her departure, provided there is one available. Suppose that the probability that it is raining at the time of any departure is p. Let X_n denote the number of umbrellas available at the place where Ella arrives after walk number n; $n = 1, 2, \ldots$, including the one that she possibly brings with her. Calculate the limiting probability that it rains and no umbrella is available.

1.31 A mouse is let loose in the maze of Figure 1.10. From each compartment the mouse chooses one of the adjacent compartments with equal probability, independent of the past. The mouse spends an exponentially distributed amount of time in each compartment. The mean time spent in each of the compartments 1, 3, and 4 is two seconds; the mean time spent in compartments 2, 5, and 6 is four seconds. Let $\{X_t, t \geqslant 0\}$ be the Markov jump process that describes the position of the mouse for times $t \geqslant 0$. Assume that the mouse starts in compartment 1 at time $t = 0$.

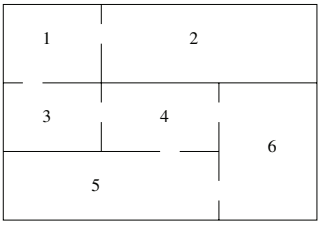

Figure 1.10: A maze.

What are the probabilities that the mouse will be found in each of the compartments $1, 2, \ldots, 6$ at some time t far away in the future?

1.32 In an $M/M/\infty$-queueing system, customers arrive according to a Poisson process with rate a. Every customer who enters is immediately served by one of an infinite number of servers; hence there is no queue. The service times are exponentially distributed, with mean $1/b$. All service and interarrival times are independent. Let X_t be the number of customers in the system at time t. Show that the limiting distribution of X_t, as $t \to \infty$, is Poisson with parameter a/b.

Optimization

1.33 Let $\mathbf{a}$ and let $\mathbf{x}$ be n-dimensional column vectors. Show that $\nabla_{\mathbf{x}}\, \mathbf{a}^\top \mathbf{x} = \mathbf{a}$.

1.34 Let A be a symmetric $n \times n$ matrix and $\mathbf{x}$ be an n-dimensional column vector. Show that $\nabla_{\mathbf{x}}\, \frac{1}{2}\, \mathbf{x}^\top A \mathbf{x} = A\mathbf{x}$. What is the gradient if A is not symmetric?

1.35 Show that the optimal distribution $\mathbf{p}^*$ in Example 1.16 is given by the uniform distribution.

1.36 Derive the program (1.71).

1.37 Consider the MinxEnt program

$$\min_{\mathbf{p}} \quad \sum_{i=1}^{n} p_i \ln \frac{p_i}{q_i}$$

$$\text{subject to:} \quad \mathbf{p} \geqslant \mathbf{0}, \quad A\mathbf{p} = \mathbf{b}, \quad \sum_{i=1}^{n} p_i = 1 \,,$$

where $\mathbf{p}$ and $\mathbf{q}$ are probability distribution vectors and A is an $m \times n$ matrix.

a) Show that the Lagrangian for this problem is of the form

$$\mathcal{L}(\mathbf{p}, \boldsymbol{\lambda}, \beta, \boldsymbol{\mu}) = \mathbf{p}^\top \boldsymbol{\xi}(\mathbf{p}) - \boldsymbol{\lambda}^\top (A\mathbf{p} - \mathbf{b}) - \boldsymbol{\mu}^\top \mathbf{p} + \beta(\mathbf{1}^\top \mathbf{p} - 1) \,.$$

b) Show that $p_i = q_i \exp(-\beta - 1 + \mu_i + \sum_{j=1}^{m} \lambda_j \, a_{ji})$, for $i = 1, \ldots, n$.

c) Explain why, as a result of the KKT conditions, the optimal $\boldsymbol{\mu}^*$ must be equal to the zero vector.

d) Show that the solution to this MinxEnt program is exactly the same as for the program where the nonnegativity constraints are omitted.

Further Reading

An easy introduction to probability theory with many examples is [13], and a more detailed textbook is [8]. A classical reference is [6]. An accurate and accessible treatment of various stochastic processes is given in [3]. For convex optimization we refer to [2] and [7].

REFERENCES

1. Z. I. Botev, D. P. Kroese, and T. Taimre. Generalized cross-entropy methods for rare-event simulation and optimization. *Simulation: Transactions of the Society for Modeling and Simulation International*, 83(11):785–806, 2007.
2. S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, Cambridge, UK, 2004.
3. E. Çinlar. *Introduction to Stochastic Processes*. Prentice Hall, Englewood Cliffs, NJ, 1975.
4. T. M. Cover and J. A. Thomas. *Elements of Information Theory*. John Wiley & Sons, New York, 1991.
5. C. W. Curtis. *Linear Algebra: An Introductory Approach*. Springer-Verlag, New York, 1984.
6. W. Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. John Wiley & Sons, New York, 2nd edition, 1970.
7. R. Fletcher. *Practical Methods of Optimization*. John Wiley & Sons, New York, 1987.
8. G. R. Grimmett and D. R. Stirzaker. *Probability and Random Processes*. Oxford University Press, Oxford, 3rd edition, 2001.
9. J. N. Kapur and H. K. Kesavan. *Entropy Optimization Principles with Applications*. Academic Press, New York, 1992.

10. A. I. Khinchin. *Information Theory*. Dover Publications, New York, 1957.

11. N. V. Krylov. *Introduction to the Theory of Random Processes*, volume 43 of *Graduate Studies in Mathematics*. American Mathematical Society, Providence, RI, 2002.

12. E. L. Lehmann. *Testing Statistical Hypotheses*. Springer-Verlag, New York, 1997.

13. S. M. Ross. *A First Course in Probability*. Prentice Hall, Englewood Cliffs, NJ, 7th edition, 2005.

CHAPTER 2

RANDOM NUMBER, RANDOM VARIABLE, AND STOCHASTIC PROCESS GENERATION

2.1 INTRODUCTION

This chapter deals with the computer generation of random numbers, random variables, and stochastic processes. In a typical stochastic simulation, randomness is introduced into simulation models via independent uniformly distributed random variables. These random variables are then used as building blocks to simulate more general stochastic systems.

The rest of this chapter is organized as follows. We start, in Section 2.2, with the generation of uniform random variables. Section 2.3 discusses general methods for generating one-dimensional random variables. Section 2.4 presents specific algorithms for generating variables from commonly used continuous and discrete distributions. In Section 2.5 we discuss the generation of random vectors. Sections 2.6 and 2.7 treat the generation of Poisson processes, Markov chains, and Markov jump processes. The generation of Gaussian and diffusion processes is given in Sections 2.8 and 2.9. Finally, Section 2.10 deals with the generation of random permutations.

2.2 RANDOM NUMBER GENERATION

In the early days of simulation, randomness was generated by *manual* techniques, such as coin flipping, dice rolling, card shuffling, and roulette spinning. Later on,

Simulation and the Monte Carlo Method, Third Edition. By R. Y. Rubinstein and D. P. Kroese

physical devices, such as noise diodes and Geiger counters, were attached to computers for the same purpose. The prevailing belief held that only mechanical or electronic devices could produce truly random sequences. Although mechanical devices are still widely used in gambling and lotteries, these methods were abandoned by the computer-simulation community for several reasons: (1) mechanical methods were too slow for general use, (2) the generated sequences could not be reproduced, and (3) it was found that the generated numbers exhibit both bias and dependence. Although certain modern physical generation methods are fast and would pass most statistical tests for randomness (e.g., those based on the universal background radiation or the noise of a PC chip), their main drawback remains their lack of repeatability. Most of today's random number generators are not based on physical devices but on simple algorithms that can be easily implemented on a computer. They are fast, require little storage space, and can readily reproduce a given sequence of random numbers. Importantly, a good random number generator captures all the important statistical properties of true random sequences, even though the sequence is generated by a deterministic algorithm. For this reason these generators are sometimes called *pseudorandom*.

Most computer languages already contain a built-in pseudorandom number generator. The user is typically requested only to input the initial seed, X_0, and upon invocation the random number generator produces a sequence of independent, uniform $(0, 1)$ random variables. We therefore assume in this book the availability of such a "black box" that is capable of producing a stream of pseudorandom numbers. In Matlab, for example, this is provided by the `rand` function. The "seed" of the random number generator, which can be set by the `rng` function, determines which random stream is used, and this is very useful for testing purposes.

■ EXAMPLE 2.1 Generating Uniform Random Variables in Matlab

This example illustrates the use of the `rand` function in Matlab to generate samples from the $\mathsf{U}(0, 1)$ distribution. For clarity we have omitted the "`ans =` " output in the Matlab session below.

```
>> rand                 % generate a uniform random number
    0.0196
>> rand                 % generate another uniform random number
    0.823
>> rand(1,4)            % generate a uniform random vector
    0.5252  0.2026  0.6721  0.8381
>> rng(1234)            % set the seed to 1234
>> rand                 % generate a uniform random number
    0.1915
>> rng(1234)            % reset the seed to 1234
>> rand
    0.1915               % the previous outcome is repeated
```

The simplest methods for generating pseudorandom sequences use the so-called *linear congruential generators*, introduced in [12]. These generate a deterministic sequence of numbers by means of the recursive formula

$$X_{t+1} = aX_t + c \pmod{m} , \tag{2.1}$$

where the initial value, X_0, is called the *seed* and the a, c, and m (all positive integers) are called the *multiplier*, the *increment*, and the *modulus*, respectively. Note that applying the modulo-m operator in (2.1) means that $aX_i + c$ is divided by m, and the remainder is taken as the value for X_{t+1}. Thus each *state* X_t can only assume a value from the set $\{0, 1, \ldots, m-1\}$, and the quantities

$$U_t = \frac{X_t}{m} \,, \tag{2.2}$$

called *pseudorandom numbers*, constitute approximations to a true sequence of uniform random variables. Note that the sequence $X_0, X_1, X_2, \ldots$ will repeat itself after at most m steps and will therefore be periodic, with a period not exceeding m. For example, let $a = c = X_0 = 3$ and $m = 5$. Then the sequence obtained from the recursive formula $X_{t+1} = 3X_t + 3 \pmod 5$ is $3, 2, 4, 0, 3$, which has period 4. A careful choice of a, m, and c can lead to a linear congruential generator that passes most standard statistical tests for uniformity and independence. An example is the linear congruential generator with $m = 2^{31} - 1$, $a = 7^4$, and $c = 0$ by Lewis, Goodman, and Miller [13]. However, nowadays linear congruential generators no longer meet the requirements of modern Monte Carlo applications (e.g., [11]) and have been replaced by more general linear recursive algorithms, which are discussed next.

2.2.1 Multiple Recursive Generators

A *multiple-recursive generator* (MRG) of *order* k is determined by a sequence of k-dimensional state vectors $\mathbf{X}_t = (X_{t-k+1}, \ldots, X_t)^\top$, $t = 0, 1, 2, \ldots$, whose components satisfy the linear recurrence

$$X_t = (a_1 X_{t-1} + \cdots + a_k X_{t-k}) \bmod m, \quad t = k, k+1, \ldots \tag{2.3}$$

for some modulus m, multipliers $\{a_i, i = 1, \ldots, k\}$, and a given seed $\mathbf{X}_0 = (X_{-k+1}, \ldots, X_0)$. The maximum period length for this generator is $m^k - 1$. To yield fast algorithms, all but a few of the multipliers should be 0. When m is a large integer, the output stream of random numbers is obtained via $U_t = X_t/m$.

MRGs with very large periods can be implemented efficiently by combining several smaller period MRGs — yielding *combined multiple-recursive generators*.

■ **EXAMPLE 2.2**

One of the most successful combined MRGs is `MRG32k3a` by L'Ecuyer [9], which employs two MRGs of order 3, with recurrences

$$X_t = (1403580\, X_{t-2} - 810728\, X_{t-3}) \bmod m_1 \quad (m_1 = 2^{32} - 209) \,,$$

$$Y_t = (527612\, Y_{t-1} - 1370589\, Y_{t-3}) \bmod m_2 \quad (m_2 = 2^{32} - 22853) \,,$$

and output

$$U_t = \begin{cases} \dfrac{X_t - Y_t + m_1}{m_1 + 1} & \text{if } X_t \leqslant Y_t \,, \\[2ex] \dfrac{X_t - Y_t}{m_1 + 1} & \text{if } X_t > Y_t \,. \end{cases}$$

The period length is approximately 3×10^{57}. The generator `MRG32k3a` passes all statistical tests in today's most comprehensive test suit *TestU01* [11] and has been implemented in many software packages, including Matlab, Mathematica, Intel's MKL Library, SAS, VSL, Arena, and Automod. It is also the core generator in L'Ecuyer's SSJ simulation package, and is easily extendable to generate multiple random streams.

