallel algorithm using random numbers should be the same no matter how many processors are used. Another is that, in order to debug a parallel program, it is often better to drop down the number of processes, and if the random number sequence is different for the smaller number of processes, then the results may not be repeatable.
- The generation algorithm should not require much, if any, data movement among processes. Once the generator is seeded and its initial state created, each process should generate its own stream of numbers independently of the others.

Most parallel random number generators work by using a sequential generator and distributing the numbers it generates among the parallel processes in one way or another. Some methods that were once very common, such as the manager-worker method, are not in use any more and are not described here. The three most common methods in use are

1. ***Leapfrogging***. This is a cyclic partitioning of the numbers to the processes, just like a deck of cards being dealt to players.
2. ***Sequence splitting***. In this method, runs of consecutive numbers are given to each process in turn, analogous to giving each card player his or her full hand, one after the other.

3. ***Independent sequencing***. In this method, the initial seeds and sometimes the choices of parameters are different for each process, so that in effect, the single generator acts like multiple generators, one for each process, producing independent streams for each.

Each of these methods has weaknesses, and in general parallel random number generation is fraught with dangers. We will make specific recommendations after reviewing the choices.

8.7.1 The Leapfrog Method

In this method, each process uses the same sequential random number generator to produce a sequence of numbers, but the different processes take numbers from the sequence in a cyclic allocation, in the same way as cards are dealt in a card game. To be precise, if there are p processes, then the process with rank r takes the subsequence

$$u_r,\ u_{r+p},\ u_{r+2p}, \ldots$$

This method is called the ***leapfrog method*** because each process leapfrogs over the other p processes to grab its next number from the generator. Figure 8.4 shows how process 3 in a set of 6 processes takes the elements u_3, u_9, u_{15}, u_{21} and so on from the stream.

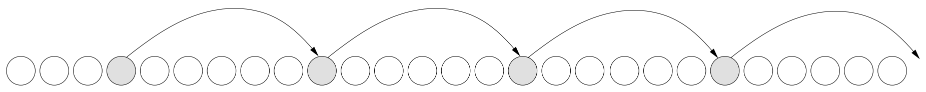

Figure 8.4: The leapfrog method of parallel random number generation. There is one stream of random numbers and each process gets a subsequence of this stream by taking every p^{th} value in a cyclic fashion.

The practical issue is how each process can jump ahead in the sequence to get its next number. In other words, if process r has just used u_r and now needs u_{r+p}, how can it generate it efficiently? It would be very inefficient for it to generate all of the numbers in between and just throw them away. When the sequential generator is an LCG, it is relatively easy, as the following theorem shows.

Theorem 3. *Let a linear congruential generator have parameters a, c, and m. Then*

$$u_{k+p} = (a^p u_k\ + c(a^p - 1)/(a-1))\ mod\, m$$

Proof. Formally, this can be proved by induction on p. When $p = 1$, it reduces to

$$u_{k+1} = (au_k + c)\ mod\, m$$

which is true by definition. Assume it is true for $p-1$. Then,

$$\begin{aligned} u_{k+p} &= (au_{k+p-1} + c)\ mod\, m \\ &= \left(a\left(a^{p-1}u_k + c\left(a^{p-1} - 1\right)/(a-1)\right) + c\right)\ mod\, m \\ &= \left(a^p u_k + c\left(a^p - a\right)/(a-1) + c\right)\ mod\, m \\ &= \left(a^p u_k + c\left(\left(a^p - a + a - 1\right)/(a-1)\right)\right)\ mod\, m \\ &= (a^p u_k\ + c(a^p - 1)/(a-1))\ mod\, m \end{aligned}$$

which shows it is true for p. We implicitly used two properties of modulo arithmetic, namely

- $(a(b\, mod\, m))\ mod\, m = ab\, mod\, m$
- $(a + (b\, mod\, m))\ mod\, m = (a+b)\, mod\, m.$

□

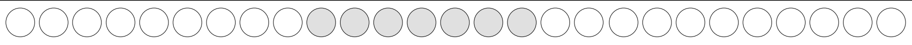

Figure 8.5: The sequence splitting method of parallel random number generation. There is one stream of random numbers and each process gets a consecutive sequence of this stream by taking the next n values from the stream. The value of n is determined by the application.

The theorem implies that each process can use the same algorithm to generate its sequence. Let us denote the sequence of values that process r generates by $u_0^{(r)}, u_1^{(r)}, u_2^{(r)}, \ldots$. Then this sequence is defined as follows:

- For every process r, $u_0^{(r)} = u_0$. In other words, every process seeds the generator with the same seed, u_0.
- For every process r, $u_1^{(r)} = u_r = a^r u_0 + c(a^r - 1)/(a - 1)$. In other words, each process obtains its own distinct first random number by leapfrogging to u_r.
- Each process generates the remainder of its sequence using the formula

  $$u_{k+1}^{(r)} = Au_k^{(r)} + C \, mod \, m$$

  where $A = a^p$ and $C = c(a^p - 1)/(a - 1)$. Notice that A and C only need to be computed once and stored as variables.

This ensures that each process has a random sample from the same uniform distribution but has its own sequence.

Problems with leapfrogging include:

- Although the original LCG may pass the spectral test, when a is replaced by $A = a^p$, the parallel RNG will fail it.
- When the modulus is a power of 2, there are correlations between numbers that are powers of 2 apart in the sequence, which can become magnified in the parallel generator.
- The sequence that each process sees will change when the number of processes is changed.

Coddington discourages the use of leapfrogging using LCGs, but recommends that if it going to be used, the modulus should be a prime number and at least 2^{48}. Alternatively, leapfrogging can be used with a lagged Fibonacci generator, but this is more computationally intensive.

8.7.2 Sequence Splitting

In sequence splitting, each process gets a consecutive sequence of numbers that the sequential generator generates, as shown in Figure 8.5. Several different strategies for dividing up the sequence have been tried. One strategy gives each process a sequence of length $l = m/p$, where m is the period of the generator and p is the number of processes. To use this method, each process would have to jump ahead to its starting place in the sequence. If the sequential generator is an LCG, the same method could be used as we described for leapfrogging. If it is an LFG, although it is more time-consuming to jump ahead than with an LCG, it only has to be done once for each process.

This method has the potential for inter-process correlations, because the long-range correlations in the original sequence may become manifested as correlations between the sequences of parallel processes. A second disadvantage is that the sequence seen by each process will change if the number of processes is changed.

8.7.3 Independent Sequencing

In both leapfrogging and sequence splitting, processes have to jump ahead in the sequence to get their numbers. A method that avoids this, and also works for any number of processes without changing the sequences seen by the different processes, is to use an LFG but assign a different initial state, i.e., a different lag table, to each process. Each process uses the same lag values but has a different set of starting seeds and so generates its own unique sequence, with the same distribution. If the number of processes is increased, new initial states are generated for them.

The quality of this method depends critically on the method of initializing the lag tables for the different processes. The best way to do this is to use a different random number generator to seed the tables. One approach is to use an LCG to initialize the lag tables. Another is to use a different sequential LFG, with different lags than the parallel LFG. In either case *it is recommended that a large lag be used for the parallel generator, to ensure that the sequences seen by the different processes do not overlap.* Of course the problem with giving each process its own lag table and also having a large lag value is that the amount of memory required by the parallel random number generation might be too large.

8.8 Generating Non-Uniform Random Variables

Random number generators generate uniformly distributed random variables, but very often we need to solve a problem in which the random events are not uniformly distributed, but are instead variates of some other probability distribution. For example, what if we wanted to simulate the random toss of two fair dice. The outcomes are numbers between 2 and 12, but these are not equally likely. The following table contains the actual probabilities associated with the face value of the toss of two dice. The top row is a list of possible outcomes x_i of the random variable X; the bottom row is the probability p_i of the given outcome

Face Value (x_i)	2	3	4	5	6	7	8	9	10	11	12
Probability (p_i)	$\frac{1}{36}$	$\frac{2}{36}$	$\frac{3}{36}$	$\frac{4}{36}$	$\frac{5}{36}$	$\frac{6}{36}$	$\frac{5}{36}$	$\frac{4}{36}$	$\frac{3}{36}$	$\frac{2}{36}$	$\frac{1}{36}$

This is an example of a problem in which the random variable X comes from a ***discrete probability distribution***, which more generally is of the form

$$P\{X = x_i\} = p_i,\ 1 \le i \le n \tag{8.5}$$

To simulate tossing the dice, the respective outcomes must occur with their respective probabilities. To be precise, we need to be able to repeatedly generate random numbers x_i, $2 \le x_i \le 12$, such that each x_i has probability p_i of being generated. For an arbitrary discrete probability distribution, one way to solve this problem is by repeatedly generating a random number u in the interval $[0, 1]$ uniformly and setting the value of X depending on the value of u, using the following piece-wise defined function of u:

$$X = \begin{cases} x_1 & 0 < u \le p_1 \\ x_2 & p_1 < u \le p_1 + p_2 \\ x_3 & p_1 + p_2 < u \le p_1 + p_2 + p_3 \\ \vdots & \\ x_n & p_1 + p_2 + \cdots + p_{n-1} < u \le p_1 + p_2 + \cdots + p_{n-1} + p_n = 1 \end{cases}$$

With this scheme, the lengths of the successive intervals are p_1, p_2, $p_3, \ldots$, p_n and so each x_i has probability p_i of being generated. The probability distribution of X will be that of Eq. 8.5. For the random variable representing the face value of a toss of the pair of dice, the function would be defined by

$$X = \begin{cases} 2 & 0 < u \leq 1/36 \\ 3 & 1/36 < u \leq 3/36 \\ 4 & 3/36 < u \leq 6/36 \\ 5 & 6/36 < u \leq 10/36 \\ 6 & 10/36 < u \leq 15/36 \\ 7 & 15/36 < u \leq 21/36 \\ 8 & 21/36 < u \leq 26/36 \\ 9 & 26/36 < u \leq 30/36 \\ 10 & 30/36 < u \leq 33/36 \\ 11 & 33/36 < u \leq 35/36 \\ 12 & 35/36 < u \leq 1 \end{cases}$$

We can generalize this idea, but first we need some background definitions.

Definition 4. A ***cumulative distribution function*** (***CDF***) describes the probability that a real-valued random variate X with a given probability distribution takes on a value less than or equal to a number x. The CDF of a random variable X, denoted $F_X(x)$ is given by

$$F_X(x) = Pr\{X \leq x\}$$

The CDF of a random variable is always a non-decreasing function. If the variable is a continuous random variable, then the CDF is also continuous and represents the area under the curve of the ***probability density function*** (***pdf***)[6] of the variable from $-\infty$ to x, i.e., it is the definite integral of the pdf. If $f_X(x)$ denotes the pdf, then

$$Pr\{X \leq x\} = F_X(x) = \int_{-\infty}^{x} f_X(t)dt$$

and the probability that a random variable lies in an interval $[a, b]$ is $Pr\{a \leq X \leq b\} = \int_a^b f_X(t)dt$.

Definition 5. Given a non-decreasing function $F(x)$, the ***generalized function inverse*** of F, denoted F^{-1}, is defined as

$$F^{-1}(y) = \inf\{x \,|\, F(x) \geq y\}$$

where inf is the ***infimum*** of the set, also called the ***greatest lower bound*** of the set.

If the function $F(x)$ is continuous and strictly increasing, then the generalized inverse is the inverse of the function in the usual sense, namely that

$$F^{-1}(F(x)) = F(F^{-1}(x)) = x \tag{8.6}$$

for any x.