2.2.2 Modulo 2 Linear Generators

Good random generators must have very large state spaces. For a linear congruential generator, this means that the modulus m must be a large integer. However, for multiple recursive generators, it is not necessary to take a large modulus, as the period length can be as large as $m^k - 1$. Because binary operations are in general faster than floating point operations (which are in turn faster than integer operations), it makes sense to consider MRGs and other random number generators that are based on linear recurrences modulo 2. A general framework for such random number generators is given in [10], where the state is a k-bit vector $\mathbf{X}_t = (X_{t,1}, \ldots, X_{t,k})^\top$ that is mapped via a linear transformation to a w-bit output vector $\mathbf{Y}_t = (Y_{t,1}, \ldots, Y_{t,w})^\top$, from which the random number $U_t \in (0, 1)$ is obtained by *bitwise decimation* as follows:

Algorithm 2.2.1: Generic Linear Recurrence Modulo 2 Generator

input : Seed distribution μ on state space $\mathcal{S} = \{0,1\}^k$, and sample size N.
output: Sequence $U_1, \ldots, U_N$ of pseudo-random numbers.

Draw the seed $\mathbf{X}_0$ from the distribution μ. `// initialize`
for $t = 1$ **to** N **do**
  $\mathbf{X}_t \leftarrow A\mathbf{X}_{t-1}$ `// transition`
  $\mathbf{Y}_t \leftarrow B\mathbf{X}_t$ `// output transformation`
  $U_t \leftarrow \sum_{\ell=1}^{w} Y_{t,\ell}\, 2^{-\ell}$ `// decimation`
return $U_1, \ldots, U_N$

Here, A and B are $k \times k$ and $w \times k$ binary matrices, respectively, and all operations are performed modulo 2. In particular, addition corresponds to the bitwise `XOR` operation (in particular, $1 + 1 = 0$). The integer w can be thought of as the word length of the computer (i.e., $w = 32$ or 64). Usually (but there are exceptions, see [10]) k is taken much larger than w.

EXAMPLE 2.3 Mersenne Twister

A popular modulo 2 generator was introduced by Matsumoto and Nishimura [16]. The dimension k of the state vector $\mathbf{X}_t$ in Algorithm 2.2.1 is in this case $k = w\,n$, where w is the word length (default 32) and n a large integer (default 624). The period length for the default choice of parameters can be shown to be $2^{w(n-1)+1} - 1 = 2^{19937} - 1$. Rather than take the state $\mathbf{X}_t$ as a $w\,n \times 1$ vector, it is convenient to consider it as an $n \times w$ matrix with rows $\mathbf{x}_t, \ldots, \mathbf{x}_{t+n-1}$. Starting from the seed rows $\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}$, at each step $t = 0, 1, 2, \ldots$ the $(t+n)$-th row is calculated according to the following rules:

1. Take the first r bits of $\mathbf{x}_t$ and the last $w-r$ bits of $\mathbf{x}_{t+1}$ and concenate them together in a binary vector $\mathbf{x}$.

2. Apply the following binary operation to $\mathbf{x} = (x_1, \ldots, x_w)$ to give a new binary vector $\widetilde{\mathbf{x}}$:
$$\widetilde{\mathbf{x}} = \begin{cases} \mathbf{x} \gg 1 & \text{if } x_w = 0 \ , \\ (\mathbf{x} \gg 1) \oplus \mathbf{a} & \text{if } x_w = 1 \ . \end{cases}$$

3. Let $\mathbf{x}_{t+n} = \mathbf{x}_{t+m} \oplus \widetilde{\mathbf{x}}$.

Here $\oplus$ stands for the `XOR` operation and $\gg 1$ for the rightshift operation (shift the bits one position to the right, adding a 1 from the left). The binary vector $\mathbf{a}$ and the numbers m and r are specified by the user (see below).

The output at step t of the algorithm is performed as the bitwise decimation of a vector $\mathbf{y}$ that is obtained via the following five steps:

1. $\mathbf{y} = \mathbf{x}_{t+n}$
2. $\mathbf{y} = \mathbf{y} \oplus (\mathbf{y} \gg u)$
3. $\mathbf{y} = \mathbf{y} \oplus ((\mathbf{y} \ll s) \ \& \ \mathbf{b})$
4. $\mathbf{y} = \mathbf{y} \oplus ((\mathbf{y} \ll v) \ \& \ \mathbf{c})$
5. $\mathbf{y} = \mathbf{y} \oplus (\mathbf{y} \gg l)$

Here & denotes the `AND` operation (bitwise multiplication), and $(\mathbf{y} \ll s)$ indicates a leftshift by s positions, adding 0s from the right; similarly, $(\mathbf{y} \gg u)$ is a rightshift by u positions, adding 1s from the left. The vectors $\mathbf{b}$ and $\mathbf{c}$ as well as the integers u and s are provided by the user. The recommended parameters for the algorithm are

$$(w, n, m, r) = (32, 624, 397, 31)$$

and

$$(\mathbf{a}, \mathbf{b}, \mathbf{c}, u, s, v, l) = (9908\text{B}0\text{DF}_{16}, 9\text{D}2\text{C}5680_{16}, \text{EFC}60000_{16}, 11, 7, 15, 18) \ ,$$

where the subscript $_{16}$ indicates hexadecimal notation; for example, $7\text{B}_{16} = 0111\ 1101$.

As a concrete example of the workings of the Mersenne twister, suppose that the seed values are as in Table 2.1 (assuming default parameters).

Table 2.1: Initial state of the Mersenne twister.

$\mathbf{x}_0$	00110110110100001010111000101001
$\mathbf{x}_1$	10101010101010101010101010101011
$\mathbf{x}_2$	10001000100000001111100010101011
$\vdots$	$\vdots$
$\mathbf{x}_m$	10010101110100010101011110100011
$\vdots$	$\vdots$
$\mathbf{x}_{n-1}$	00100111001010101110110101100 10

Let us generate the first random number. Suppose that $r = 31$, so,

$$\mathbf{x} = \underbrace{00110110110100000101011100010100}_{r \text{ bits of } \mathbf{x}_0} \quad \underbrace{1}_{w-r \text{ bits of } \mathbf{x}_1} .$$

The least significant bit (most right bit) of $\mathbf{x}$ is 1, so $\widetilde{\mathbf{x}}$ is obtained by right-shifting $\mathbf{x}$ and XOR-ing the resulting vector with $\mathbf{a}$, so

$$\begin{aligned}\widetilde{\mathbf{x}} &= 10011011011010000101011100010100 \oplus 00000001011010100011010010111001 \\ &= 10011010000000100110001110101101.\end{aligned}$$

We complete the calculation of $\mathbf{x}_n = \mathbf{x}_m \oplus \widetilde{\mathbf{x}}$ and get

$$\begin{aligned}\mathbf{x}_n &= 10010101110100010101011110100011 \oplus 10011010000000100110001110101101 = \\ &= 00001111110100110011010000001110.\end{aligned}$$

Next, we determine the output vector $\mathbf{y}$ in five steps.

1. $\mathbf{y} = \mathbf{x}_n = 00001111110100110011010000001110$.

2. $\mathbf{y} = \mathbf{y} \oplus (\mathbf{y} \gg 11)$
   $= 00001111110100110011010000001110 \oplus 11111111111000011111101001100110$
   $= 11110000001100101100111001101000$.

3. $\mathbf{y} = \mathbf{y} \oplus ((\mathbf{y} \ll 7)\ \&\ 9\text{D2C}5680_{16})$

   $= 11110000001100101100111001101000$
   $\oplus (00011001011001110011010000000000 \& 00000000001100101100111001101000)$
   $= 11110000001100101100111001101000 \oplus 00000000001000100000010000000000$
   $= 11110000000100001100101001101000$.

4. $\mathbf{y} = \mathbf{y} \oplus ((\mathbf{y} \ll 15)\ \&\ 00000000000000000110001111110111)$ — similar to 3, which results in
   $$\mathbf{y} = 11110001010100000111110100 .$$

5. $\mathbf{y} = \mathbf{y} \oplus (\mathbf{y} \gg 18)$ — similar to 2, which results in
   $$\mathbf{y} = 00001110101011110000011000111100 .$$

   Having in mind that the decimal representation of final $\mathbf{y}$ is equal to $1.0130 \cdot 10^9$, the algorithm returns $1.0130 \cdot 10^9 / \left(2^{32} - 1\right) = 0.2359$ as an output.

For an updated version of the code of the Mersenne twister we refer to

`http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html`

Note that for initialization, a full $w \times n$ matrix has to be specified. This is often done by running a basic linear generator. The algorithm is very fast when implemented in compiled languages such as C and passes most statistical test but is known to recover too slowly from the states near zero; see [11, Page 23].

2.3 RANDOM VARIABLE GENERATION

In this section we discuss various general methods for generating one-dimensional random variables from a prescribed distribution. We consider the inverse-transform method, the alias method, the composition method, and the acceptance–rejection method.

2.3.1 Inverse-Transform Method

Let X be a random variable with cdf F. Since F is a nondecreasing function, the inverse function F^{-1} may be defined as

$$F^{-1}(y) = \inf\{x : F(x) \geqslant y\} , \quad 0 \leqslant y \leqslant 1 . \tag{2.4}$$

(Readers not acquainted with the notion inf should read min.) It is easy to show that if $U \sim \mathsf{U}(0,1)$, then

$$X = F^{-1}(U) \tag{2.5}$$

has cdf F. That is to say, since F is invertible and $\mathbb{P}(U \leqslant u) = u$, we have

$$\mathbb{P}(X \leqslant x) = \mathbb{P}(F^{-1}(U) \leqslant x) = \mathbb{P}(U \leqslant F(x)) = F(x) . \tag{2.6}$$

Thus, to generate a random variable X with cdf F, draw $U \sim \mathsf{U}(0,1)$ and set $X = F^{-1}(U)$. Figure 2.1 illustrates the inverse-transform method given by the following algorithm:

Algorithm 2.3.1: Inverse-Transform Method

input : Cumulative distribution function F.
output: Random variable X distributed according to F.

1 Generate U from $\mathsf{U}(0,1)$.
2 $X \leftarrow F^{-1}(U)$
3 **return** X

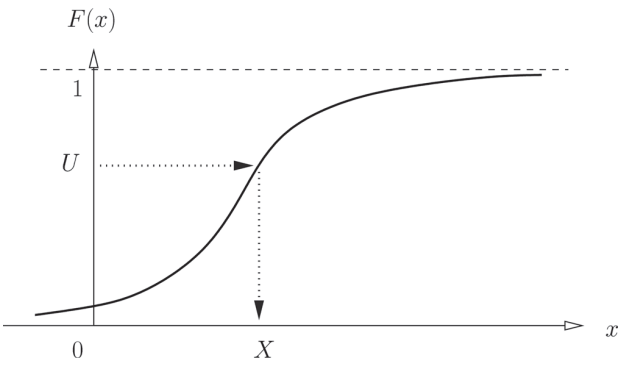

Figure 2.1: Inverse-transform method.

■ **EXAMPLE 2.4**

Generate a random variable from the pdf

$$f(x) = \begin{cases} 2x, & 0 \leqslant x \leqslant 1, \\ 0 & \text{otherwise.} \end{cases} \tag{2.7}$$

The cdf is

$$F(x) = \begin{cases} 0, & x < 0, \\ \int_0^x 2y \, \mathrm{d}y = x^2, & 0 \leqslant x \leqslant 1, \\ 1, & x > 1. \end{cases}$$

Applying (2.5), we have

$$X = F^{-1}(U) = \sqrt{U} \, .$$

Therefore, to generate a random variable X from the pdf (2.7), first generate a random variable U from $\mathsf{U}(0,1)$ and then take its square root.