The CDF F of any probability distribution is always a non-decreasing function, which implies that it always has a generalized inverse. That inverse may not always be one that we can easily derive, a problem we will address below. For now, we show how we can use this generalized inverse as a means of generating random variables that are not uniformly distributed.

[6]The pdf of a continuous random variable is a function whose value at any given sample can be interpreted as providing a relative likelihood that the value of the random variable would equal that sample. It is the analog in continuous space to that of a probability mass function in the discrete case.

Let $F_U(u)$ denote the CDF of the uniform distribution over the interval $[0,1]$, $U(0,1)$. Then $F_U(u)$ is defined piecewise by

$$F_U(u) = \begin{cases} 0 & u < 0 \\ u & 0 \le u \le 1 \\ 1 & 1 < u \end{cases}$$

Thus, on the interval $[0,1]$, $F_U(u) = u$. We can state our problem formally as follows.

Problem. Given a cumulative distribution function $F(x)$, and a sequence of independent and identically distributed uniform random variables U_1, U_2, $,U_3, \ldots$, from $U(0,1)$, generate a sequence of random variables X_1, X_2, $,X_3, \ldots$, that are governed by the CDF $F(x)$.

8.8.1 The Method of Inversion

To generate random variables whose CDF is some function $F(x)$, we do the following:

1. Generate a uniformly distributed random variable over the interval [0,1].
2. Return the number $X = F^{-1}(u)$.

This algorithm is known as the ***inversion method***. We now justify why it works, how to use it in practice, and when it should not be used.

Let U be a $U(0,1)$ random variable. Then $F_U(u) = u$ if $0 \le u \le 1$. Recall that the meaning of this statement is that $Pr\{U \le u\} = u$. Suppose that $F_G(x)$ represents the CDF of the random variate that we get by the above procedure. In other words, when we apply steps 1 and 2 above, we generate a random number u uniformly in the interval $[0,1]$ and then return $F^{-1}(u)$. The CDF F_G is the CDF of the set of numbers so generated. Then by definition

$$\begin{aligned} F_G(x) &= Pr\{F^{-1}(U) \le x\} \\ &= Pr\{FF^{-1}(U) \le F(x)\} \\ &= Pr\{U \le F(x)\} \\ &= F_U(F(x)) \end{aligned}$$

Since U is $U(0,1)$, (which means that $F_U(v) = v$ for any $v \in [0,1]$, this last line implies

$$F_G(x) = F_U(F(x)) = F(x)$$

We have established that the set of numbers that this procedure generates follows the probability distribution defined by the CDF F.

8.8.1.1 Continuous Example: The Exponential Distribution

The ***exponential distribution*** is a continuous probability distribution that is associated with various kinds of natural processes such as

- the decay of radioactive atoms,
- the length of time between successive arrivals of cosmic ray alpha particles,
- the length of time between successive electrical impulses generated by neurotransmitters,

and also associated with less natural events such as

- the time to failure of an electrical or other component in a complex machine,
- the duration of a telephone call,

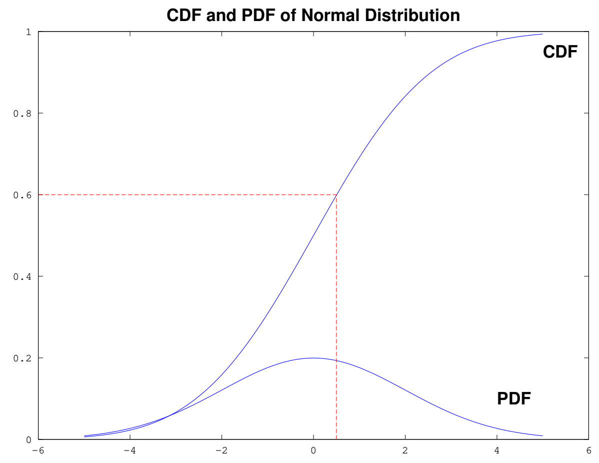

Figure 8.6: A probability density function (pdf) and its cumulative distribution function (CDF), in this case a normal distribution $n(x;\mu,\sigma)$, and the inverse mapping from the interval $[0,1]$ to a sample from $n(x;\mu,\sigma)$.

- the length of time before the next customer arrives to be serviced at a service center, and
- the length of time before the next car arrives at an entrance ramp of a highway.

In general, the exponential distribution is characterized by the fact that it is the only continuous ***memory-less*** distribution: the probability that an event will not occur within the next s time units from the current time given that it has not occurred within the past t time units is the same as the probability that it will not occur within the next s time units, regardless of whether or not it occurred within the past t time units. It has no memory of the past, so to speak.

The exponential distribution has a single parameter, λ, which is the inverse of its mean, and its probability density function (pdf) is

$$f(x) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

Its CDF is obtained by solving

$$F(x) = \int_0^x \lambda e^{-\lambda t} dt$$

and is therefore

$$F(x) = \begin{cases} 1 - e^{-\lambda x} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

Figure 8.7 shows a graph of the pdf and the CDF of an exponential distribution with mean 3.0 (i.e., parameter $\lambda = 1/3$). The inverse of an exponential CDF is $F^{-1}(u) = -\ln(1-u)/\lambda$, which has the same distribution as $F^{-1}(u) = -\ln(u)/\lambda$, because u is uniformly distributed over $[0,1]$. (The distribution of u in $[0,1]$ is the same as the distribution of $1-u$ in $[0,1]$.) Therefore, to simulate a sequence of numbers from an exponential distribution with parameter λ, we can generate random numbers u_1, u_2, u_3, ... from $U(0,1)$ and then return the sequence $-\ln(u_1)/\lambda, -\ln(u_2)/\lambda, -\ln(u_3)/\lambda, \ldots$ and so on, which will be distributed exponentially. Listing 8.2 contains C functions that can be used for generating such a sequence.

Listing 8.2: A module for generating exponentially-distributed random numbers.

```
/*  This function initializes the C library random() function with a much larger
    lag table than the default and seeds it. The argument is the size of the lag
    table. It allocates memory for the lag table. */
char*  init_sequence( int state_size )
```

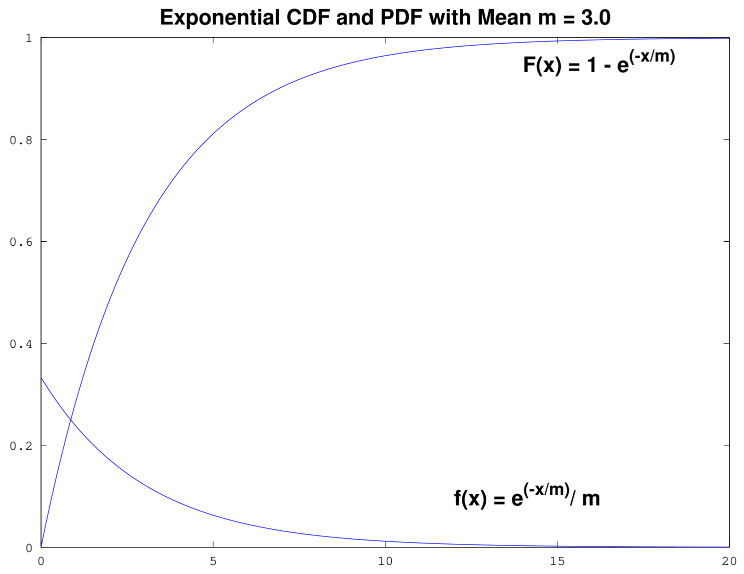

Figure 8.7: Graph of the exponential distribution pdf and cdf with mean 3.0.

```
{
    char * state;
    state  = (char*) malloc ( state_size * sizeof(char));
    if ( NULL != state )
        initstate(time(NULL), state, state_size);
    return state;
}

/* This function must be called to free the memory allocated by init_sequence. */
void   finalize ( char* state )
{
    free (state );
}

/* This function generates a number that is exponentially distributed with
   parameter lambda. */
double gen_number( double lambda )
{
    double u = (double) (random()) / RAND_MAX;
    return ( - log ( u ) / lambda );
}
```

If we want to simulate a single server system with arriving customers, and the customer arrivals obey an exponential distribution with parameter 0.2 (a mean of 5 minutes to the next arrival), we can generate a sequence of inter-arrival times using this algorithm and create a table of arrival times. If the following 9 times-to-next-arrival are so generated:

3.13, 6.60, 7.30, 0.44, 0.15, 3.29, 11.82, 3.75, 7.47

then the table of arrival times would be

Customer	Inter-arrival Time	Time of Arrival
1	0	0
2	3.13	3.13
3	6.60	9.73
4	7.30	17.03
5	0.44	17.47
6	0.15	17.62
7	3.29	20.91
8	11.82	32.73
9	3.75	36.48
10	7.47	43.95

The average of these 9 inter-arrival times is about 4.83, very close to 5. The longer the sequence, the closer to the mean the average will be.

8.8.1.2 Discrete Example: The Geometric Distribution

The ***geometric distribution*** is a discrete probability distribution that models the probability that a given number of failures occur before a success when the random event is a ***Bernoulli trial***. A Bernoulli trial is a random event with two possible outcomes, 1 and 0, representing success and failure respectively. The Bernoulli probability distribution with parameter p of success is

$$Pr\{X = x\} = \begin{cases} p & x = 1 \\ 1-p & x = 0 \end{cases}$$

It is easy to generate random variables from this distribution:

1. Generate a random variable U from $U(0,1)$.
2. If $U < p$ then return 1 and otherwise return 0.

A classic example of the geometric distribution involves a gambler betting at a roulette wheel. Imagine a gambler in the Monte Carlo casino who places a bet each time on the same number until he wins for the first time, increasing the amount of the bet each time. Assuming that the wheel is a fair wheel and is not rigged in any way, there is a fixed parameter of success p with each spin. Let the random variable X be the number of bets he places up to and including his winning bet. If he increases the bet by a large enough amount each time, he may make up for his losses when he finally wins, assuming that he has enough money to do this. The geometric distribution can associate a probability with how much he might win or lose. The geometric distribution is the only discrete distribution that is memory-less: the probability of a success in the next trial is independent of whatever events have happened so far. Most people find this counter-intuitive.

A geometric distribution with parameter p is defined by the probability function

$$Pr\{X = k\} = p(1-p)^{k-1} \quad k = 1, 2, 3, \ldots$$

The parameter p is the probability of success of the Bernoulli trial. This definition of the geometric distribution is the probability that exactly k trials are necessary to get one success[7]. To generate random variables from this distribution using the method of inversion, we need the generalized inverse of its CDF. The CDF $F_X(x)$ of this distribution is

$$F_X(x) = 1 - (1-p)^x$$

The inverse, $F_U^{-1}(u)$, is easily derived to be

$$F_U^{-1}(u) = \frac{\ln(1-u)}{\ln(1-p)}$$

[7]An alternate form is the number of trials *until* the first success.

Since U is $U(0,1)$, this is distributed in the same way as the function

$$F_U^{-1}(u) = \frac{\ln(u)}{\ln(1-p)}$$

Therefore, the method of inversion for this distribution is

1. Generate a random variable U from $U(0,1)$.
2. Return $\ln(U)/\ln(1-p)$.

There is an alternative to this approach, if for some reason, the calls to the math library are to be avoided. Instead one can repeatedly generate Bernoulli-distributed random variables with parameter p, Y_1, Y_2, Y_3, ..., using the method described above, until one of these is a success. If Y_k is the first such that $Y_k = 1$, then the procedure returns k. Of course this is much less efficient than evaluating the fraction.

8.8.2 The Acceptance-Rejection Method

Inversion is an efficient solution when the function F^{-1} can be determined in closed form for the given CDF F. This is not always possible though, as the closed form does not always exist. For example, one of the most prevalent and important probability distribution functions is the ***normal distribution***, also called the ***Gaussian distribution***, denoted $N(\mu,\sigma)$. Its pdf is defined by

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

and has no closed form inverse. In particular, the pdf of the standard normal, $N(0,1)$, defined by

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

has no closed form inverse. The CDF of a normal distribution has no closed form either; it is defined by

$$F(x) = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{x} e^{\frac{(t-\mu)^2}{2\sigma^2}} dt$$

To compute the value of this function, the integral must be evaluated numerically. Because the CDF has no closed form generalized inverse, the inversion method cannot be used to simulate a sequence of normally distributed random variates. For normal distributions, the ***Box-Muller transformation*** can be used to generate random variates, but ***the acceptance-rejection method***, originally proposed by John von Neumann in 1947 [9], is a more general method that works for all probability distributions, and so in these notes we describe how to use it and why it works.

To use this method, we need to find a second probability density function $h(x)$ with two properties:

- it is easy to generate random variables from h, and
- there is a constant c such that $c \cdot h(x)$ ***majorizes*** $f(x)$, which means that for all x

  $$c \cdot h(x) \geq f(x)$$

  which implies that

  $$0 \leq f(x)/(c \cdot h(x)) \leq 1.$$

Because h is a probability density function, the area under its graph is 1:

$$\int_{-\infty}^{\infty} h(x)dx = 1$$

We call such a function a ***blanketing function***, because it acts like a blanket covering the function that we want to sample. The procedure is as follows:

1. Generate a random variable U from $U(0,1)$.
2. Generate a random variable Z from h (using the inversion method for example).
3. If $U \leq f(Z)/(c \cdot h(Z))$ then return $X = Z$ otherwise go back to step 1 and repeat these steps (which means that Z is rejected.)

Notes

- In step 3, when the procedure returns $X = Z$, we say that it ***accepts*** Z. If it does not return Z we say that it ***rejects*** Z.
- U and Z are independently distributed random variables from distributions $U(0,1)$ and h respectively. For clarity we will write h_Z as the density function for Z instead of h.
- Because Z is a random variable, so is the ratio $f(Z)/(c \cdot h_Z(Z))$, which has the property that for all Z, $0 \leq f(Z)/(c \cdot h_Z(Z)) \leq 1$. Furthermore $f(Z)/(c \cdot h_Z(Z))$ is independent of U.
- Let n be the number of times that steps 1 through 3 are executed in order to return a random variable X. Then this is a sequence of $n-1$ failures followed by a success, and n is a random variable itself from a geometric distribution with some parameter p we have yet to determine. The value of p is the probability of success, which in this case is acceptance, which occurs when $U \leq f(Z)/(c \cdot h_Z(Z))$ for a random Z chosen from the distribution h_Z. Below we establish that $p = 1/c$. Since a geometric distribution with parameter p has an expected value of $1/p$, the expected number of tries n is c. This implies that the smaller the value of c, the more efficient the procedure will be. Therefore the choice of function h_Z should be made to make c small.
- It is worth noting that this procedure is itself a Monte Carlo method for finding solutions to a specific problem, namely the generation of random variables that are from a given probability distribution.

8.8.2.1 Example 1

Suppose that we need random variables from the probability distribution whose CDF is defined by

$$F_X(x) = \begin{cases} 0 & x \leq 0 \\ \frac{1}{2}\left(x^5 + x^4\right) & 0 \leq x \leq 1 \\ 1 & 1 \leq x \end{cases}$$

The probability density function for this CDF is

$$f_X(x) = \frac{d}{dx}F_X(x) = \begin{cases} 0 & x \leq 0 \\ \frac{5}{2}x^4 + 2x^3 & 0 \leq x \leq 1 \\ 0 & 1 \leq x \end{cases}$$

We cannot find a closed form inverse for this CDF because we cannot solve the equation $y = \frac{1}{2}\left(x^5 + x^4\right)$ in terms of y. (It must be solved numerically.) Instead we use acceptance-rejection. We need a pdf $h(x)$ for which it is easy to generate random variables and for which there is a constant c such that $c \cdot h(x)$ majorizes $f_X(x)$. Notice that the function $f_X(x)$ is at most $9/2$ for all x. (It has a maximum at $x = 1$.) Let

$$h(x) = \begin{cases} 0 & x \leq 0 \\ 1 & 0 \leq x \leq 1 \\ 0 & 1 \leq x \end{cases}$$

This is just the pdf of the standard uniform distribution, $U(0,1)$. Then we let $c = 9/2$ and we have $c \cdot h(x) \geq f(x)$ for all x. The procedure is

1. Generate a random variable U_1 from $U(0,1)$.
2. Generate a second uniform random variable U_2 from $U(0,1)$.
3. If $U_1 \leq \frac{2}{9}(\frac{5}{2}U_2^4 + 2U_2^3)$ then return $X = U_2$ otherwise go back to step 1 and repeat this (meaning reject U_2.)

8.8.2.2 Example 2: Standard Normal

To generate random variables from the standard normal distribution, we need a function that majorizes that distribution, whose pdf is redisplayed here for convenience:

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

We let $h(x) = e^{-x}$, an exponential distribution with $\lambda = 1$. Figure 8.8 shows how this exponential density majorizes the standard normal density. The constant $c = \sqrt{2e/\pi}$ will have the property that $c \cdot h(x) \geq f(x)$, or equivalently, that $1 \geq f(x)/(c \cdot h(x)$, because

$$\begin{aligned} \frac{f(x)}{c \cdot h(x)} &= \frac{\frac{2}{\sqrt{2\pi}} e^{-x^2/2}}{\sqrt{2e/\pi} e^{-x}} \\ &= \frac{2e^{x-x^2/2}}{\sqrt{4e}} \\ &= e^{x-x^2/2-1/2} \\ &= e^{-(x-1)^2/2} \\ &\leq 1 \end{aligned}$$

because in this last step, the exponent is negative and so the fraction is at most 1. The procedure accepts if a uniform random variable U is at most $e^{-(Z-1)^2/2}$. Thus the method is

1. Generate a random variable U_1 from $U(0,1)$.
2. Generate an exponential Z with mean 1 using the inverse CDF, $-\ln(U_2)$, by generating a second uniform random variable U_2 and setting $Z = -\ln(U_2)$.
3. If $U_1 \leq e^{-(Z-1)^2/2}$ then return either $X = Z$ or $X = -Z$ with equal probability; otherwise go back to step 1 and repeat this. To return either value with equal probabaility, we can generate a third random variable U_3 from $U(0,1)$ and return $X = Z$ if $U_3 < 0.5$ and $X = -Z$ otherwise.

8.8.2.3 Digging Deeper: Why This Works

It may seem a bit puzzling that this procedure, originally proposed by John von Neumann in 1947, will produce random variables from the given distribution. Roughly speaking, the procedure picks random points from the area under the graph of $c \cdot h_Z(x)$ and accepts a point if it lies under the graph of $f(x)$. This is depicted in Figure 8.8.

We now prove that the method works. Those who have not had a course in basic probability theory can skip this.

1. Recall that the two random variables, U, from $U(0,1)$, and Z, from h_Z, are independently distributed.
2. We start by proving that the parameter p of the geometric distribution defined above equals $1/c$. Because U is a $U(0,1)$ random variable, for all y, $Pr\{U \leq y\} = y$. Therefore, for all x

$$Pr\{U \leq \frac{f(Z)}{c \cdot h_Z(Z)} \mid Z = x\} = \frac{f(x)}{c \cdot h_Z(x)}$$

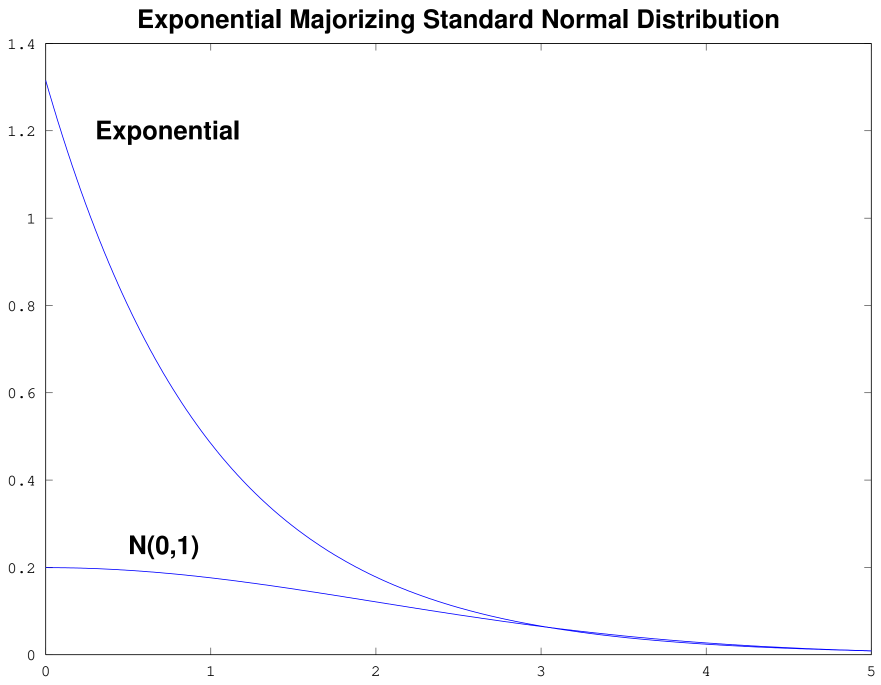

Figure 8.8: A graph showing the exponential pdf majorizing the standard normal pdf.

This ratio is just the fraction of the line vertical segment from $(x,0)$ to $(x, c\cdot h_Z(x))$ that lies under $f(x)$. From probability theory, we know that, if $g(u,z)$ is a probability density of two variables u and z, then the marginal distribution of u, $g(u) = \int_{-\infty}^{\infty} g(u,z)dz = \int_{-\infty}^{\infty} g(u|z)g_Z(z)dz$ where $g_Z(z)$ is the density for z. In this case $g(u,z)$ is the density of the probability distribution of U and Z in $Pr\{U \leq \frac{f(Z)}{c\cdot h_Z(Z)}\}$ and g_Z is h_Z, so we have

$$\begin{aligned}
p &= Pr\{U \leq \frac{f(Z)}{c\cdot h_Z(Z)}\} \\
&= \int_{-\infty}^{\infty} Pr\{U \leq \frac{f(Z)}{c\cdot h_Z(Z)} \mid Z = x\}\cdot h_Z(x)dx \\
p &= \int_{-\infty}^{\infty} \frac{f(x)}{c\cdot h_Z(x)}\cdot h_Z(x)dx \\
&= \frac{1}{c}\int_{-\infty}^{\infty} f(x)dx \\
&= \frac{1}{c}
\end{aligned}$$

because f is a pdf and so its integral over all real numbers is 1.

3. We use this result to prove that the probability density function of the variables returned by the method is $f_X(x)$, or equivalently, that its CDF is $F_X(x)$. Let us denote the CDF of the variables that are returned, $F_G(x)$.