■ **EXAMPLE 2.5 Order Statistics**

Let $X_1, \ldots, X_n$ be iid random variables with cdf F. We wish to generate random variables $X_{(n)}$ and $X_{(1)}$ that are distributed according to the order statistics $\max(X_1, \ldots, X_n)$ and $\min(X_1, \ldots, X_n)$, respectively. From Example 1.7 we see that the cdfs of $X_{(n)}$ and $X_{(1)}$ are $F_n(x) = [F(x)]^n$ and $F_1(x) = 1 - [1 - F(x)]^n$, respectively. Applying (2.5), we get

$$X_{(n)} = F^{-1}(U^{1/n}) \, ,$$

and, since $1 - U$ is also from $\mathsf{U}(0,1)$,

$$X_{(1)} = F^{-1}(1 - U^{1/n}) \, .$$

In the special case where $F(x) = x$, that is, $X_i \sim \mathsf{U}(0,1)$, we have

$$X_{(n)} = U^{1/n} \qquad \text{and} \qquad X_{(1)} = 1 - U^{1/n} \, .$$

■ **EXAMPLE 2.6 Drawing from a Discrete Distribution**

Let X be a discrete random variable with $\mathbb{P}(X = x_i) = p_i, \ i = 1, 2, \ldots,$ with $\sum_i p_i = 1$ and $x_1 < x_2 < \ldots$. The cdf F of X is given by $F(x) = \sum_{i : x_i \leqslant x} p_i, \ i = 1, 2, \ldots$ and is illustrated in Figure 2.2.

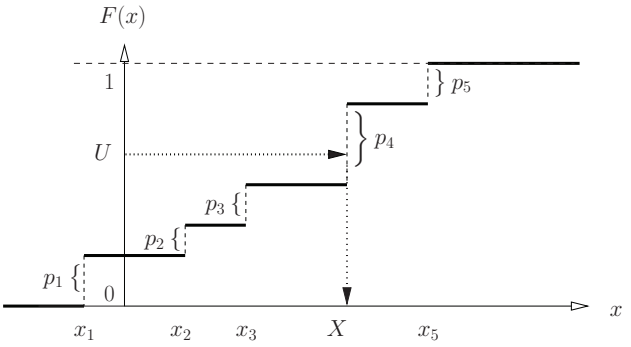

Figure 2.2: Inverse-transform method for a discrete random variable.

Hence the algorithm for generating a random variable from F can be written as follows:

Algorithm 2.3.2: Inverse-Transform Method for a Discrete Distribution

input : Discrete cumulative distribution function F.
output: Discrete random variable X distributed according to F.
1 Generate $U \sim \mathsf{U}(0,1)$.
2 Find the smallest positive integer, k, such that $U \leqslant F(x_k)$. Let $X \leftarrow x_k$.
3 **return** X

Much of the execution time in Algorithm 2.3.2 is spent in making the comparisons of Step 2. This time can be reduced by using efficient search techniques (see [2]).

In general, the inverse-transform method requires that the underlying cdf, F, exist in a form for which the corresponding inverse function F^{-1} can be found analytically or algorithmically. Applicable distributions are, for example, the exponential, uniform, Weibull, logistic, and Cauchy distributions. Unfortunately, for many other probability distributions, it is either impossible or difficult to find the inverse transform, that is, to solve

$$F(x) = \int_{-\infty}^{x} f(t)\,\mathrm{d}t = u$$

with respect to x. Even in the case where F^{-1} exists in an explicit form, the inverse-transform method may not necessarily be the most efficient random variable generation method (see [2]).

2.3.2 Alias Method

An alternative to the inverse-transform method for generating discrete random variables, which does not require time-consuming search techniques as per Step 2 of Algorithm 2.3.2, is the so-called *alias method* [19]. It is based on the fact that an arbitrary discrete n-point pdf f, with

$$f(x_i) = \mathbb{P}(X = x_i), \quad i = 1, \ldots, n\,,$$

can be represented as an equally weighted mixture of n pdfs, $q^{(k)}$, $k = 1, \ldots, n$, each having at most *two* nonzero components. That is, any n-point pdf f can be

represented as

$$f(x) = \frac{1}{n} \sum_{k=1}^{n} q^{(k)}(x) \tag{2.8}$$

for suitably defined two-point pdfs $q^{(k)},\ k = 1, \ldots, n$; see [19].

The alias method is rather general and efficient but requires an initial setup and extra storage for the n pdfs, $q^{(k)}$. A procedure for computing these two-point pdfs can be found in [2]. Once the representation (2.8) has been established, generation from f is simple and can be written as follows:

Algorithm 2.3.3: Alias Method

input : Two-point pdfs $q^{(k)}, k = 1, \ldots, n$ representing discrete pdf f.
output: Discrete random variable X distributed according to f.
1 Generate $U \sim \mathsf{U}(0, 1)$ and set $K \leftarrow \lceil n\,U \rceil$.
2 Generate X from the two-point pdf $q^{(K)}$.
3 **return** X

2.3.3 Composition Method

This method assumes that a cdf, F, can be expressed as a *mixture* of cdfs $\{G_i\}$, that is,

$$F(x) = \sum_{i=1}^{m} p_i\, G_i(x)\ , \tag{2.9}$$

where

$$p_i > 0, \quad \sum_{i=1}^{m} p_i = 1\ .$$

Let $X_i \sim G_i$ and let Y be a discrete random variable with $\mathbb{P}(Y = i) = p_i$, independent of X_i, for $1 \leqslant i \leqslant m$. Then a random variable X with cdf F can be represented as

$$X = \sum_{i=1}^{m} X_i\, I_{\{Y=i\}}\ .$$

It follows that in order to generate X from F, we must first generate the discrete random variable Y and then, given $Y = i$, generate X_i from G_i. We thus have the following method:

Algorithm 2.3.4: Composition Method

input : Mixture cdf F.
output: Random variable X distributed according to F.
1 Generate the random variable Y according to $\mathbb{P}(Y = i) = p_i,\ i = 1, \ldots, m$.
2 Given $Y = i$, generate X from the cdf G_i.
3 **return** X

2.3.4 Acceptance–Rejection Method

The inverse-transform and composition methods are direct methods in the sense that they deal directly with the cdf of the random variable to be generated. The acceptance–rejection method, is an indirect method due to Stan Ulam and John von Neumann. It can be applied when the above-mentioned direct methods either fail or turn out to be computationally inefficient.

To introduce the idea, suppose that the *target* pdf f (the pdf from which we want to sample) is bounded on some finite interval $[a, b]$ and is zero outside this interval (see Figure 2.3). Let

$$c = \sup\{f(x) : x \in [a, b]\} .$$

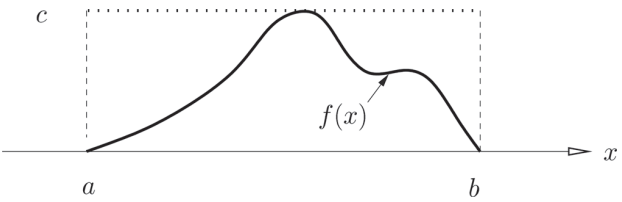

Figure 2.3: The acceptance–rejection method.

In this case, generating a random variable $Z \sim f$ is straightforward, and it can be done using the following acceptance–rejection steps:

1. Generate $X \sim \mathsf{U}(a, b)$.
2. Generate $Y \sim \mathsf{U}(0, c)$ independently of X.
3. If $Y \leqslant f(X)$, return $Z = X$. Otherwise, return to Step 1.

It is important to note that each generated vector (X, Y) is uniformly distributed over the rectangle $[a, b] \times [0, c]$. Therefore the accepted pair (X, Y) is uniformly distributed under the graph f. This implies that the distribution of the accepted values of X has the desired pdf f.

We can generalize this as follows: Let g be an arbitrary density such that $\phi(x) = C\, g(x)$ *majorizes* $f(x)$ for some constant C (Figure 2.4); that is, $\phi(x) \geqslant f(x)$ for all x. Note that of necessity $C \geqslant 1$. We call $g(x)$ the *proposal* pdf and assume that it is easy to generate random variables from it.

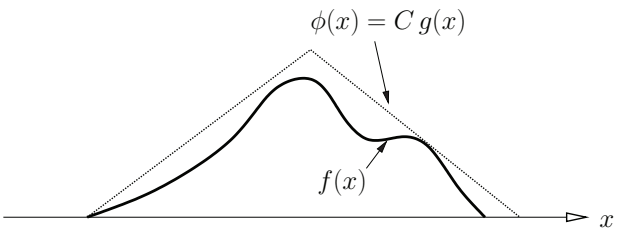

Figure 2.4: The acceptance–rejection method with a majorizing function ϕ.

The acceptance–rejection algorithm can be written as follows:

Algorithm 2.3.5: Acceptance–Rejection Method

input : Pdf g and constant C such that $Cg(x) \geqslant f(x)$ for all x.
output: Random variable X distributed according to pdf f.

```
1 found ← false
2 while not found do
3 |   Generate X from g(x).
4 |   Generate Y ~ U(0, C g(X)).
5 |   if Y ⩽ f(X) then found ← true
6 return X
```

The theoretical basis of the acceptance–rejection method is provided by the following theorem:

Theorem 2.3.1 *The random variable generated according to Algorithm 2.3.5 has the desired pdf $f(x)$.*

Proof: Define the following two subsets:

$$\mathscr{A} = \{(x, y) : 0 \leq y \leq Cg(x)\} \quad \text{and} \quad \mathscr{B} = \{(x, y) : 0 \leq y \leq f(x)\}\,, \tag{2.10}$$

which represent the areas below the curves $Cg(x)$ and $f(x)$, respectively. Note first that Lines 3 and 4 of Algorithm 2.3.5 imply that the random vector (X, Y) is uniformly distributed on $\mathscr{A}$. To see this, let $q(x, y)$ denote the joint pdf of (X, Y), and let $q(y \,|\, x)$ denote the conditional pdf of Y given $X = x$. Then we have

$$q(x, y) = \begin{cases} g(x)\, q(y \,|\, x) & \text{if} (x, y) \in \mathscr{A}, \\ 0 & \text{otherwise.} \end{cases} \tag{2.11}$$

Now Line 4 states that $q(y \,|\, x)$ equals $1/(Cg(x))$ for $y \in [0, Cg(x)]$ and is zero otherwise. Therefore, $q(x, y) = C^{-1}$ for every $(x, y) \in \mathscr{A}$.

Let (X^*, Y^*) be the first accepted point, that is, the first point that is in $\mathscr{B}$. Since the vector (X, Y) is uniformly distributed on $\mathscr{A}$, the vector (X^*, Y^*) is uniformly distributed on $\mathscr{B}$. Also, since the area of $\mathscr{B}$ equals unity, the joint pdf of (X^*, Y^*) on $\mathscr{B}$ equals unity as well. Thus, the marginal pdf of $Z = X^*$ is

$$\int_0^{f(x)} 1 \,\mathrm{d}y = f(x)\,.$$

□

The *efficiency* of Algorithm 2.3.5 is defined as

$$\mathbb{P}\left((X, Y) \text{ is accepted}\right) = \frac{\text{area } \mathscr{B}}{\text{area } \mathscr{A}} = \frac{1}{C}\,. \tag{2.12}$$

Often, a slightly modified version of Algorithm 2.3.5 is used. This is because $Y \sim \mathsf{U}(0, C\, g(X))$ in Line 4 is the same as setting $Y = U\, C\, g(X)$, where $U \sim \mathsf{U}(0, 1)$, and we can then write $Y \leqslant f(X)$ in Line 5 as $U \leqslant f(X)/(C\, g(X))$. In other words, generate X from $g(x)$ and accept it with probability $f(X)/(C\, g(X))$; otherwise reject X and try again. Thus the modified version of Algorithm 2.3.5 can be rewritten as follows:

Algorithm 2.3.6: Modified Acceptance–Rejection Method

input : Pdf g and constant C such that $Cg(x) \geqslant f(x)$ for all x.
output: Random variable X distributed according to pdf f.

```
found ← false
while not found do
    Generate X from g(x).
    Generate U from U(0,1) independently of X.
    if U ⩽ f(X)/(Cg(X)) then found ← true
return X
```

EXAMPLE 2.7 Example 2.4 (Continued)

We show how to generate a random variable Z from the pdf

$$f(x) = \begin{cases} 2\,x, & 0 < x < 1, \\ 0 & \text{otherwise}, \end{cases}$$

using the acceptance–rejection method. For simplicity, take $g(x) = 1,\ 0 < x < 1$, and $C = 2$. In this case our proposal distribution is simply the uniform distribution on $(0, 1)$. Consequently, $f(x)/(C\,g(x)) = x$ and Algorithm 2.3.6 becomes: keep generating X and U independently from $\mathsf{U}(0, 1)$ until $U \leqslant X$; then return X. Note that this example is merely illustrative, since it is more efficient to simulate from this pdf using the inverse-transform method.