4. Observe that the probability distribution of the X variables returned by the method is the probability distribution of the Z variables that are not rejected. In other words, the probability distribution of the X variables is the same as the conditional probability distribution of Z given that $U \leq f(Z)/(c\cdot h_Z(Z))$.

5. Therefore, we must show that $F_X(x) = Pr\{Z \leq x \mid U \leq f(Z)/(c\cdot h_Z(Z))$.

6. Because U is a $U(0,1)$ random variable, $Pr\{U \leq y\} = y$. Therefore $Pr\{U \leq f(Z)/(c \cdot h_Z(Z)) \mid Z = x\} = f(x)/(c \cdot h_Z(x))$.

7. Let B denote the event $U \leq f(Z)/(c \cdot h_Z(Z))$ and let A denote the event $Z \leq x$. Then $F_G(x) = Pr\{Z \leq x \mid U \leq f(Z)/(c \cdot h_Z(Z)) = Pr\{A|B\}$. Recall that

$$Pr\{A|B\} = Pr\{B|A\} \cdot Pr\{A\}/Pr\{B\}$$

8. We established in Step 2 above that $Pr\{B\} = Pr\{U \leq f(Z)/(c \cdot h_Z(Z))\} = 1/c$. Also $Pr\{A\} = Pr\{Z \leq x\} = H_Z(x)$ where H_Z is the CDF of h_Z. Therefore

$$\begin{aligned} F_G(x) &= Pr\{A|B\} \\ &= Pr\{B|A\} \cdot Pr\{A\}/Pr\{B\} \\ &= Pr\{U \leq \frac{f(Z)}{c \cdot h_Z(Z)} \mid Z \leq x\} \cdot H_Z(x)/(1/c) \end{aligned}$$

Since

$$\begin{aligned} Pr\{U &\leq \frac{f(Z)}{c \cdot h_Z(Z)} \mid Z \leq x\} = \frac{Pr\{U \leq \frac{f(Z)}{c \cdot h_Z(Z)} \ \&\ Z \leq x\}}{H_Z(x)} \\ &= \int_{-\infty}^{x} \frac{Pr\{U \leq \frac{f(Z)}{c \cdot h_Z(Z)} \mid Z = t \leq x\}}{H_Z(x)} h_Z(t)dt \\ &= \frac{1}{H_Z(x)} \int_{-\infty}^{x} \frac{f(t)}{c \cdot h_Z(t)} \cdot h_Z(t)dt \\ &= \frac{1}{c \cdot H_Z(x)} \int_{-\infty}^{x} f(t)dt \\ &= \frac{F_X(x)}{c \cdot H_Z(x)} \end{aligned}$$

it follows that

$$\begin{aligned} F_G(x) &= \frac{F_X(x)}{c \cdot H_Z(x)} \cdot H_Z(x)/(1/c) \\ &= F_X(x) \end{aligned}$$

which proves that the procedure generates random variables with the distribution CDF $F_X(x)$.

8.9 Applications of Monte Carlo Methods

There are many different types of applications of Monte Carlo methods. We discuss and elaborate on a few of the ones presented by Quinn [12].

8.9.1 Neutron Transport

When a neutron collides with an atom, it may get absorbed by the nucleus of the atom, forming a heavier atom. It may not be absorbed, but instead might be scattered by the atom. The neutron transport problem is the problem of modeling the interaction of neutrons and atoms. One method of modeling this interaction is to shoot neutrons towards a homogeneous metal plate of thickness T and infinite height and width (i.e., the height and width do not matter.) Some neutrons are reflected by the plate, others are absorbed by the plate, and still others pass through the plate completely. The thickness of the plate, the type of particle, and the material of the plate are three factors that influence the outcome of the bombardment of the plate by these particles. The fractions of neutrons that are reflected, absorbed, and transmitted through the plate are

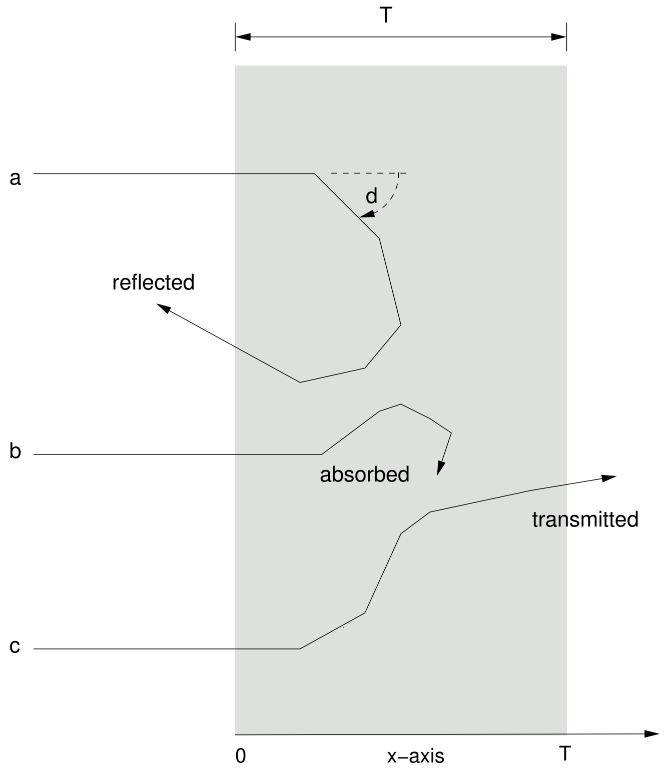

Figure 8.9: Particle interaction with metal plate of thickness T. Particles have an initial direction perpendicular to the plate and upon reaching it, they interact with its atoms. The result of this interaction is that a particle may eventually be reflected, absorbed, or transmitted through the plate.

recorded and form the experimental observations. Such an experiment can be simulated in software. Figure 8.9 illustrates the physical model.

The problem is parametrized by the following constants:

- C_c is called the ***cross-section of capture***, measured in a unit of measurement known as a ***barn*** ($10^{-28}m^2$). It can be thought of as being proportional to the probability that a neutron will be absorbed by an atom with which it collides.
- C_s is called the ***cross-section of scatter***. It can be thought of as indicating the probability that the neutron will be scattered away from the atom.
- $C = C_c + C_s$ is the ***total cross-section***.

The distance that a neutron travels in the plate before it interacts with an atom obeys an exponential distribution with parameter C (i.e., with mean C^{-1}.) Thus, the probability that the neutron travels a distance at most L is the CDF of an exponential and is thus $1 - e^{-CL}$. When a neutron reaches an atom, the probability that it is absorbed is C_c/C and the probability that it is scattered is C_s/C. If it is scattered, the direction of its new trajectory is random. The angle that the new direction makes with the horizontal axis we will denote by d. With this information, we can simulate the experiment as follows.

We choose a number of neutrons n to bombard the plate. For each neutron, we repeatedly simulate the motion of the neutron in the plate until one of three events happens:

- The neutron is absorbed by an atom. We increment an *absorption counter* in this event.
- The neutron is transmitted through the plate (its x-coordinate is greater than T.) We increment a *transmission counter* in this event.

- The neutron is reflected by the plate (its x-coordinate is less than 0.) We increment a *reflection counter* in this event.

Simulating the motion is one of several steps in which the Monte Carlo method is used. We generate a random exponential variable L with parameter C; this is the distance that it travels both initially and if it has interacted with an atom and not been absorbed. Its direction with respect to the x-axis is also a random variable, but in this case a uniformly distributed random variable d over the interval $[0, \pi)$. Therefore, the new x-coordinate is the old coordinate x plus $L \cdot \cos(d)$. By the choice of interval, $\sin(d)$ is always positive. This implies that the neutron bounces upwards. This does not matter in this simulation because the plate has infinite height and all we care about is its motion in the horizontal direction.

If the new position is outside of the plate it has been transmitted or reflected. Otherwise it interacts with a new atom. To simulate what happens when it interacts with an atom, a third randomization step takes place, as the probability that it is absorbed is modeled as a Bernoulli trial with success probability C_c/C. Therefore, a uniform random variable is generated, one from $U(0, 1)$, and if it is less than C_c/C then it is absorbed and otherwise it is scattered. The sequential algorithm for the simulation of a single neutron's interaction is shown in Listing 8.3. In the listing, whenever the variable `U` appears, it is meant as a short-hand notation for a randomly-generated $U(0, 1)$ value.

Listing 8.3: Simulation of a single neutron in a plate.

```
const double  Thickness;     // thickness of plate
const double  C_capture;     // cross section of capture
const double  C_scatter;     // cross section of scatter
const double  C_total =  C_capture + C_scatter;

double L;                   // length of trajectory to next location
double d;                   // angle of trajectory wrt x-axis
double x;                   // x-coordinate of neutron position at end of
    trajectory
int    num_reflections   = 0;
int    num_absorptions   = 0;
int    num_transmissions = 0;

d = 0;
x = 0;
still_bouncing = TRUE;

while ( still_bouncing ) {
    L = - (1/C_total) * ln (U);       /* U is U(0,1); L is exponential, param C */
    x = x + L * cos (d);                 /* new x-position */
    if ( x < 0 ) {                       /* reflected */
        num_reflections++;
        still_bouncing = FALSE;
    }
    else if ( x > Thickness ) {          /* transmitted */
        num_transmissions++;
        still_bouncing = FALSE;
    }
    else if ( U < (C_capture/C_total)) { /* absorbed */
        num_absorptions++;
        still_bouncing = FALSE;
    }
    else {
        d = U * PI;                      /* new angle */
    }
}
```

Notes

1. This problem is highly data parallel, because it is assumed that neutrons are independent and do not interfere with each others trajectories.
2. The simulation implicitly takes place over time, but because the problem does not require any output regarding how much time it takes before a particle is reflected, absorbed, or transmitted, there is no reason to keep track of actual time. Each time that the while loop is re-entered, time has elapsed implicitly, but not necessarily the same amount of time. In simulations of this nature in which the iterations are not constant time intervals, the time is refered to as ***Monte Carlo time***. If we needed to record the actual elapsed time, we would require velocity and other forces, because these will affect the time that the neutron travels.

8.9.2 Random Walks and the Dirichlet Problem

In introducing Foster's design methodology in Chapter 3, we applied it to a boundary value problem in which we had a one-dimensional insulated rod that had ice baths at its ends. This is actually a simple instance of a more general problem known as the ***Dirichlet problem***. The Dirichlet problem consists of finding a function u that is defined, continuous, and differentiable over a closed domain D with a boundary C and that satisfies ***Laplace's Equation*** on D and the boundary condition on C:

$$\begin{aligned} \nabla^2 u &= 0 \quad \text{on } D \\ u &= f \quad \text{on } C \end{aligned}$$

where f is some predefined function on the boundary and ∇^2 is the Laplacian operator. All of this will now be explained in simple terms. The reason we state it in its most general form is that it has a great many applications beyond the particular example we will study, and you should be aware of them. It has application to fluid flow, to heat transfer in general, to plasma simulation, and to the distribution of energy in general.

Imagine that an object has heat applied to it at various contact points, and then the heat is removed. Over time, the heat is transferred to other parts of the object. In the beginning, at each point in the object, the temperature at that point is in flux; it is changing as a function of time. Eventually, though, the temperature within the object becomes independent of time; it reaches a state where, for each point in the object, the temperature at that point is independent of time. When this has happened, the object has reached a ***steady state temperature distribution***. The temperature at any point of the object can be represented as a function of the point's coordinates and a time variable. Conventionally, the function symbol u is used to represent the temperature as a function of position and time. In three dimensions, it would be $u(x, y, z, t)$. In two dimensions, it would be $u(x, y, t)$. This function is generally continuous and differentiable on the interior of the object.

In the past, physical experiments would have to be conducted to understand the conduction of heat. Now computers can be used to do this. One can model how the heat is transferred when heat is applied to an object at specific points, by finding the form of this function u. This is, of course, a non-trivial problem. What is known from physics is that there are natural laws that govern how the heat is transferred, and that they reduce the problem to solving Laplace's Equation over the set of points within the object.

The problem gets very complicated for objects of complicated shape, but we can demonstrate all of the same principles with a two-dimensional version of it. Imagine a two-dimensional plate of ***isotropic***, ***homogeneous*** material, insulated along its entire top and bottom. The fact that the material is isotropic means that the properties of the material are uniform in all directions; that it is homogeneous means that it is the same material throughout. Assume the insulator is perfect and that no heat is lost or gained anywhere in the interior. Now let us assume further that along each of the four edges, the temperature is held fixed and not necessarily the same. For example, we could make the temperature 0° C along the north and south edges, and 100° C along the east and west edges. We can easily define a function $f(x, y)$ that satisfies these conditions. We assume that the temperature within the interior is unknown.

When the plate reaches a steady state temperature distribution, it means the temperature at every point is not changing with respect to time. Mathematically, this means that the derivative with respect to time is zero for all points:

$$\frac{d}{dt}u(x,y,t) = 0 \tag{8.7}$$

A consequence of the laws of physics, which can be proved, is that the rate of change of the temperature at any point is described by the equation

$$\frac{d}{dt}u(x,y,t) = k \cdot \left(\frac{\partial^2 u(x,y,t)}{\partial x^2} + \frac{\partial^2 u(x,y,t)}{\partial y^2}\right) \tag{8.8}$$

which states, in English, that the rate of change in the temperature over time is a constant times the sum of the second derivatives of the temperature with respect to the x and y directions. This has an averaging type of effect. Combining Eqs. 8.7 and 8.8, we conclude that

$$0 \;=\; \frac{\partial^2 u(x,y,t)}{\partial x^2} + \frac{\partial^2 u(x,y,t)}{\partial y^2} \tag{8.9}$$

This is known as ***Laplace's Equation***. The operator ∇^2 is a shorthand notation often used for the ***differential operator***

$$\left(\frac{\partial^2}{\partial x}, \frac{\partial^2}{\partial y}\right)$$

in any number of dimensions. (Think of it as a vector and set the inner product of it with the vector $(u(x,y,t), u(x,y,t))$ to zero and you get Eq. 8.9.) Eq. 8.9 is a partial differential equation that sometimes must be solved numerically, because an analytical solution may not exist. The numerical solution is often converted to a discrete problem because it is computationally faster. In the discrete version, instead of the plate's having infinitely many points, it has a two-dimensional grid of points. It can be shown that Eq. 8.9 is approximated using finite difference methods by the finite difference equation

$$0 = \frac{u(x+h,y,t) - 2u(x,y,t) + u(x-h,y,t)}{h^2} + \frac{u(x,y+k,t) - 2u(x,y,t) + u(x,y-k,t)}{k^2}$$

and if our grid points are $u_{i,j}$ and we drop the variable t this becomes

$$0 = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{k^2}$$

Since the grid is evenly spaced, we can let $h = k = 1$. Simplifying and solving for $u_{i,j}$, the Laplace equation becomes

$$u_{i,j} = \frac{u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}}{4}$$

Stated in a simpler way, if u_O represents the temperature at an arbitrary point in the interior of the grid, and u_N, u_E, u_S, and u_W represent the temperatures at its north, east, south, and west neighbors respectively, then the equation is

$$u_O = \frac{u_N + u_S + u_E + u_W}{4}$$

To solve the problem, we could repeatedly apply this equation to every grid point until the value converges at that point. Convergence occurs when the difference between successive values at the point is below some predefined threshold value. Since the temperature at each point is being replaced by the average of the neighboring points, the number of iterations required for convergence can be very large. As we must solve this equation for every point, we are looking at a problem that can be computationally expensive. For example, if the grid were $10^4 \times 10^4$, then we would need to iterate the numerical procedure at each of 10^8 points. For this reason, it is common to use a Monte Carlo method instead.

The mention of Monte Carlo in this context might lead you to think that we can just randomly pick points in the grid and solve for the function at those points, but this will not work. Instead we use a Monte Carlo

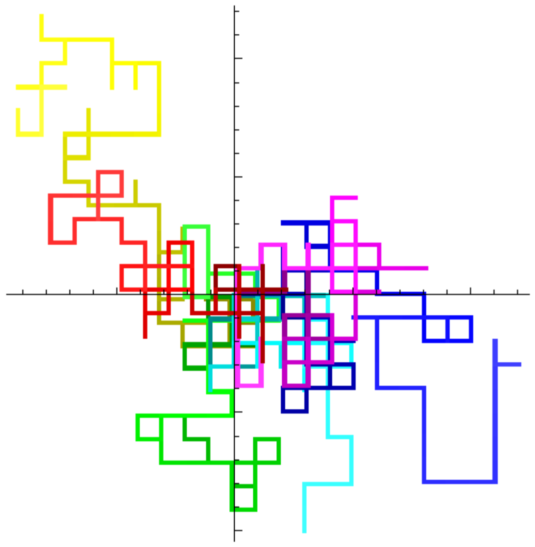

Figure 8.10: A sequence of random walks from the origin of a 2D grid.

method known as a ***random walk***. Informally, a random walk is a sequence of random steps in a ***state space*** that solves a problem by sampling that space in a specific way. Figure 8.12 shows the result of a random walk solution to this problem, color-coded by temperature. (Random walks are a type of ***Markov chain***, for those who know what this is.)

Example 6. Imagine a very drunk person walking in a city with a rectangular street grid, with city walls enclosing the entire grid. This person starts out at some position within the grid. The person is so drunk that his walk is completely random. He walks along the current street until he arrives at an intersection, which represents a decision point. There, he chooses the direction to take next randomly. This can even include going back down the street from which he just arrived. He takes that direction and walks until the next intersection, where a decision is made again. He walks and walks until he reaches a city wall, at which point he stops. This is a random walk. Figure 8.10 illustrates a sequence of random walks that such a drunk might make from the origin of a grid.

There are many questions that can be asked about such a walk, such as

- Is it guaranteed to reach a city wall?
- If so, what is the average length of the path to the wall?
- If he starts from position $[0, 0]$, how often will he return to $[0, 0]$?
- Are there intersections he will never visit?
- Are there intersections he will visit infinitely many times?

Let us now define a random walk more rigorously. Let S denote some ***state space***. You can think of it as the set of intersections on a two or three dimensional grid, or more generally, as a vector space over the integers. It can even be more general than this; it can be the set of states that a collection of interacting particles can assume. For now, it is easiest to visualize it as a grid of some kind. We will call the elements of this state space, ***points***. The space S is assumed to have some structure, such as a coordinate system, so that it makes sense to talk about the relative positions of two points in the space. Let $\{X_0, X_1, X_2, \ldots\}$ denote a sequence of random variables that take on values from S. In other words, these random variables can be points in the space. We think of the subscripts as discrete time steps, so that X_n is a random event at time n. In Example 6, each X_n is an intersection, or equivalently, a grid point $[i, j]$. Let us call this sequence of random variables, a ***chain***.

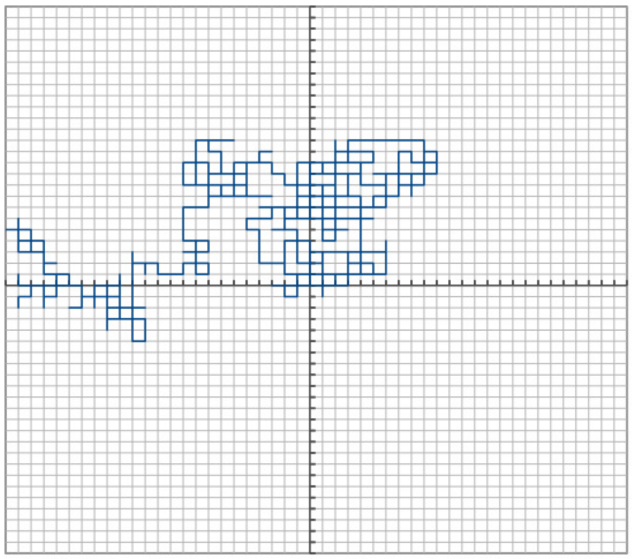
Figure 8.11: A random walk generated by Listing 8.4.

Definition 7. The ***transition probability*** of a chain $\{X_n, n \in \mathbb{N}\}$ is the function

$$p_{i,j}(m,n) = Pr\{X_n = j \mid X_m = i\}$$

which is the conditional probability that, if the chain is in state i at time m it will be in state j at time $n \geq m$.

Definition 8. A chain is ***time-homogeneous*** if the transition probability for that chain does not depend on time:

$$p_{i,j}(n,n+1) = p_{i,j}(m,m+1)$$

for all $n, m \in \mathbb{N}$. For a time-homogeneous chain, the transition probability is not a function of n and so is written $p_{i,j}$.

Definition 9. A chain is ***space-homogeneous*** if $p_{i,j}$ only depends upon the relative positions of i and j in space.

The concept of spatial homogeneity is that the transition from a point i to a point j depends only on the directed distance from i to j. If in our drunkard example, the drunk walked in a space homogeneous way, then if the drunk is at position $i = (3,4)$, the probability of going to position $j = (3,5)$ is the same as the probability that the drunk will go to position $j = (8,3)$ if he is currently at $i = (8,2)$, because in both cases, the vector difference $j - i = (0,1)$.

Now it is easy to define a random walk.

Definition 10. A ***random walk*** is a chain that is both time-homogeneous and space-homogeneous.

These two properties together imply that the next state that the walk visits is independent of wherever it has been before, and that the choice of direction is independent of the current position, pretty much like the drunkard from the example. Listing 8.4 is a small program that performs a single random walk on a two-dimensional grid of fixed size, and Figure 8.11 shows the the results of a single run, plotted on a grid.