As a consequence of (2.12), the efficiency of the modified acceptance–rejection method is determined by the acceptance probability $p = \mathbb{P}(U \leqslant f(X)/(Cg(X))) = \mathbb{P}(Y \leqslant f(X)) = 1/C$ for each trial (X, U). Since the trials are independent, the number of trials, N, before a successful pair (Z, U) occurs has the following geometric distribution:

$$\mathbb{P}(N = n) = p\,(1 - p)^{n-1}, \quad n = 1, 2, \ldots, \tag{2.13}$$

with the expected number of trials equal to $1/p = C$.

For this method to be of practical interest, the following criteria must be used in selecting the proposal density $g(x)$:

1. It should be easy to generate a random variable from $g(x)$.
2. The efficiency, $1/C$, of the procedure should be large; that is, C should be close to 1 (which occurs when $g(x)$ is close to $f(x)$).

EXAMPLE 2.8

Generate a random variable Z from the semicircular density

$$f(x) = \frac{2}{\pi R^2}\sqrt{R^2 - x^2}, \quad -R \leqslant x \leqslant R\ .$$

Take the proposal distribution to be uniform over $[-R, R]$; that is, take $g(x) = 1/(2R)$, $-R \leqslant x \leqslant R$ and choose C as small as possible such that $Cg(x) \geqslant$

$f(x)$; hence $C = 4/\pi$. Then Algorithm 2.3.6 leads to the following generation algorithm:

1. *Generate two independent random variables, U_1 and U_2, from* $\mathsf{U}(0,1)$.

2. *Use U_2 to generate X from $g(x)$ via the inverse-transform method, namely $X = (2U_2 - 1)R$, and calculate*

$$\frac{f(X)}{C\, g(X)} = \sqrt{1 - (2U_2 - 1)^2}\ .$$

3. *If $U_1 \leqslant f(X)/(C\, g(X))$, which is equivalent to $(2U_2 - 1)^2 \leqslant 1 - U_1^2$, return $Z = X = (2U_2 - 1)R$; otherwise, return to Step 1.*

The expected number of trials for this algorithm is $C = 4/\pi$, and the efficiency is $1/C = \pi/4 \approx 0.785$.

2.4 GENERATING FROM COMMONLY USED DISTRIBUTIONS

The next two subsections present algorithms for generating variables from commonly used continuous and discrete distributions. Of the numerous algorithms available (e.g., [2]), we have tried to select those that are reasonably efficient and relatively simple to implement.

2.4.1 Generating Continuous Random Variables

2.4.1.1 Exponential Distribution We start by applying the inverse-transform method to the exponential distribution. If $X \sim \mathsf{Exp}(\lambda)$, then its cdf F is given by

$$F(x) = 1 - \mathrm{e}^{-\lambda x}, \qquad x \geqslant 0\ . \tag{2.14}$$

Hence, solving $u = F(x)$ in terms of x gives

$$F^{-1}(u) = -\frac{1}{\lambda}\ln(1-u)\ .$$

Because $U \sim \mathsf{U}(0,1)$ implies $1 - U \sim \mathsf{U}(0,1)$, we arrive at the following algorithm:

Algorithm 2.4.1: Generation of an Exponential Random Variable

input : $\lambda > 0$
output: Random variable X distributed according to $\mathsf{Exp}(\lambda)$.
1 Generate $U \sim \mathsf{U}(0,1)$.
2 $X \leftarrow -\frac{1}{\lambda}\ln U$
3 **return** X

There are many alternative procedures for generating variables from the exponential distribution. The interested reader is referred to [2].

2.4.1.2 Normal (Gaussian) Distribution If $X \sim \mathsf{N}(\mu, \sigma^2)$, its pdf is given by

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left\{-\frac{(x-\mu)^2}{2\sigma^2}\right\}, \qquad -\infty < x < \infty, \tag{2.15}$$

where μ is the mean (or expectation) and σ^2 the variance of the distribution.

Since inversion of the normal cdf is numerically inefficient, the inverse-transform method is not very suitable for generating normal random variables, so other procedures must be devised. We consider only generation from $\mathsf{N}(0,1)$ (standard normal variables), since any random $Z \sim \mathsf{N}(\mu, \sigma^2)$ can be represented as $Z = \mu + \sigma X$, where X is from $\mathsf{N}(0,1)$. One of the earliest methods for generating variables from $\mathsf{N}(0,1)$ was developed by Box and Muller as follows:

Let X and Y be two independent standard normal random variables; so (X, Y) is a random point in the plane. Let (R, Θ) be the corresponding polar coordinates. The joint pdf $f_{R,\Theta}$ of R and Θ is given by

$$f_{R,\Theta}(r,\theta) = \frac{1}{2\pi}\, \mathrm{e}^{-r^2/2}\, r \quad \text{for } r \geqslant 0 \text{ and } \theta \in [0, 2\pi).$$

This can be seen by writing x and y in terms of r and θ, to get

$$x = r\cos\theta \quad \text{and} \quad y = r\sin\theta. \tag{2.16}$$

The Jacobian of this coordinate transformation is

$$\det\begin{pmatrix} \frac{\partial x}{\partial r} & \frac{\partial x}{\partial \theta} \\ \frac{\partial y}{\partial r} & \frac{\partial y}{\partial \theta} \end{pmatrix} = \begin{vmatrix} \cos\theta & -r\sin\theta \\ \sin\theta & r\cos\theta \end{vmatrix} = r.$$

The result now follows from the transformation rule (1.20), noting that the joint pdf of X and Y is $f_{X,Y}(x,y) = \frac{1}{2\pi}\,\mathrm{e}^{-(x^2+y^2)/2}$. It is not difficult to verify that R and Θ are independent, that $\Theta \sim \mathsf{U}[0, 2\pi)$, and that $\mathbb{P}(R > r) = \mathrm{e}^{-r^2/2}$. This means that R has the same distribution as $\sqrt{V}$, with $V \sim \mathsf{Exp}(1/2)$. Namely, $\mathbb{P}(\sqrt{V} > v) = \mathbb{P}(V > v^2) = \mathrm{e}^{-v^2/2}$, $v \geqslant 0$. Thus both Θ and R are easy to generate and are transformed via (2.16) into independent standard normal random variables. This leads to the following algorithm:

Algorithm 2.4.2: Normal Random Variable Generation: Box–Muller Approach

output: Independent standard normal random variables X and Y.
1 Generate two independent random variables, U_1 and U_2, from $\mathsf{U}(0,1)$.
2 $X \leftarrow (-2\ln U_1)^{1/2}\cos(2\pi U_2)$
3 $Y \leftarrow (-2\ln U_1)^{1/2}\sin(2\pi U_2)$
4 **return** X, Y

An alternative generation method for $\mathsf{N}(0,1)$ is based on the acceptance–rejection method. First, note that in order to generate a random variable Y from $\mathsf{N}(0,1)$, one can first generate a positive random variable X from the pdf

$$f(x) = \sqrt{\frac{2}{\pi}}\, \mathrm{e}^{-x^2/2}, \qquad x \geqslant 0, \tag{2.17}$$

and then assign to X a random sign. The validity of this procedure follows from the symmetry of the standard normal distribution about zero.

To generate a random variable X from (2.17), we bound $f(x)$ by $C\,g(x)$, where $g(x) = \mathrm{e}^{-x}$ is the pdf of the $\mathsf{Exp}(1)$ distribution. The smallest constant C such that $f(x) \leqslant Cg(x)$ is $\sqrt{2\mathrm{e}/\pi}$ (see Figure 2.5). The efficiency of this method is therefore $\sqrt{\pi/2\mathrm{e}} \approx 0.76$.

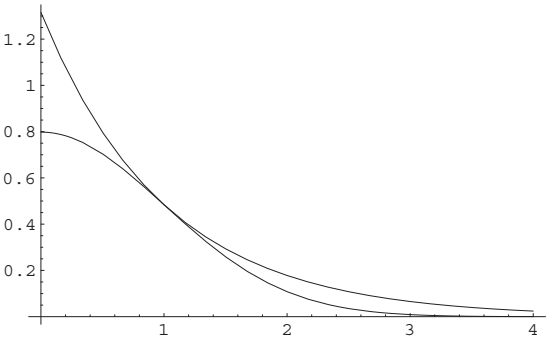

Figure 2.5: Bounding the positive normal density.

The acceptance condition, $U \leqslant f(X)/(C\mathrm{e}^{-X})$, can be written as

$$U \leqslant \exp[-(X-1)^2/2]\ , \tag{2.18}$$

which is equivalent to

$$-\ln U \geqslant \frac{(X-1)^2}{2}\ , \tag{2.19}$$

where X is from $\mathsf{Exp}(1)$. Since $-\ln U$ is also from $\mathsf{Exp}(1)$, the last inequality can be written as

$$V_1 \geqslant \frac{(V_2-1)^2}{2}\ , \tag{2.20}$$

where $V_1 = -\ln U$ and $V_2 = X$ are independent and both $\mathsf{Exp}(1)$ distributed.

2.4.1.3 Gamma Distribution If $X \sim \mathsf{Gamma}(\alpha, \lambda)$ then its pdf is of the form

$$f(x) = \frac{x^{\alpha-1}\lambda^{\alpha}\mathrm{e}^{-\lambda x}}{\Gamma(\alpha)}\ , \qquad x \geqslant 0\ . \tag{2.21}$$

The parameters $\alpha > 0$ and $\lambda > 0$ are called the *shape* and *scale* parameters, respectively. Since λ merely changes the scale, it suffices to consider only random variable generation of $\mathsf{Gamma}(\alpha, 1)$. In particular, if $X \sim \mathsf{Gamma}(\alpha, 1)$, then $X/\lambda \sim \mathsf{Gamma}(\alpha, \lambda)$ (see Exercise 2.16). Because the cdf for the gamma distributions does not generally exist in explicit form, the inverse-transform method cannot always be applied to generate random variables from this distribution. Alternative methods are thus called for. We discuss one such method for the case $\alpha \geqslant 1$. Let $f(x) = x^{\alpha-1}\mathrm{e}^{-x}/\Gamma(\alpha)$ and $\psi(x) = d\,(1 + c\,x)^3$, $x > -1/c$ and zero otherwise, where c and d are positive constants. Note that $\psi(x)$ is a strictly increasing function. Let Y

have density $k(y) = f(\psi(y))\,\psi'(y)\,c_1$, where c_1 is a normalization constant. Then $X = \psi(Y)$ has density f. Namely, by the transformation rule (1.16), we obtain

$$f_X(x) = \frac{k(\psi^{-1}(x))}{\psi'(\psi^{-1}(x))} = f(\psi(\psi^{-1}(x)))\,\frac{\psi'(\psi^{-1}(x))}{\psi'(\psi^{-1}(x))} = f(x)\ .$$

We draw Y via the acceptance–rejection method, using the standard normal distribution as our proposal distribution. We choose c and d such that $k(y) \leqslant C\varphi(y)$, with $C > 1$ close to 1, where φ is the pdf of the $\mathsf{N}(0,1)$ distribution. To find such c and d, we first write $k(y) = c_2\,\mathrm{e}^{h(y)}$, where some algebra will show that

$$h(y) = (1-3\,\alpha)\ln(1+c\,y) - d\,(1+c\,y)^3 + d\ .$$

(Note that $h(0) = 0$.) Next, a Taylor series expansion of $h(y)$ around 0 yields

$$h(y) = c\,(-1-3\,d+3\,\alpha)\,y - \frac{1}{2}\,c^2\,(-1+6\,d+3\,\alpha)\,y^2 + O(y^3)\ .$$

This suggests taking c and d such that the coefficients of y and y^2 in the expansion above are 0 and $-1/2$, respectively, as in the exponent of the standard normal density. So we take $d = \alpha - 1/3$ and $c = \frac{1}{3\sqrt{d}}$. It is not difficult to check that then

$$h(y) \leqslant -\frac{1}{2}\,y^2 \quad \text{for all} \quad y > -\frac{1}{c}\ ,$$

and therefore $\mathrm{e}^{h(y)} \leqslant \mathrm{e}^{-\frac{1}{2}y^2}$, which means that $k(y)$ is dominated by $c_2\sqrt{2\pi}\,\varphi(y)$ for all y. Hence, the acceptance–rejection method for drawing from $Y \sim k$ is as follows: Draw $Z \sim \mathsf{N}(0,1)$ and $U \sim \mathsf{U}(0,1)$ independently. If

$$U < \frac{c_2\,\mathrm{e}^{h(Z)}}{c_2\sqrt{2\pi}\,\varphi(Z)}\ ,$$

or equivalently, if

$$\ln U < h(Z) + \frac{1}{2}Z^2\ ,$$

then return $Y = Z$; otherwise, repeat (we set $h(Z) = -\infty$ if $Z \leqslant -1/c$). The efficiency of this method, $\int_{-1/c}^{\infty} \mathrm{e}^{h(y)}\mathrm{d}y / \int_{-\infty}^{\infty} \mathrm{e}^{-\frac{1}{2}y^2}\,\mathrm{d}y$, is greater than 0.95 for all values of $\alpha \geqslant 1$. Finally, we complete the generation of X by taking $X = \psi(Y)$. For the case where $\alpha < 1$, we can use the fact that if $X \sim \mathsf{Gamma}(1+\alpha, 1)$ and $U \sim \mathsf{U}(0,1)$ are independent, then $XU^{1/a} \sim \mathsf{Gamma}(\alpha, 1)$; see Problem 2.17. Summarizing, we have the following algorithm [15]:

Algorithm 2.4.3: Sampling from the $\mathsf{Gamma}(\alpha, \lambda)$ Distribution

function `Gamrnd`(α, λ)
  if $\alpha > 1$ **then**
    Set $d \leftarrow \alpha - 1/3$ and $c \leftarrow 1/\sqrt{9\,d}$
    `continue` $\leftarrow$ **true**
    while `continue` **do**
      Generate $Z \sim \mathsf{N}(0,1)$.
      if $Z > -1/c$ **then**
        $V \leftarrow (1 + cZ)^3$
        Generate $U \sim \mathsf{U}(0,1)$.
        if $\ln U < \frac{1}{2}Z^2 + dV + d\ln(V)$ **then** `continue` $\leftarrow$ **false**
    $X \leftarrow dV/\lambda$
  else
    $X \leftarrow$ `Gamrnd`$(\alpha + 1, \lambda)$
    Generate $U \sim \mathsf{U}(0,1)$.
    $X \leftarrow XU^{1/\alpha}$
  return X

A gamma distribution with an integer shape parameter, say $\alpha = m$, is also called an *Erlang distribution*, denoted $\mathsf{Erl}(m, \lambda)$. In this case, X can be represented as the sum of iid exponential random variables Y_i. That is, $X = \sum_{i=1}^{m} Y_i$, where the $\{Y_i\}$ are iid exponential variables, each with mean $1/\lambda$; see Example 1.9. Using Algorithm 2.4.1, we can write $Y_i = -\frac{1}{\lambda}\ln U_i$, whence

$$X = -\frac{1}{\lambda}\sum_{i=1}^{m} \ln U_i \ . \tag{2.22}$$

Equation (2.22) suggests the following generation algorithm:

Algorithm 2.4.4: Generation of an Erlang Random Variable

input : Positive integer m and $\lambda > 0$.
output: Random variable $X \sim \mathsf{Erl}(m, \lambda)$.
Generate iid random variables $U_1, \ldots, U_m$ from $\mathsf{U}(0,1)$.
$X \leftarrow -\frac{1}{\lambda}\sum_{i=1}^{m} \ln U_i$
return X

2.4.1.4 Beta Distribution If $X \sim \mathsf{Beta}(\alpha, \beta)$, then its pdf is of the form

$$f(x) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}\, x^{\alpha-1}(1-x)^{\beta-1}, \quad 0 \leqslant x \leqslant 1 \ . \tag{2.23}$$

Both parameters α and β are assumed to be greater than 0. Note that $\mathsf{Beta}(1,1)$ is simply the $\mathsf{U}(0,1)$ distribution.

To sample from the beta distribution, let us consider first the case where either α or β equals 1. In that case, we can simply use the inverse-transform method. For example, for $\beta = 1$, the $\mathsf{Beta}(\alpha, 1)$ pdf is

$$f(x) = \alpha x^{\alpha-1}, \qquad 0 \leqslant x \leqslant 1 \ ,$$

and the corresponding cdf becomes

$$F(x) = x^\alpha, \qquad 0 \leqslant x \leqslant 1 \ .$$

Thus a random variable X can be generated from this distribution by drawing $U \sim \mathsf{U}(0,1)$ and returning $X = U^{1/\alpha}$.

A general procedure for generating a $\mathsf{Beta}(\alpha, \beta)$ random variable is based on the fact that if $Y_1 \sim \mathsf{Gamma}(\alpha, 1)$, $Y_2 \sim \mathsf{Gamma}(\beta, 1)$, and Y_1 and Y_2 are independent, then

$$X = \frac{Y_1}{Y_1 + Y_2}$$

is distributed $\mathsf{Beta}(\alpha, \beta)$. The reader is encouraged to prove this assertion (see Problem 2.18). The corresponding algorithm is as follows:

Algorithm 2.4.5: Generation of a Beta Random Variable

input : $\alpha, \beta > 0$
output: Random variable $X \sim \mathsf{Beta}(\alpha, \beta)$.
1 Generate independently $Y_1 \sim \mathsf{Gamma}(\alpha, 1)$ and $Y_2 \sim \mathsf{Gamma}(\beta, 1)$.
2 $X \leftarrow Y_1/(Y_1 + Y_2)$
3 **return** X

For integer $\alpha = m$ and $\beta = n$, another method may be used, based on the theory of order statistics. Let $U_1, \ldots, U_{m+n-1}$ be independent random variables from $\mathsf{U}(0,1)$. Then the m-th order statistic, $U_{(m)}$, has a $\mathsf{Beta}(m, n)$ distribution. This gives the following algorithm.

Algorithm 2.4.6: Generation of a Beta Random Variable with Integer Parameters $\boldsymbol{\alpha} = m$ and $\boldsymbol{\beta} = n$

input : Positive integers m and n.
output: Random variable $X \sim \mathsf{Beta}(m, n)$.
1 Generate $m + n - 1$ iid random variables $U_1, \ldots, U_{m+n-1}$ from $\mathsf{U}(0,1)$.
2 $X \leftarrow U_{(m)}$ `// m-th order statistic`
3 **return** X

It can be shown that the total number of comparisons needed to find $U_{(m)}$ is $(m/2)(m + 2n - 1)$, so that this procedure loses efficiency for large m and n.

2.4.2 Generating Discrete Random Variables

2.4.2.1 Bernoulli Distribution If $X \sim \mathsf{Ber}(p)$, its pdf is of the form

$$f(x) = p^x (1-p)^{1-x}, \qquad x = 0, 1 \ , \tag{2.24}$$

where p is the success probability. Applying the inverse-transform method, we can readily obtain the following generation algorithm:

Algorithm 2.4.7: Generation of a Bernoulli Random Variable

input : $p \in (0,1)$
output: Random variable $X \sim \mathsf{Ber}(p)$.
Generate $U \sim \mathsf{U}(0,1)$.
if $U \leqslant p$ **then**
  $X \leftarrow 1$
else
  $X \leftarrow 0$
return X

In Figure 2.6, three typical outcomes (realizations) are given for 100 independent Bernoulli random variables, each with success parameter $p = 0.5$.

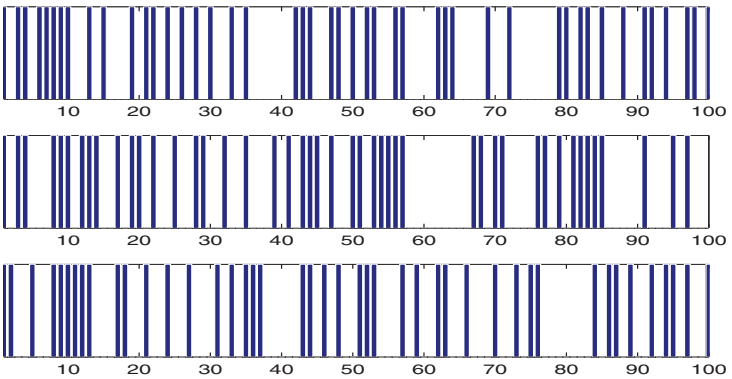

Figure 2.6: Results of three experiments with 100 independent Bernoulli trials, each with $p = 0.5$. The dark bars indicate where a success appears.

2.4.2.2 Binomial Distribution If $X \sim \mathsf{Bin}(n,p)$ then its pdf is of the form

$$f(x) = \binom{n}{x} p^x (1-p)^{n-x}, \quad x = 0, 1, \ldots, n \; . \tag{2.25}$$

Recall that a binomial random variable X can be viewed as the total number of successes in n independent Bernoulli experiments, each with success probability p; see Example 1.1. Denoting the result of the i-th trial by $X_i = 1$ (success) or $X_i = 0$ (failure), we can write $X = X_1 + \cdots + X_n$, with the $\{X_i\}$ being iid $\mathsf{Ber}(p)$ random variables. The simplest generation algorithm can thus be written as follows:

Algorithm 2.4.8: Generation of a Binomial Random Variable

input : Positive integer n and $p \in (0,1)$.
output: Random variable $X \sim \mathsf{Bin}(n,p)$.
Generate iid random variables $X_1, \ldots, X_n$ from $\mathsf{Ber}(p)$.
$X \leftarrow \sum_{i=1}^{n} X_i$
return X

Since the execution time of Algorithm 2.4.8 is proportional to n, we may be motivated to use alternative methods for large n. For example, we could consider the normal distribution as an approximation to the binomial. In particular, by the central limit theorem, as n increases, the distribution of X is close to that of $Y \sim \mathsf{N}(np, np(1-p))$; see (1.26). In fact, the cdf of $\mathsf{N}(np - 1/2, np(1-p))$ approximates the cdf of X even better. This is called the *continuity correction*.

Thus, to obtain a binomial random variable, we could generate Y from $\mathsf{N}(np - 1/2, np(1-p))$ and truncate to the nearest nonnegative integer. Equivalently, we could generate $Z \sim \mathsf{N}(0,1)$ and set

$$\max\left\{0, \left\lfloor np + \frac{1}{2} + Z\sqrt{np(1-p)} \right\rfloor\right\} \tag{2.26}$$

as an approximate sample from the $\mathsf{Bin}(n,p)$ distribution. Here $\lfloor \alpha \rfloor$ denotes the integer part of α. One should consider using the normal approximation for $np > 10$ with $p \geqslant \frac{1}{2}$, and for $n(1-p) > 10$ with $p < \frac{1}{2}$.

2.4.2.3 Geometric Distribution If $X \sim \mathsf{G}(p)$, then its pdf is of the form

$$f(x) = p\,(1-p)^{x-1}, \qquad x = 1, 2 \ldots \ . \tag{2.27}$$

The random variable X can be interpreted as the number of trials required until the first success occurs in a series of independent Bernoulli trials with success parameter p. Note that $\mathbb{P}(X > m) = (1-p)^m$.

We now present an algorithm based on the relationship between the exponential and geometric distributions. Let $Y \sim \mathsf{Exp}(\lambda)$, with λ such that $1 - p = \mathrm{e}^{-\lambda}$. Then $X = \lfloor Y \rfloor + 1$ has a $\mathsf{G}(p)$ distribution. This is because

$$\mathbb{P}(X > x) = \mathbb{P}(\lfloor Y \rfloor > x - 1) = \mathbb{P}(Y \geqslant x) = \mathrm{e}^{-\lambda x} = (1-p)^x \ .$$

Hence, to generate a random variable from $\mathsf{G}(p)$, we first generate a random variable from the exponential distribution with $\lambda = -\ln(1-p)$, truncate the obtained value to the nearest integer, and add 1.