Listing 8.4: randomwalk_2Dgrid.c

```
/* include directives omitted */

```

```
#define NORTH   1
#define EAST    2
#define SOUTH   3
#define WEST    4

typedef struct {   /* A 2D Point */
    int x;
    int y;
} point2d;

const point2d East  = { 1, 0};
const point2d West  = {-1, 0};
const point2d North = { 0, 1};
const point2d South = { 0,-1};

/* Randomly generate a new direction to walk */
point2d next_dir()
{
    double u = uniform_random();
    if         ( u < 0.25 )  return North;
    else if    ( u < 0.50 )  return East;
    else if    ( u < 0.75 )  return South;
    else    /* 0.75 <=u */  return West;
}

/* Generate next point from current point and direction */
point2d next_point(point2d oldpoint, point2d direction)
{
    point2d temp;
    temp.x = oldpoint.x + direction.x;
    temp.y = oldpoint.y + direction.y;
    return temp;
}

/* Test if given point is on a boundary, Returns non-zero if on a boundary, 0 if
    not. */
int  on_boundary(point2d point, int width, int height)
{
    if       ( 0 == point.x )            return WEST;
    else if ( width -1 == point.x )     return EAST;
    else if ( 0 == point.y )            return NORTH;
    else if ( height - 1 == point.y )  return SOUTH;
    else                                return 0;
}


int main ( int argc, char * argv[] )
{
    point2d current, next;
    int     width         = 50;   /* Width of grid           */
    int     height        = 50;   /* Height of grid          */
    char   *gaptable;             /* Gap table for random() */
    int     gaptable_size = 1000; /* Size of gap table       */

    /* Start walk from center of grid */
    current.x = width/2;
    current.y = height/2;
    printf("%d\t%d\n", current.x, current.y);

    /* Generate a larger gap table for random() function */
    gaptable = init_random(gaptable_size);

    /* Iterate until the walk reaches a boundary */
    while ( 1 ) {
        next = next_point(current, next_dir());
        printf("%d\t%d\n", next.x, next.y);
        if ( on_boundary(next, width, height) )
            break;
        else
            current = next;
    }
    printf("%d\t%d\n", next.x, next.y);
    free(gaptable); /* Release memory allocated to gap table */
    return 0;
}
```

Now that we know what a random walk is, how can we use it to solve our simplified example of the Dirichlet problem? The idea is that for each grid position, we will do a random walk from that position until we reach a boundary point. When we reach a boundary, we take the value of that boundary point and add it to an accumulator, initialized to zero. At each grid point we will do n random walks and use the average of the values stored in the accumulator as the value of that grid point. Our random walk will randomly choose, with equal probability, one of the four directions, north, east, south, and west, as the direction to move to the next point, until a boundary point is reached.

Some treatments of random walks define a random walk as a sum of independently and identically distributed random variables:

$$S_n = X_0 + X_1 + X_2 + \cdots + X_n$$

with appropriate conditions on these variables. We can model our example using this formalism by defining two unit vectors, $e_1 = (1,0)$ and $e_2 = (0,1)$, and then observing that the next point is always the current point plus or minus one of these two vectors, i.e., one of $X_i \pm e_1$or $X_i \pm e_2$, with the choice of $\pm e_j$, $j = 1,2$ being randomly generated. If we do this, then the current point can be viewed as the sum of the initial location as a vector plus the randomly generated unit vectors.

It is a remarkable fact that the probability of reaching the boundary is 1 when starting at any interior point. Another way to say this is that the boundary is reached in finite time, no matter where the walk starts. Let $f_k(X)$ denote the edge temperature added to the accumulator for point X in the k^{th} random walk from point X. It is also a remarkable fact that the average so obtained

$$\frac{1}{n}\sum_{k=1}^{n} f_k(X)$$

is an unbiased estimate of the actual temperature at that point.

The last question is how to decide how many iterations are necessary to be certain that the heat distribution has approached an approximate steady state. Since steady state implies that the temperatures of the neighboring points no longer differ enough to cause a change to the temperature at the current point, one solution is to define a threshold value $\varepsilon > 0$ and check after each iteration whether all points have changed value by less than this amount. Therefore, in each iteration, at each point, the absolute difference between the current average and the new average at that point is compared to the maximum of such differences among all points in that iteration. When the maximum difference across all points in the grid in a given iteration is less than ε, the computation terminates.

Listing 8.5 contains the significant parts of a program to carry out the algorithm we just described. Various initializations and the function definitions have been omitted to save space. The definition of the `point2d` structure, the functions `on_boundary()` and `next_point()` are the same as those from Listing 8.4.

Listing 8.5: Random walk algorithm for 2D Heat Distribution.

```

```

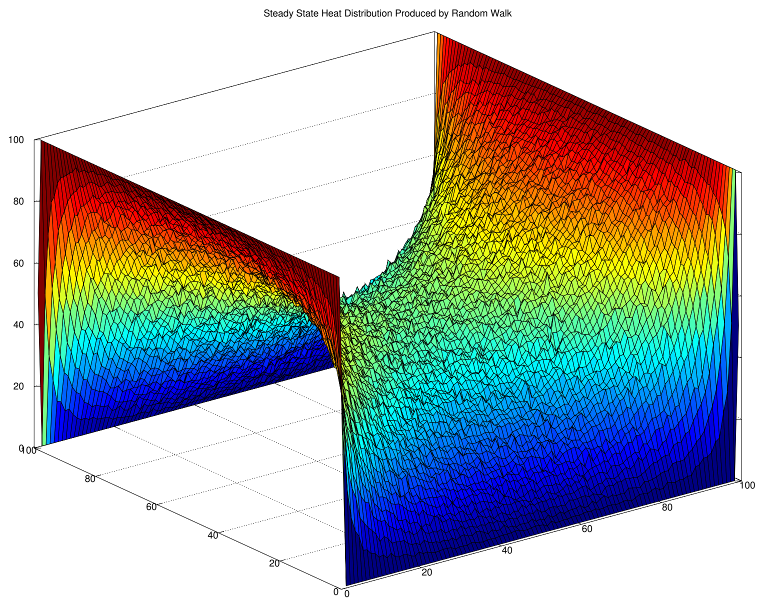

Figure 8.12: Visualization of the result of a random walk solution to the heat diffusion problem in a 2D plate in which the temperature is fixed at 100° C along the east and west edges and 0° C along north and south edges.

```
#define CONVERGENCE_THRESHOLD  0.05

/* on_boundary() returns 0 if point is not on boundary, or if it is, it returns
    the integer representing the boundary. */
int      on_boundary(point2d point, int width, int height);

/* next_point() returns the next point, using a uniformly distributed random
    direction added to the old point. */
point2d next_point (point2d oldpoint, point2d direction);

int main ( int argc, char * argv[] )
{
    point2d current;
    int      i, j;
    int      width, height;
    double   oldvalue, diff;
    double   maxdiff;
    double   tolerance;
    int      location;
    double   boundary_temp[4];
    double   **plate;
    double   *plate_storage;

    /* 1. Initialize temperatures at the four corners as the average of the
        temperatures of the adjacent edges. */
    plate[0][0]                = (boundary_temp[0] + boundary_temp[3])/2;
    plate[0][width-1]          = (boundary_temp[0] + boundary_temp[1])/2;
    plate[height-1][0]         = (boundary_temp[3] + boundary_temp[2])/2;
    plate[height-1][width-1] = (boundary_temp[2] + boundary_temp[1])/2;

    /* 2. Initialize the temperatures along the edges of the plate. */
    for ( j = 1; j < width -1; j++ ) {
        plate[0][j]        = boundary_temp[0];
        plate[height-1][j] = boundary_temp[2];
    }
    for ( i = 1; i < height -1; i++ ) {
        plate[i][0]        = boundary_temp[3];
        plate[i][width-1]  = boundary_temp[1];
    }

    /* 3. Initialize the interior temperatures to 0. */
    for ( i = 1; i < height -1 ; i++ )
        for ( j = 1; j < width -1 ; j++ )
            plate[i][j] = 0.0;

    /* 4. Set the acceptable tolerance to a small value and start an iteration
       counter. */
    tolerance = CONVERGENCE_THRESHOLD;
    count = 0;
    while ( 1 ) {
        /* Reset maximum difference to 0 at the start of each iteration */
        maxdiff = 0;
        /* For each grid point [i,j] */
        for ( i = 1; i < height -1 ; i++ ) {
            for ( j = 1; j < width -1 ; j++ ) {
                /* Do a random walk until a boundary is reached. */
                current.x = j;
                current.y = i;
                while ( 0 == (location = on_boundary(current, width, height)) ) {
                    current = next_point(current, next_dir());
                }
                /* Get difference between old average at this point and average
                   with the new boundary point factored into it. */
                oldvalue = plate[i][j];
                plate[i][j] = (oldvalue*count + boundary_temp[location-1]) / (
                   count + 1);
                diff = fabs(plate[i][j] - oldvalue);

                /* If the difference at this point is the largest so far in this
                   iteration, update maxdiff. */
                if ( diff > maxdiff )
                    maxdiff = diff;
            }
        }
        if ( maxdiff < tolerance )
            /* The algorithm has converged sufficiently, so break the loop. */
            break;
        else
            count++;
    }
}
```

8.9.3 The Metropolis Algorithm and the Two-Dimensional Ising Model

In this section we introduce a fundamental and powerful Monte Carlo algorithm called the ***Metropolis algorithm.*** The Metropolis algorithm is at the intersection of several key ideas, in that

- it is a type of random walk;
- it is a type of acceptance-rejection sampling that does not require a blanketing function;
- it contains the key element of an important class of algorithms known as simulated annealing algorithms.

In the preceding problem, we saw an example of a random walk in which the states were locations in a grid, but as we mentioned when we defined a random walk in Section 8.9.2, a state can be something more general than this. We use the ***Ising model*** to illustrate this concept. The Ising model is a mathematical model of ferromagnetism used in statistical mechanics to study the behavior of materials such as magnets. Materials such as iron stay magnetized for a long time, but it was not understood why or for how long. The model was invented to try to explain this phenomenon [4]. Subsequently it was applied to other problems, such as the motion of atoms in a gas and the activity of neurons in the brain.

For simplicity, we explore the two-dimensional Ising model, which was the one that Ising originally solved. A square plate is treated as having a finite Cartesian grid or lattice, whose intersection points are called ***sites***. Each site σ_k can be in one of two ***spin*** states, ***up*** or ***down***. Usually the values $+1$ and -1 are associated with the spins up and down respectively [8]. The state of a site represents the direction of an atom's spin coordinate, i.e., the direction of its nuclear magnetic dipole moment[9]. The term *spin* does not imply that anything is actually spinning; it is a concept from quantum mechanics that has no correspondence in classical physics. Because atoms have electromagnetic fields, adjacent atoms interact with each other and can affect each others spins and energy levels.

Imagine a square lattice consisting of 2500 sites, $\sigma_k,\ k = 0, 1, \ldots, 2499$, arranged in a 50 by 50 grid. Each site can have either up or down spin. A ***spin configuration***, or ***configuration*** for short, is a specific assignment σ of spin values to each site in the lattice. In this example, there are 2^{2500} possible configurations, corresponding to all possible assignments of the two values $+1$ or -1 to each site. (It is the number of different bit-strings of length 2500.) Different configurations represent different thermodynamic states of the material, i.e., its temperature, electromagnetic properties, available energy, and so on. The problem that the Ising model tries to solve is to find the configuration associated with thermodynamic equilibrium.

We let Λ denote the set of all sites. The energy of a given configuration σ is given by the function

$$E(\sigma) = -\sum_{i,j\in\Lambda} J\sigma_i\sigma_j - B\sum_{i\in\Lambda}\sigma_i \tag{8.10}$$

where:

- the first sum is restricted to only those sites σ_i and σ_j that interact with each other, which is usually pairs of sites that are adjacent to each other,
- J is a constant that is the strength of the spin-spin interaction, and
- B is a constant indicating the strength of an external magnetic field.

The energy of a configuration is related to the probability that the material can be in that configuration. Configurations that have very high energy, for example, are very unlikely because they are unstable, whereas configurations that are very low energy are much more likely. The probability of any given configuration is given by the following probability density function:

$$\mu(\sigma) = \frac{e^{-E(\sigma)/\kappa T}}{Z(T)} \tag{8.11}$$

where κ is the Boltzmann constant, T is the temperature, and

$$Z(T) = \sum_{\sigma\in\Lambda} e^{-E(\sigma)/\kappa T}$$

[8]Quinn uses $+1/2$ and $-1/2$ instead in his book [12].
[9]The nuclear dipole moment is a vector associated with the direction of the spin of the nucleus of the atom.

is a normalization factor. By dividing by $Z(T)$ in Eq. 8.11, we turn $\mu(\sigma)$ into a pdf:

$$\sum_{\sigma\in\Lambda}\mu(\sigma)=\sum_{\sigma\in\Lambda}\frac{e^{-E(\sigma)/\kappa T}}{Z(T)}=\frac{\sum_{\sigma\in\Lambda}e^{-E(\sigma)/\kappa T}}{\sum_{\sigma\in\Lambda}e^{-E(\sigma)/\kappa T}}=1$$

Notice that large values of $E(\sigma)$ in Eq. 8.11 give rise to small values of $\mu(\sigma)$ because the exponent is negated. To simplify the notation, when the meaning is clear we will assume that all summations are over all sites in Λ.