Algorithm 2.4.9: Generation of a Geometric Random Variable

input : $p \in (0,1)$
output: Random variable $X \sim \mathsf{G}(p)$.
1 Generate $Y \sim \mathsf{Exp}(-\ln(1-p))$.
2 $X \leftarrow 1 + \lfloor Y \rfloor$
3 **return** X

2.4.2.4 Poisson Distribution If $X \sim \mathsf{Poi}(\lambda)$, its pdf is of the form

$$f(n) = \frac{\mathrm{e}^{-\lambda}\,\lambda^n}{n!} \ , \qquad n = 0, 1, \ldots \ , \tag{2.28}$$

where λ is the *rate* parameter. There is an intimate relationship between Poisson and exponential random variables, highlighted by the properties of the Poisson process; see Section 1.12. In particular, a Poisson random variable X can be interpreted as the maximal number of iid exponential variables (with parameter λ)

whose sum does not exceed 1. That is,

$$X = \max\left\{ n : \sum_{j=1}^{n} Y_j \leqslant 1 \right\} , \tag{2.29}$$

where the $\{Y_j\}$ are independent and $\mathsf{Exp}(\lambda)$ distributed. Since $Y_j = -\frac{1}{\lambda}\ln U_j$, with $U_j \sim \mathsf{U}(0,1)$, we can rewrite (2.29) as

$$\begin{aligned} X &= \max\left\{ n : \sum_{j=1}^{n} -\ln U_j \leqslant \lambda \right\} \\ &= \max\left\{ n : \ln\left(\prod_{j=1}^{n} U_j\right) \geqslant -\lambda \right\} \\ &= \max\left\{ n : \prod_{j=1}^{n} U_j \geqslant \mathrm{e}^{-\lambda} \right\} . \end{aligned} \tag{2.30}$$

This leads to the following algorithm:

Algorithm 2.4.10: Generation of a Poisson Random Variable

input : $\lambda > 0$
output: Random variable $X \sim \mathsf{Poi}(\lambda)$.
Set $n \leftarrow 0$ and $a \leftarrow 1$.
while $a \geqslant \mathrm{e}^{-\lambda}$ **do**
  Generate $U \sim \mathsf{U}(0,1)$.
  $a \leftarrow a\,U$
  $n \leftarrow n + 1$
$X \leftarrow n - 1$
return X

It is readily seen that for large λ, this algorithm becomes slow ($\mathrm{e}^{-\lambda}$ is small for large λ, and more random numbers, U_j, are required to satisfy $\prod_{j=1}^{n} U_j < \mathrm{e}^{-\lambda}$). Alternative approaches can be found in [2] and [7].

2.5 RANDOM VECTOR GENERATION

Say we need to generate a random vector $\mathbf{X} = (X_1, \ldots, X_n)$ from a given n-dimensional distribution with pdf $f(\mathbf{x})$ and cdf $F(\mathbf{x})$. When the components $X_1, \ldots, X_n$ are *independent*, the situation is easy: we simply apply the inverse-transform method or another generation method of our choice to each component individually.

EXAMPLE 2.9

We want to generate uniform random vectors $\mathbf{X} = (X_1, \ldots, X_n)$ from the n-dimensional rectangle $D = \{(x_1, \ldots, x_n) : a_i \leqslant x_i \leqslant b_i,\ i = 1, \ldots, n\}$. It is

clear that the components of $\mathbf{X}$ are independent and uniformly distributed: $X_i \sim U[a_i, b_i]$, $i = 1, \ldots, n$. Applying the inverse-transform method to X_i, we can therefore write $X_i = a_i + (b_i - a_i)\, U_i$, $i = 1, \ldots, n$, where $U_1, \ldots, U_n$ are iid from $\mathsf{U}(0, 1)$.

For *dependent* random variables $X_1, \ldots, X_n$, we can represent the joint pdf $f(\mathbf{x})$, using the product rule (1.4), as

$$f(x_1, \ldots, x_n) = f_1(x_1)\, f_2(x_2 \,|\, x_1) \cdots f_n(x_n \,|\, x_1, \ldots, x_{n-1}) \;, \tag{2.31}$$

where $f_1(x_1)$ is the marginal pdf of X_1 and $f_k(x_k \,|\, x_1, \ldots, x_{k-1})$ is the conditional pdf of X_k given $X_1 = x_1, X_2 = x_2, \ldots, X_{k-1} = x_{k-1}$. Thus one way to generate $\mathbf{X}$ is to first generate X_1, then, given $X_1 = x_1$, to generate X_2 from $f_2(x_2 \,|\, x_1)$, and so on, until we generate X_n from $f_n(x_n \,|\, x_1, \ldots, x_{n-1})$.

The applicability of this approach depends, of course, on knowledge of the conditional distributions. In certain models, such as Markov models, this knowledge is easily obtainable.

2.5.1 Vector Acceptance–Rejection Method

The acceptance–rejection Algorithm 2.3.6 is directly applicable to the multidimensional case. We need only to bear in mind that the random variable X (see Line 3 of Algorithm 2.3.6) becomes an n-dimensional random vector $\mathbf{X}$. Consequently, we need a convenient way of generating $\mathbf{X}$ from the multidimensional proposal pdf $g(\mathbf{x})$, for example, by using the vector inverse-transform method. The next example demonstrates the vector version of the acceptance–rejection method.

■ **EXAMPLE 2.10**

We want to generate a random vector $\mathbf{Z}$ that is uniformly distributed over an irregular n-dimensional region G (see Figure 2.7). The algorithm is straightforward:

1. *Generate a random vector,* $\mathbf{X}$*, uniformly distributed in* W*, where* W *is a regular region (multidimensional hypercube, hyperrectangle, hypersphere, hyperellipsoid, etc.).*

2. *If* $\mathbf{X} \in G$*, accept* $\mathbf{Z} = \mathbf{X}$ *as the random vector uniformly distributed over* G*; otherwise, return to Step 1.*

As a special case, let G be the n-dimensional unit ball, that is, $G = \{\mathbf{x} : \sum_i x_i^2 \leqslant 1\}$, and let W be the n-dimensional hypercube $\{-1 \leqslant x_i \leqslant 1\}_{i=1}^n$. To generate a random vector that is uniformly distributed over the interior of the n-dimensional unit ball, we generate a random vector $\mathbf{X}$ that is uniformly distributed over W and then accept or reject it, depending on whether it falls inside or outside the n-dimensional ball. The corresponding algorithm is as follows:

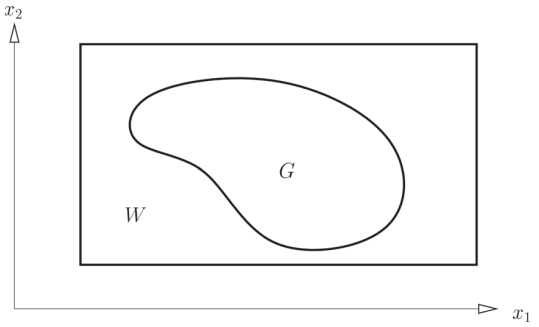

Figure 2.7: The vector acceptance–rejection method.

Algorithm 2.5.1: Generation of a Random Vector Uniformly Distributed Within the n-Dimensional Unit Ball

output: Random vector $\mathbf{X}$ in the unit ball.

$R \leftarrow \infty$
while $R > 1$ **do**
  Generate $U_1, \ldots, U_n$ as iid random variables from $\mathsf{U}(0,1)$.
  $X_1 \leftarrow 1 - 2U_1, \ldots, X_n \leftarrow 1 - 2U_n$
  $R \leftarrow \sum_{i=1}^{n} X_i^2$
$\mathbf{X} \leftarrow (X_1, \ldots, X_n)$
return $\mathbf{X}$

Remark 2.5.1 To generate a random vector that is uniformly distributed over the *surface* of an n-dimensional unit ball — in other words, uniformly over the unit sphere $\{\mathbf{x} : \sum_i x_i^2 = 1\}$ — we need only to scale the vector $\mathbf{X}$ such that it has unit length. That is, we return $\mathbf{Z} = \mathbf{X}/\sqrt{R}$ instead of $\mathbf{X}$.

The efficiency of the vector acceptance–rejection method is equal to the ratio

$$\frac{1}{C} = \frac{\text{volume of the hyperball}}{\text{volume of the hypercube}} = \frac{1}{n\,2^{n-1}} \frac{\pi^{n/2}}{\Gamma(n/2)} ,$$

where the volumes of the ball and cube are $\frac{\pi^{n/2}}{(n/2)\Gamma(n/2)}$ and 2^n, respectively. Note that for even $n\,(n = 2m)$ we have

$$\frac{1}{C} = \frac{\pi^m}{m!\,2^{2m}} = \frac{1}{m!} \left(\frac{\pi}{2}\right)^m 2^{-m} \to 0 \quad \text{as} \quad m \to \infty.$$

In other words, the acceptance–rejection method grows inefficient in n, and is asymptotically useless.

2.5.2 Generating Variables from a Multinormal Distribution

The key to generating a multivariate normal (or simply multinormal) random vector $\mathbf{Z} \sim \mathsf{N}(\boldsymbol{\mu}, \Sigma)$ is to write it as $\mathbf{Z} = \boldsymbol{\mu} + B\mathbf{X}$, where B a matrix such that $BB^\top = \Sigma$,

and $\mathbf{X}$ is a vector of iid $\mathsf{N}(0,1)$ random variables; see Section 1.10. Note that $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_n)$ is the mean vector and Σ is the $(n \times n)$ covariance matrix of $\mathbf{Z}$. For any covariance matrix Σ, such a matrix B can always be found efficiently using the Cholesky square root method; see Section A.1 of the Appendix.

The next algorithm describes the generation of a $\mathsf{N}(\boldsymbol{\mu}, \Sigma)$ distributed random vector $\mathbf{Z}$:

Algorithm 2.5.2: Generation of Multinormal Vectors

input : Mean vector $\boldsymbol{\mu}$ and covariance matrix Σ.
output: Random vector $\mathbf{Z} \sim \mathsf{N}(\boldsymbol{\mu}, \Sigma)$.
1 Generate $X_1, \ldots, X_n$ as iid variables from $\mathsf{N}(0,1)$.
2 Derive the lower Cholesky decomposition $\Sigma = BB^\top$.
3 Set $\mathbf{Z} \leftarrow \boldsymbol{\mu} + B\mathbf{X}$.
4 **return Z**

2.5.3 Generating Uniform Random Vectors over a Simplex

Consider the n-dimensional simplex,

$$\mathscr{Y} = \left\{ \mathbf{y} : y_i \geqslant 0, \quad i = 1, \ldots, n, \ \sum_{i=1}^{n} y_i \leqslant 1 \right\} . \tag{2.32}$$

$\mathscr{Y}$ is a simplex on the points $\mathbf{0}, \mathbf{e}_1, \ldots, \mathbf{e}_n$, where $\mathbf{0}$ is the zero vector and $\mathbf{e}_i$ is the i-th unit vector in $\mathbb{R}^n$, $i = 1, \ldots, n$. Let $\mathscr{X}$ be a second n-dimensional simplex:

$$\mathscr{X} = \{ \mathbf{x} : x_i \geqslant 0, \quad i = 1, \ldots, n, \ x_1 \leqslant x_2 \leqslant \cdots \leqslant x_n \leqslant 1 \} .$$

$\mathscr{X}$ is a simplex on the points $\mathbf{0}, \mathbf{e}_n, \mathbf{e}_n + \mathbf{e}_{n-1}, \ldots, \mathbf{1}$, where $\mathbf{1}$ is the sum of all unit vectors (a vector of 1s). Figure 2.8 illustrates the two-dimensional case.

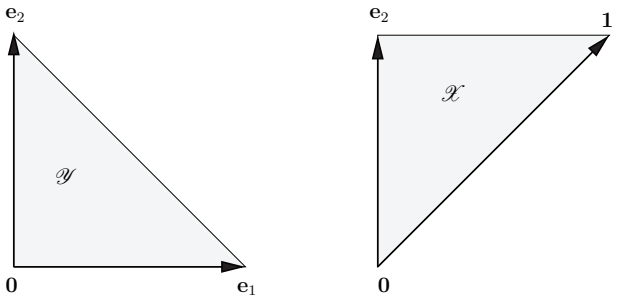

Figure 2.8: Simplexes $\mathscr{Y}$ and $\mathscr{X}$.

Simplex $\mathscr{Y}$ can be obtained from simplex $\mathscr{X}$ by the linear transformation $\mathbf{y} = A\mathbf{x}$ with

$$A = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ -1 & 1 & \cdots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & -1 & 1 \end{pmatrix} .$$

Now, drawing a vector $\mathbf{X} = (X_1, \ldots, X_n)$ according to the uniform distribution on $\mathscr{X}$ is easy: simply take X_i to be the i-th order statistic of iid random variables $U_1, \ldots, U_n$ from $\mathsf{U}(0,1)$. Since a linear transformation preserves uniformity, applying matrix A to $\mathbf{X}$ yields a vector $\mathbf{Y}$ that is uniformly distributed on $\mathscr{Y}$.

Algorithm 2.5.3: Generating a Vector over a Unit Simplex $\mathscr{Y}$

output: Random vector $\mathbf{Y}$ uniformly distributed over the simplex $\mathscr{Y}$.
1 Generate n independent random variables $U_1, \ldots, U_n$ from $\mathsf{U}(0,1)$.
2 Sort $U_1, \ldots, U_n$ into the order statistics $U_{(1)}, \ldots, U_{(n)}$.
3 $Y_1 \leftarrow U_{(1)}$
4 **for** $i = 2$ **to** n **do** $Y_i = U_{(i)} - U_{(i-1)}$
5 $\mathbf{Y} \leftarrow (Y_1, \ldots, Y_n)$
6 **return** $\mathbf{Y}$

If we define $Y_{n+1} = 1 - \sum_{i=1}^{n} Y_i = 1 - U_{(n)}$, then the resulting $(n+1)$-dimensional vector $(Y_1, \ldots, Y_{n+1})$ will be uniformly distributed over the set

$$\mathscr{F} = \left\{ \mathbf{y} : y_i \geqslant 0, \quad i = 1, \ldots, n+1, \ \sum_{i=1}^{n+1} y_i = 1 \right\} ,$$

that is, over the dominant face of the simplex defined by the points $\mathbf{0}, \mathbf{e}_1, \ldots, \mathbf{e}_{n+1}$.

Finally, in order to generate random vectors uniformly distributed over an n-dimensional simplex defined by arbitrary vertices, say $\mathbf{z}_0, \mathbf{z}_1, \ldots, \mathbf{z}_n$, we simply generate $\mathbf{Y}$ uniformly on $\mathscr{Y}$ and apply the linear transformation

$$\mathbf{Z} = C\mathbf{Y} + \mathbf{z}_0 ,$$

where C is the matrix whose columns are $\mathbf{z}_1 - \mathbf{z}_0, \ldots, \mathbf{z}_n - \mathbf{z}_0$.

2.5.4 Generating Random Vectors Uniformly Distributed over a Unit Hyperball and Hypersphere

Algorithm 2.5.1 and Remark 2.5.1 explain how, using the multidimensional acceptance–rejection method, one can generate random vectors that are uniformly distributed over an n-dimensional unit hyperball (or simply n-ball). By simply dividing each vector by its length, one obtains random vectors that are uniformly distributed over the *surface* of the n-ball, that is, the n-sphere. The main advantage of the acceptance–rejection method is its simplicity. Its main disadvantage is that the number of trials needed to generate points inside the n-ball increases explosively with n. For this reason, it can be recommended only for low dimensions ($n \leqslant 5$). An alternative algorithm is based on the following result:

Theorem 2.5.1 *Let $X_1, \ldots, X_n$ be iid random variables from $\mathsf{N}(0,1)$, and let $||\mathbf{X}|| = (\sum_{i=1}^{n} X_i^2)^{\frac{1}{2}}$. Then the vector*

$$\mathbf{Y} = \left(\frac{X_1}{||\mathbf{X}||} , \ldots , \frac{X_n}{||\mathbf{X}||} \right) \tag{2.33}$$

is distributed uniformly over the n-sphere $\{\mathbf{y} : ||\mathbf{y}|| = 1\}$.

Proof: Note that $\mathbf{Y}$ is simply the projection of $\mathbf{X} = (X_1, \ldots, X_n)$ onto the n-sphere. The fact that $\mathbf{Y}$ is uniformly distributed follows immediately from the fact that the pdf of $\mathbf{X}$ is spherically symmetrical: $f_{\mathbf{X}}(\mathbf{x}) = c\, \mathrm{e}^{-||\mathbf{x}||^2/2}$. □

To obtain uniform random variables within the n-ball, we simply multiply the vector $\mathbf{Y}$ by $U^{1/n}$, where $U \sim \mathsf{U}(0,1)$. To see this, note that for a random vector $\mathbf{Z} = (Z_1, \ldots, Z_n)$ that is uniformly distributed over the n-ball, the radius $R = ||\mathbf{Z}||$ satisfies $\mathbb{P}(R \leqslant r) = r^n$. Hence, by the inverse-transform method, we can write $R = U^{1/n}$. This motivates the following alternative:

Algorithm 2.5.4: Generating Uniform Random Vectors inside the n-Ball

output: Random vector $\mathbf{Z}$ uniformly distributed within the n-ball.

1 Generate a random vector $\mathbf{X} = (X_1, \ldots, X_n)$ with iid $\mathsf{N}(0,1)$ components.
2 Generate $R = U^{1/n}$, with $U \sim \mathsf{U}(0,1)$.
3 $\mathbf{Z} \leftarrow R\,\mathbf{X}/||\mathbf{X}||$
4 **return** $\mathbf{Z}$

2.5.5 Generating Random Vectors Uniformly Distributed inside a Hyperellipsoid

The equation for a hyperellipsoid, centered at the origin, can be written as

$$\mathbf{x}^\top \Sigma \mathbf{x} = r^2 \;, \tag{2.34}$$

where Σ is a positive definite and symmetric $(n \times n)$ matrix ($\mathbf{x}$ is interpreted as a column vector). The special case where $\Sigma = I$ (identity matrix) corresponds to a hypersphere of radius r. Since Σ is positive definite and symmetric, there exists a unique lower triangular matrix B such that $\Sigma = BB^\top$; see (1.25). We can thus view the set $\mathscr{X} = \{\mathbf{x} : \mathbf{x}^\top \Sigma \mathbf{x} \leqslant r^2\}$ as a linear transformation $\mathbf{y} = B^\top \mathbf{x}$ of the n-dimensional ball $\mathscr{Y} = \{\mathbf{y} : \mathbf{y}^\top \mathbf{y} \leqslant r^2\}$. Since linear transformations preserve uniformity, if the vector $\mathbf{Y}$ is uniformly distributed over the interior of an n-dimensional sphere of radius r, then the vector $\mathbf{X} = (B^\top)^{-1}\mathbf{Y}$ is uniformly distributed over the interior of a hyperellipsoid (see (2.34)). The corresponding generation algorithm is given below.

Algorithm 2.5.5: Generating Uniform Random Vectors in a Hyperellipsoid

input : Σ, $r > 0$

output: Random vector $\mathbf{X}$ uniformly distributed within the hyperellipsoid.

1 Generate $\mathbf{Y} = (Y_1, \ldots, Y_n)$ uniformly distributed within the n-ball of radius r.
2 Calculate the (lower Cholesky) matrix B, satisfying $\Sigma = BB^\top$.
3 $\mathbf{X} \leftarrow (B^\top)^{-1}\mathbf{Y}$
4 **return** $\mathbf{X}$

2.6 GENERATING POISSON PROCESSES

This section treats the generation of Poisson processes. Recall from Section 1.12 that there are two different (but equivalent) characterizations of a Poisson process

$\{N_t, t \geqslant 0\}$. In the first (see Definition 1.12.1), the process is interpreted as a counting measure, where N_t counts the number of arrivals in $[0, t]$. The second characterization is that the interarrival times $\{A_i\}$ of $\{N_t, t \geqslant 0\}$ form a *renewal process*, that is, a sequence of iid random variables. In this case, the interarrival times have an $\mathsf{Exp}(\lambda)$ distribution, and we can write $A_i = -\frac{1}{\lambda} \ln U_i$, where the $\{U_i\}$ are iid $\mathsf{U}(0, 1)$ distributed. Using the second characterization, we can generate the arrival times $T_i = A_1 + \cdots + A_i$ during the interval $[0, T]$ as follows:

Algorithm 2.6.1: Generating a Homogeneous Poisson Process

input : Final time T, rate $\lambda > 0$.
output: Number of arrivals N and arrival times $T_1, \ldots, T_N$.
Set $T_0 \leftarrow 0$ and $N \leftarrow 0$.
while $T_N < T$ **do**
  Generate $U \sim \mathsf{U}(0, 1)$.
  $T_{N+1} \leftarrow T_N - \frac{1}{\lambda} \ln U$
  $N \leftarrow N + 1$
return $N, T_1, \ldots, T_N$

The first characterization of a Poisson process, that is, as a random counting measure, provides an alternative way of generating such processes, which works also in the multidimensional case. In particular (see the end of Section 1.12), the following procedure can be used to generate a homogeneous Poisson process with rate λ on any set A with "volume" $|A|$:

Algorithm 2.6.2: Generating an n-Dimensional Poisson Process

input : Rate $\lambda > 0$.
output: Number of points N and positions $\mathbf{X}_1, \ldots, \mathbf{X}_N$.
Generate a Poisson random variable $N \sim \mathsf{Poi}(\lambda\,|A|)$.
Draw N points $\mathbf{X}_1, \ldots, \mathbf{X}_N$ independently and uniformly in A.
return $\mathbf{X}_1, \ldots, \mathbf{X}_N$

A *nonhomogeneous Poisson process* is a counting process $N = \{N_t, t \geqslant 0\}$ for which the number of points in nonoverlapping intervals are independent — similar to the ordinary Poisson process — but the rate at which points arrive is *time dependent*. If $\lambda(t)$ denotes the rate at time t, the number of points in any interval (b, c) has a Poisson distribution with mean $\int_b^c \lambda(t)\,\mathrm{d}t$.

Figure 2.9 illustrates a way to construct such processes. We first generate a two-dimensional homogeneous Poisson process on the strip $\{(t, x), t \geqslant 0, 0 \leqslant x \leqslant \lambda\}$, with constant rate $\lambda = \max \lambda(t)$, and then simply project all points below the graph of $\lambda(t)$ onto the t-axis.

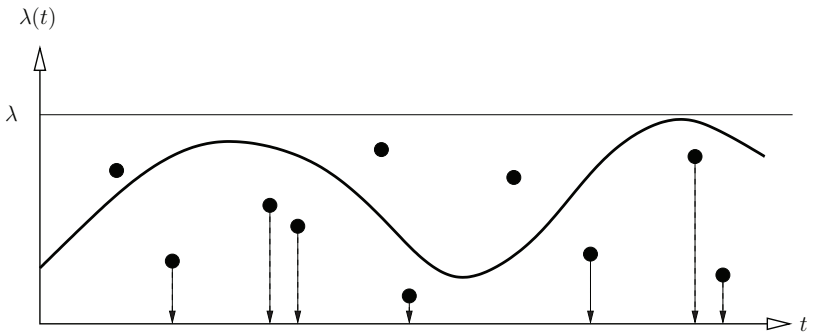

Figure 2.9: Constructing a nonhomogeneous Poisson process.

Note that the points of the two-dimensional Poisson process can be viewed as having a time and space dimension. The arrival epochs form a one-dimensional Poisson process with rate λ, and the positions are uniform on the interval $[0, \lambda]$. This suggests the following alternative procedure for generating nonhomogeneous Poisson processes: each arrival epoch of the one-dimensional homogeneous Poisson process is rejected (thinned) with probability $1 - \frac{\lambda(T_n)}{\lambda}$, where T_n is the arrival time of the n-th event. The surviving epochs define the desired nonhomogeneous Poisson process.

Algorithm 2.6.3: Generating a Nonhomogeneous Poisson Process

input : Final time T and rate function $\lambda(t)$, with $\max \lambda(t) = \lambda$.
output: Number of arrivals N and arrival times $T_1, T_2, \ldots, T_N$.