Usually, we are trying to obtain an estimate of some physical quantity of the material, such as the mean magnetization or the mean energy per site. The physical property of interest can be represented by a function $f(\sigma)$. The problem then reduces to finding the expected value of $f(\sigma)$ with respect to the probability distribution $\mu(\sigma)$. Stated mathematically, we are given a function $f(\sigma)$ and we need to find the value of the expression

$$\mathcal{F}=\sum_{\sigma}f(\sigma)\mu(\sigma)=\frac{1}{Z(T)}\sum_{\sigma}f(\sigma)e^{-E(\sigma)/\kappa T}$$

which is, by definition, the expected value of $f(\sigma)$ with respect to the pdf $\mu(\sigma)$. Typically the function $f(\sigma)$ is very difficult to compute, and the energy $E(\sigma)$ is even harder to compute. Clearly it is impossible to find the energy of every possible configuration, even for a small number of sites. Instead some type of random sampling must be done. There are two problems with random sampling:

1. We do not want to pick configurations uniformly, because some configurations have very small probabilities, contributing very little to the expected value of the function $f(\sigma)$, whereas others have large probabilities, and they contribute much more. We would not get a statistically meaningful answer using uniform random sampling.
2. Sampling from the distribution μ is extremely difficult, because it requires computing $E(\sigma)$ using Eq. 8.10, which must iterate over all pairs of sites, just to get the sample probability, and while this may be feasible for small two-dimensional lattices, it is not for large lattices and for higher-dimensional problems. Some problems can have dimensions on the order of 100 or more.

In 1947, Nicholas Metropolis, Stanislaw Ulam, John von Neumann, and other colleagues had a similar problem to solve while working at Los Alamos National Laboratories. They were trying to solve the problem of predicting the behavior of neutrons as they interacted with other atoms[10]. Von Neumann proposed using a statistical approach to solve this problem, laying the foundation of Monte Carlo computation as well as the acceptance-rejection method [3, 11, 1]. Their success with this approach led to further applications of the Monte Carlo technique. In 1950, Marshall Rosenbluth developed a theoretical scattering law, which was the key to the development and publication in 1953 by Metropolis and others, of an algorithm that could be used for efficient calculation of the properties of any large system [10]. They showed, in particular, how the algorithm could be used to calculate the expected value $\mathcal{F}$ of some property of a collection of N particles in equilibrium at a given temperature. We first summarize the algorithm as they first described it and then show in detail how it is applied to the Ising model.

In short, in each step of their algorithm, a single particle is displaced by a random amount and the change in the total energy ΔE of the system of particles is calculated. If the change is to a lower energy state ($\Delta E < 0$), the new position of the particle is accepted. If the change is to a higher energy state ($\Delta E > 0$), then a probabilistic approach is used to determine whether or not to accept the change. If the change is accepted, the particle is allowed to remain in the new position and the system is put into a higher energy state. If the change is not accepted, then the particle is returned to its old position. In either case, for the purpose of calculating the expected value of the function $\mathcal{F}$, the system is considered to be in a new state. I.e., the algorithm computes

$$\hat{\mathcal{F}}=\frac{1}{M}\sum_{j=1}^{M}\mathcal{F}_j$$

[10]To be clear, they were trying to predict how much energy would be released in the chain reaction that takes place when neutrons are released during atomic fission.

where F_j is the energy of the system after the j^{th}particle move is made. What makes this algorithm so brilliant is the choice of probabilistic decision, the idea of making a small perturbation so that evaluating the function F and the energy would be easy, and the fact that the system is allowed to jump to seemingly worse states on occasion. The key features of the algorithm are that

- It uses the acceptance-rejection method to generate random variables that are governed by the probability density distribution μ.
- It performs a random walk through a sequence of configurations that is computationally feasible because the next state is a small perturbation of the current state.
- The random walk converges to the limit distribution.
- The method of choosing the next state of the random walk samples the low-energy states with much higher probability than the high-energy states.

Notice that in this random walk, the successive states are not individual spatial locations but are vectors representing the locations of all particles in a system. They are highly multi-dimensional.

We now explain the Metropolis algorithm in terms of the Ising model. Let σ be the current configuration. Assume that $E(\sigma)$ has already been computed. We need to describe how the algorithm selects the next configuration; call it ξ.

1. The algorithm randomly picks a site whose spin it will change. (This site is selected using acceptance/rejection as well.) Call the state with this site's spin changed, ξ.
2. The change in energy, $\Delta E = E(\xi) - E(\sigma)$, is computed.
3. If $\Delta E < 0$ then the random walk advances to state ξ. (It has lower energy than the current state.)
4. Otherwise ($\Delta E > 0$), compute the transition probability
$$p_{\sigma,\xi} = \frac{\mu(\xi)}{\mu(\sigma)} = \frac{e^{-E(\xi)/\kappa T}}{e^{-E(\sigma)/\kappa T}} = e^{(-E(\xi)+E(\sigma))/\kappa T} = e^{-\Delta E/\kappa T}$$
5. The random walk chooses the new state ξ with probability $p_{\sigma,\xi}$ otherwise it remains in the same state σ.

The acceptance-rejection method is used in this last step:

1. A random variable U is generated from $U(0,1)$.
2. If $U < p_{\sigma,\xi}$ then the random walk advances to ξ, otherwise it remains in σ.

A pseudocode description of the application of the Metropolis algorithm to the Ising model appears in Listing 8.6.

Listing 8.6: Metropolis algorithm applied to Ising model.

```
/* Constants */
const double kB;      /* Boltzmann constant


/* Assume we want to compute the mean value of some function f.
   Input parameters:
        double temperature;            /* temperature of material           */
        int    number_of_sites;        /* number of sites in configuration */
        int    sigma[num_of_sites];    /* initial configuration            */
        int    max_steps;              /* how many times to iterate        */
*/
```

```

    double energy;
    int    x_current[num_of_sites];    /* current configuration            */
    int    x_next   [num_of_sites];    /* next configuration               */
    double sum_f;                      /* for accumulating mean value      */

    Let x_current  = sigma;  /* the initial configuration */
    /* Usually we want to run the algorithm a number of steps, throwing away the
        first few steps; we skip that here. */
    sum_f = 0;
    i = 0;
    repeat until i == max_steps {
        u = random number in [0,num_of_sites);    /* pick a random site */
        x_next = flip(u, x_current);              /* let x_next be a configuration
             differing from x_current by flipping the spin at site u */
        energy_change = E(x_next) - E(x_current); /* would not actually compute
            both of these energies; just the difference obtained from the change in
             spin at the site, which is much easier to compute */
        if ( energy_change < 0 ) {
            sum_f  += f(x_next);
        }
        else {
            /* acceptance/rejection step: generate a U(0,1) random number */
            u = random number in [0,1);

            /* compute value of transition probability */
            p = exp(-energy_change/(kB*temperature));

            /* do the statistical test: */
            if ( u < p )
                /* if u < p accept the change in state */
                x_current = x_next;

            /* Now either x_current is the new state or the old one. In either
                case, add value of f(x_current) to running sum */
            sum_f  += f(x_current);
        }
        i++;  /* update counter */
    }
    return sum_f/max_steps;
}
```

8.9.3.1 Does This Algorithm Work?

We would like to know whether this algorithm works, but this is a rather vague statement. One question that must be answered is

- Is the value computed by this function a good approximation to the true mean of the function of interest?

Mathematically, is

$$\hat{F} = \frac{1}{M}\sum_{j=1}^{M} F_j$$

a good approximation to

$$F = \sum_{\sigma} f(\sigma)\mu(\sigma) = \frac{1}{Z(T)} \sum_{\sigma} f(\sigma) e^{-E(\sigma)/\kappa T}$$

We will not prove it here, but the answer to this question is that $\hat{F}$ converges to F as $M \to \infty$ [10].

The second question is

- Does the algorithm actually reach a unique stationary distribution?

to which the answer is also yes. The random walk described by the Metropolis algorithm satisfies the conditions required for a Markov process to have a unique stationary distribution. This means that the limiting distribution is the underlying distribution. A sufficient condition for this statement to be true is that it satisfies what is called the ***detailed balance condition***, which basically states that the transitions are symmetric. To be precise, let $P(x_i)$ denote the probability that the walk is in state x_i and let $P(x_j|x_i)$ denote the transition probability that the walk will move into state x_j given that it is currently in x_i. The detailed balance condition is satisfied if for all states x_i and x_j

$$P(x_i|x_j)P(x_j) = P(x_j|x_i)P(x_i) \tag{8.12}$$

which states that it is just as likely to go from x_i to x_j as it is to go from x_j to x_i. It is not hard to show that the algorithm satisfies this condition. There are two cases to consider.

The first case is that $E(x_i) > E(x_j)$. In this case x_j is a lower energy state and the transition probability $P(x_j|x_i) = 1$, so the right-hand side of Eq. 8.12 is $P(x_i)$. The left-hand side is

$$\begin{aligned} P(x_i|x_j)P(x_j) &= e^{(-E(x_i)+E(x_j))/\kappa T} \cdot \frac{e^{-E(x_j)/\kappa T}}{Z(T)} \\ &= \frac{e^{-E(x_i)/\kappa T}}{Z(T)} \\ &= P(x_i) \end{aligned}$$

showing that it is true in this case. The second case is when $E(x_i) < E(x_j)$. In this case x_i is a lower energy state and the transition probability $P(x_i|x_j) = 1$, so the left-hand side of Eq. 8.12 is $P(x_j)$. The right-hand side is

$$\begin{aligned} P(x_j|x_i)P(x_i) &= e^{(-E(x_j)+E(x_i))/\kappa T} \cdot \frac{e^{-E(x_i)/\kappa T}}{Z(T)} \\ &= \frac{e^{-E(x_j)/\kappa T}}{Z(T)} \\ &= P(x_j) \end{aligned}$$

showing that it is also true in this case.

8.9.4 Simulated Annealing and the Room Assignment Problem

Optimization problems are an extremely important class of computational problems. An optimization problem is one in which the goal is to find the best solution to a problem from among the set of all feasible solutions. Stated mathematically, the goal is to minimize some real-valued function, usually of one of many variables, subject to a set of inequality and/or equality constraints. The function to be minimized is called the ***objective function***. When the set of variables is discrete, it is called ***combinatorial optimization***.

One of the most famous combinatorial optimization problems is the ***traveling salesman problem***, in which one is given a number of cities and asked to find a shortest path that a salesman can take that visits each city exactly once, returning to the original city. The objective function is the length of the path, and the set of feasible solutions is the set of all paths that visit each city exactly once and return to the original city. Another, less famous, but important problem in the design of computer processors is to decide how to partition the set of circuits onto discrete chips to maximize performance of the combined set of chips. One wants to minimize the number and total length of wires that have to cross from one chip to another, subject to constraints such as that the circuits fit into the chip packaging.

The room assignment problem is an instance of a more general combinatorial optimization problem known as the ***assignment problem***. We can state it as follows. We are given a set of $n/2$ rooms, each of which has a two-person occupancy, and a set of n college freshmen. Each student has expressed preferences regarding roommates, which are codified into an $n \times n$ symmetric ***incompatibility matrix*** C such that $C_{i,j}$ is a real number expressing the extent to which students i and j will be *incompatible* as roommates. The objective is to assign students to rooms so that every student is assigned to a room and all rooms have two students, while minimizing the total incompatibility. A solution is therefore a function $a : [0, n-1] \to [0, n/2-1]$ that maps each student into one of the $n/2$ rooms. We can represent a solution as an array. Let

$$f_C(a,i,j) = \begin{cases} C_{i,j} & \text{if } a(i) = a(j) \\ 0 & \text{otherwise} \end{cases}$$

We parametrize f by C to indicate that its value depends upon the incompatibility matrix C. The objective function $F_C(a)$ is

$$F_C(a) = \frac{1}{2} \sum_{i=0}^{n-1} \sum_{j \neq i} f_C(a,i,j)$$

The sum is divided by two because the matrix is symmetric and $F_C(a)$ counts each pair (i,j) twice. It really does not matter and we could ignore this, as we do not really care what the value of the objective function is as long as it has been minimized. Notice that the objective function decreases in value when paired students have low incompatibility.

In this problem, the objective function to be minimized is a function of a number of variables that is proportional to the number of students. If we change just a few student room assignments, its value can change, and there might be thousands of students. Therefore, it is not so simple to find an assignment that minimizes F_C. One naive approach might be to try a random walk, starting with one assignment and trying random changes to it, moving to a new solution if the change reduces the value of F_C. Unfortunately, this will not work well because it may get stuck in a local minimum rather than a global minimum.

To illustrate this, consider the function whose graph is shown in Figure 8.13. It has a single global minimum and several local minima. An algorithm that accepts a move to a new state if the move decreases the function's value can easily descend into any of these local minima, unless it just "got lucky" and happened to do this in the one place that leads to the global minimum. The algorithm should have a way of avoiding this problem.

In 1983, Kirkpatrick, Gelatt, and Vecchi had the insight to see how the Metropolis algorithm could serve as a tool for finding solutions to problems of this nature [6]. They saw a connection between the annealing process used in statistical mechanics and combinatorial optimization. When physicists need to produce a material with a uniform, defect-free crystalline structure, they heat the material until it melts, and then very slowly allow the material to cool. This process is called ***annealing***. When the material is heated, the atoms in it are in a higher energy state and are therefore more active, allowing them to rearrange themselves. As the temperature cools, the material tends to move into an equilibrium state in which its atoms take positions that minimize its energy. Kirkpatrick *et al* saw a way to simulate this annealing process in the iterative solution to combinatorial optimization. The objective function to be minimized is analogous to the energy of the material, and the current solution corresponds to a configuration of the atoms. They needed a way to add temperature to the algorithm, which is where the Metropolis algorithm comes into play.

In the Metropolis algorithm, if the new state has lower energy than the previous state, it is accepted, and if the new state has higher energy, then it is accepted with probability $e^{-\Delta E/\kappa T}$, where k is Boltzmann's constant and T is temperature. The energy difference ΔE can be replaced by the difference in the objective function from one state to the other, ΔF. Then we use the same procedure as in the Metropolis algorithm: we can accept the new state if ΔF is negative, and if not, then we accept with probability $e^{-\Delta F/T}$. But here is the twist that Kirkpatrick *et al* introduced: instead of treating the temperature as a constant, they turned it into a variable and allowed it to change during the course of the algorithm.

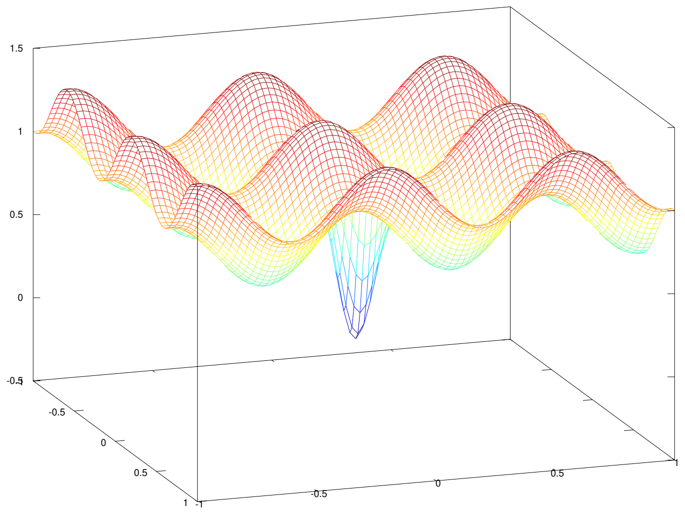

Figure 8.13: A function of two variables with many local minima and one global minimum.

As the value of the temperature changes, the probability changes as well. In particular, when $x > 0$, e^{-x} approaches 0 as $x \to \infty$. Since $\Delta F > 0$, it follows that $e^{-\Delta F/T}$ must approach zero as $T \to 0$, because T is in the denominator, so $\Delta F/T \to \infty$. Stated another way, as the temperature T is diminished, the probability of accepting a new state with a larger value of the objective function becomes smaller. This allows us to simulate the annealing process by starting the temperature at a high value and gradually diminishing it. In the beginning of the algorithm, there will be more chances for it to jump out of local minima as it starts to climb up the local hills, and as the algorithm progresses, there will be less of a tendency to climb up the hills, instead trying to descend further into the minima. This increases the probability that it will find the global minimum.

The annealing schedule used in the original article set the initial value of T, denoted T_0, to 10 and then "cooled" the system exponentially with the formula

$$T_n = 0.9^n T_0$$

which is equivalent to the recurrence

$$\begin{aligned} T_0 &= 10 \\ T_{n+1} &= 0.9 T_n \end{aligned}$$

They did not change the temperature with each new change of state. Instead, they used the following rules:

- Stay in the current temperature until there are 10 accepted changes (meaning the objective function was reduced 10 times), not necessarily consecutive, or until the number of attempts exceeds 100 times the input size, whichever comes first.
- If 10 accepted changes do not take place at three consecutive temperatures, terminate the algorithm, because it has become "frozen".

We can now describe the application of simulating annealing to the room assignment problem. The pseudocode description is given in Listing 8.7.

Listing 8.7: Pseudocode solution to room assignment problem using simulated annealing.

```

1. Initialize current_solution to be a random guess at a solution  to the problem;
2. Set initial temperature   = 10;
3. Set failure_count = 0;
4. repeat forever {
       set acceptance_counter = 0;
       set number_of_attempts = 0;
       repeat {
           Randomly pick two students in different rooms and swap them.
             Call this the new_solution;
           Compute the value of the objective function for the new_solution minus
             the value of the objective function for the current_solution;
           set u = random number in [0,1);
           set p = exp(-objective_function_change/temperature);

           if ( objective function has been reduced or u < p ) {
               increment acceptance_counter;
               set current solution to the new_solution;
           }
           else {
               increment number_of_attempts;
           }
       }
       until ( acceptance_counter == 10 ) or ( number_of_attempts >= 100*n )
       if ( acceptance_counter == 10 ) {
           set failure_count = 0;
       }
       else {
           increment failure_count;
           if ( failure_count == 3 ) {
                break;
           }
       }
       temperature = 0.9* temperature;
       if ( temperature < 0.001 ) {
           break;
       }
   }
5. Return the current_solution and value of objective function;
```

This algorithm will terminate if either the system becomes frozen or because the temperature became sufficiently close to zero. If the temperature becomes very close to zero, the algorithm will reject almost all changes of state that increase the value of the objective function, so it will not find a solution very different from what it currently proposes, and the number of consecutive rejections will cause it to fail. In principle when the latter happens the system will fail eventually, so this is just a way to stop wasting time.

8.10 Summary

This chapter introduced Monte Carlo methods, a very powerful set of approximation techniques for solving computationally intractable problems by statistical sampling. Underlying all Monte Carlo methods is the need to produce reliable streams of random numbers, and so this chapter also introduced a bit of the theory of random number generation, both sequential and parallel. The theory underlying Monte Carlo methods was explained, and various types of Monte Carlo methods were illustrated. All of the examples studied here are forms of Monte Carlo integration. Monte Carlo methods can also be used for simulations. One example in Section 8.8.1.1 showed how this could be done for exponentially-distributed arrivals.

References

[1] Isabel M. Beichl and F. F Sullivan. The metropolis algorithm. *IEEE Computing in Science and Engineering*, 2(1):65–69, 2000.

[2] Paul D. Coddington. Random number generators for parallel computers. Technical Report 13, Northeast Parallel Architecture Center, Syracuse University, 1997.

[3] Roger Eckhardt. Stan ulam, john von neumann, and the monte carlo method. *Los Alamos Science Special Issue*, pages 131–143, 1987.

[4] Ernst Ising. Beitrag zur theorie des ferromagnetismus. *Zeitschrift fÃŒr Physik*, 31(1):253–258, 1925.

[5] M. G. Kendall and B. Babington Smith. Randomness and random sampling numbers. *Journal of the Royal Statistical Society*, 101:147–166, 1938.

[6] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[7] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[8] D. H. Lehmer. Mathematical methods in large-scale computing units. In *Proceedings of a Second Symposium on Large-Scale Digital Calculating Machinery*, pages 141–146, 1949.

[9] Nicholas Metropolis. The beginning of the monte carlo method. *Los Alamos Science Special Issue*, pages 125–130, 1987.

[10] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[11] Nicholas Metropolis and Stanislaw Ulam. The monte carlo method. *Journal of the American Statistical Association*, 44:335–341, 1949.

[12] M.J. Quinn. *Parallel Programming in C with MPI and OpenMP.* McGraw-Hill Higher Education. McGraw-Hill Higher Education, 2004.

Subject Index

absolute error, 3
acceptance-rejection method, 21
annealing, 41
assignment problem, 41

Bernoulli trial, 20
blanketing function, 21
Box-Muller transformation, 21

CDF, 16
central limit theorem, 4
chain, 30
combinatorial optimization, 40
cross-section of capture, 26
cumulative distribution function, 16

detailed balance condition, 40
Dirichlet problem, 28
discrete uniform distribution, 5

exponential distribution, 17

Gaussian distribution, 21
generalized function inverse, 16
geometric distribution, 20

homogeneous, 28

iid, 4
increment, 9
independent sequencing, 13
independently and identically distributed, 4
inversion method, 17
Ising model, 36
isotropic, 28

lag, 11
lag table, 11
lagged Fibonacci generator, 11
Laplace equation, 29
Laplace's Equation, 28
leapfrogging, 12
linear congruential generator, 9

majorize, 21
Mean Value Theorem, 5
Metropolis algorithm., 35
modulus, 9
multiplicative congruential method, 10
multiplier, 9

normal distribution, 21

objective function, 40
optimization problem, 40
order, 10

primitive element, 10
PRNG, 8
pseudo-random number generator, 8

random walk, 30, 31

seed, 9
sequence splitting, 12
sites, 36
space-homogeneous, 31
spin, 36
spin configuration, 36
state space, 30
steady state temperature distribution, 28

time-homogeneous, 31
total cross-section, 26
transition probability, 31
traveling salesman problem, 40

uniformly distributed, 5

Weak Law of Large Numbers, 4