$t \leftarrow 0$ and $N \leftarrow 0$
while $t < T$ **do**
  Generate $U \sim \mathsf{U}(0,1)$.
  $t \leftarrow t - \frac{1}{\lambda} \ln U$
  Generate $V \sim \mathsf{U}(0,1)$.
  if $V \leqslant \lambda(t)/\lambda$ **then**
    $T_{N+1} \leftarrow t$
    $N \leftarrow N + 1$
return N and $T_1, \ldots, T_N$

2.7 GENERATING MARKOV CHAINS AND MARKOV JUMP PROCESSES

We now discuss how to simulate a Markov chain $X_0, X_1, X_2, \ldots, X_n$. To generate a Markov chain with initial distribution $\boldsymbol{\pi}^{(0)}$ and transition matrix P, we can use the procedure outlined in Section 2.5 for dependent random variables. That is, first generate X_0 from $\boldsymbol{\pi}^{(0)}$. Then, given $X_0 = x_0$, generate X_1 from the conditional distribution of X_1 given $X_0 = x_0$; in other words, generate X_1 from the x_0-th row of P. Suppose $X_1 = x_1$. Then, generate X_2 from the x_1-st row of P, and so on. The algorithm for a general discrete-state Markov chain with a one-step transition matrix P and an initial distribution vector $\boldsymbol{\pi}^{(0)}$ is as follows:

Algorithm 2.7.1: Generating a Markov Chain

input : Sample size N, initial distribution $\boldsymbol{\pi}^{(0)}$, transition matrix P.
output: Markov chain $X_0, \ldots, X_N$.
1 Draw X_0 from the initial distribution $\boldsymbol{\pi}^{(0)}$.
2 **for** $t = 1$ **to** N **do**
3 Draw X_t from the distribution corresponding to the X_{t-1}-th row of P.
4 **return** $X_0, \ldots, X_N$

■ **EXAMPLE 2.11 Random Walk on the Integers**

Consider the random walk on the integers in Example 1.10. Let $X_0 = 0$ (i.e., we start at 0). Suppose the chain is at some discrete time $t = 0, 1, 2 \ldots$ in state i. Then, in Line 3 of Algorithm 2.7.1, we simply need to draw from a two-point distribution with mass p and q at $i + 1$ and $i - 1$, respectively. In other words, we draw $I_t \sim \mathsf{Ber}(p)$ and set $X_{t+1} = X_t + 2I_t - 1$. Figure 2.10 gives a typical sample path for the case where $p = q = 1/2$.

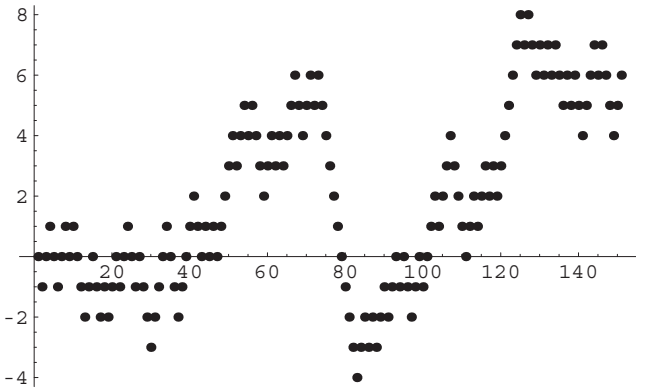

Figure 2.10: Random walk on the integers, with $p = q = 1/2$.

2.7.1 Random Walk on a Graph

As a generalization of Example 2.11, we can associate a random walk with any graph G, whose state space is the vertex set of the graph and whose transition probabilities from i to j are equal to $1/d_i$, where d_i is the degree of i (the number of edges out of i). An important property of such random walks is that they are time-reversible. This can be easily verified from Kolmogorov's criterion (1.39). In other words, there is no systematic "looping". As a consequence, if the graph is connected and if the stationary distribution $\{\pi_i\}$ exists — which is the case when the graph is finite — then the local balance equations hold:

$$\pi_i \, p_{ij} = \pi_j \, p_{ji} \ . \tag{2.35}$$

When $p_{ij} = p_{ji}$ for all i and j, the random walk is said to be *symmetric*. It follows immediately from (2.35) that in this case the equilibrium distribution is uniform over the state space $\mathscr{E}$.

EXAMPLE 2.12 Simple Random Walk on an n-Cube

We want to simulate a random walk over the vertices of the n-dimensional hypercube (or simply n-cube); see Figure 2.11 for the three-dimensional case.

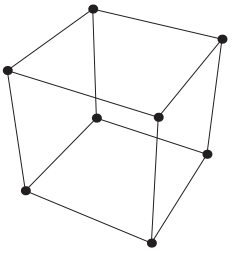

Figure 2.11: At each step, one of the three neighbors of the currently visited vertex is chosen at random.

Note that the vertices of the n-cube are of the form $\mathbf{x} = (x_1, \ldots, x_n)$, with x_i either 0 or 1. The set of all 2^n of these vertices is denoted $\{0,1\}^n$. We generate a random walk $\{X_t, t = 0, 1, 2, \ldots\}$ on $\{0,1\}^n$ as follows: Let the initial state X_0 be arbitrary, say $X_0 = (0, \ldots, 0)$. Given $X_t = (x_{t1}, \ldots, x_{tn})$, choose randomly a coordinate J according to the discrete uniform distribution on the set $\{1, \ldots, n\}$. If j is the outcome, then replace x_{jn} with $1 - x_{jn}$. By doing so, we obtain at stage $t + 1$,

$$X_{t+1} = (x_{t1}, \ldots, 1 - x_{tj}, x_{t(j+1)}, \ldots, x_{tn}) \,,$$

and so on.

2.7.2 Generating Markov Jump Processes

The generation of Markov jump processes is quite similar to the generation of Markov chains above. Suppose $X = \{X_t, t \geqslant 0\}$ is a Markov jump process with transition rates $\{q_{ij}\}$. From Section 1.13.5, recall that the Markov jump process jumps from one state to another according to a Markov chain $Y = \{Y_n\}$ (the jump chain), and the time spent in each state i is exponentially distributed with a parameter that may depend on i. The one-step transition matrix, say K, of Y and the parameters $\{q_i\}$ of the exponential holding times can be found directly from the $\{q_{ij}\}$. Namely, $q_i = \sum_j q_{ij}$ (the sum of the transition rates out of i), and $K(i,j) = q_{ij}/q_i$ for $i \neq j$ (thus, the probabilities are simply proportional to the rates). Note that $K(i,i) = 0$. Defining the holding times as $A_1, A_2, \ldots$ and the jump times as $T_1, T_2, \ldots$, we can write the algorithm now as follows:

Algorithm 2.7.2: Generating a Markov Jump Process

input : Final time T, initial distribution $\boldsymbol{\pi}^{(0)}$, transition rates $\{q_{ij}\}$.
output: Number of jumps N, jump times $T_1, \ldots, T_N$ and jump states $Y_0, \ldots, Y_N$.

Draw Y_0 from the initial distribution $\boldsymbol{\pi}^{(0)}$.
Initialize $X_0 \leftarrow Y_0$, $T_0 \leftarrow 0$, and $N \leftarrow 0$.
while $T_N < T$ **do**
    Draw A_{N+1} from $\mathsf{Exp}(q_{Y_N})$
    $T_{N+1} \leftarrow T_N + A_{N+1}$
    Set $X_t \leftarrow Y_N$ for $T_N \leqslant t < T_{N+1}$
    Draw Y_{N+1} from the distribution corresponding to the Y_N-th row of K.
    $N \leftarrow N + 1$
return $N, T_1, \ldots, T_N, Y_0, Y_1, \ldots, Y_N$

2.8 GENERATING GAUSSIAN PROCESSES

Recall from Section 1.14 that, in a Gaussian process $\{X_t, t \in \mathscr{T}\}$, each subvector $(X_{t_1}, \ldots, X_{t_n})^\top$ has a multinormal distribution, $\mathsf{N}(\boldsymbol{\mu}, \Sigma)$, for some expectation vector $\boldsymbol{\mu}$, and covariance matrix Σ, depending on $t_1, \ldots, t_n$. In particular, if the Gaussian process has expectation function μ_t, $t \in \mathscr{T}$ and covariance function $\Sigma_{s,t}, s, t \in \mathscr{T}$, then $\boldsymbol{\mu} = (\mu_{t_1}, \ldots, \mu_{t_n})^\top$ and $\Sigma_{ij} = \Sigma_{t_i,t_j}$, $i, j = 1, \ldots, n$. Consequently, Algorithm 2.5.2 can be used to generate realizations of a Gaussian process at specified times (indexes) $t_1, \ldots, t_n$.

Although it is always possible to find a Cholesky decomposition $\Sigma = BB^\top$ (see Section A.1 of the Appendix), this procedure requires in general $O(n^3)$ floating point operations (for an $n \times n$ matrix). As a result, the generation of high-dimensional Gaussian vectors becomes very time-consuming for large n, unless the process has extra structure.

■ EXAMPLE 2.13 Wiener Process Generation

In Example 1.14, the Wiener process was defined as a zero-mean Gaussian process $\{X_t, t \geqslant 0\}$ with covariance function $\Sigma_{s,t} = s$ for $0 \leqslant s \leqslant t$.

It is obviously not possible to generate the complete sample path, as an infinite amount of variables would have to be generated. However, we could use Algorithm 2.5.2 to generate outcomes of the normal random vector $\mathbf{X} = (X_{t_1}, \ldots, X_{t_n})^\top$ for times $t_1 < \cdots < t_n$. The covariance matrix Σ of $\mathbf{X}$ is in this case

$$\Sigma = \begin{pmatrix} t_1 & t_1 & t_1 & \ldots & t_1 \\ t_1 & t_2 & t_2 & \ldots & t_2 \\ t_1 & t_2 & t_3 & \ldots & t_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ t_1 & t_2 & t_3 & \ldots & t_n \end{pmatrix}.$$

We can easily verify (multiply B with $B^\top$) that the lower Cholesky matrix B is given by

$$B = \begin{pmatrix} \sqrt{t_1} & 0 & 0 & \dots & 0 \\ \sqrt{t_1} & \sqrt{t_2 - t_1} & 0 & \dots & 0 \\ \sqrt{t_1} & \sqrt{t_2 - t_1} & \sqrt{t_3 - t_2} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \sqrt{t_1} & \sqrt{t_2 - t_1} & \sqrt{t_3 - t_2} & \dots & \sqrt{t_n - t_{n-1}} \end{pmatrix}.$$

This brings us to the following simulation algorithm:

Algorithm 2.8.1: Generating a Wiener Process

input : Times $0 = t_0 < t_1 < t_2 < \cdots < t_n$.
output: Wiener process random variables at times $t_0, \dots, t_n$.

$X_0 \leftarrow 0$
for $k = 1$ **to** n **do**
  Draw $Z \sim \mathsf{N}(0,1)$
  $X_{t_k} \leftarrow X_{t_{k-1}} + \sqrt{t_k - t_{k-1}}\, Z$
return $X_{t_0}, \dots, X_{t_n}$

Figure 1.8 on Page 28 gives a typical realization of a Wiener process on the interval $[0,1]$, evaluated at grid points $t_n = n\,10^{-5}, n = 0, 1, \dots, 10^5$.

By utilizing the properties of the Wiener process (see Example 1.14), we can justify Algorithm 2.8.1 in a more direct way. Specifically, X_{t_1} has a $\mathsf{N}(0, t_1)$ distribution and can therefore be generated as $\sqrt{t_1} Z_1$, where Z_1 has a standard normal distribution. Moreover, for each $k = 2, 3, \dots, n$, the random variable X_{t_k} is equal to $X_{t_{k-1}} + U_t$, where the increment U_t is independent of $X_{t_{k-1}}$ and has a $\mathsf{N}(t_k - t_{k-1})$ distribution, and can be generated as $\sqrt{t_k - t_{k-1}} Z_k$.

2.9 GENERATING DIFFUSION PROCESSES

The Wiener process, denoted in this section by $\{W_t, t \geqslant 0\}$, plays an important role in probability and forms the basis of so-called *diffusion processes*, denoted here by $\{X_t, t \geqslant 0\}$. These are Markov processes with a continuous time parameter and with continuous sample paths, like the Wiener process.

A diffusion process is often specified as the solution of a *stochastic differential equation* (SDE), which is an expression of the form

$$\mathrm{d}X_t = a(X_t, t)\,\mathrm{d}t + b(X_t, t)\,\mathrm{d}W_t\,, \tag{2.36}$$

where $\{W_t, t \geqslant 0\}$ is a Wiener process and $a(x,t)$ and $b(x,t)$ are deterministic functions. The coefficient (function) a is called the *drift*, and b^2 is called the *diffusion* coefficient. When a and b are constants, say $a(x,t) = \mu$ and $b(x,t) = \sigma$, the resulting diffusion process is of the form

$$X_t = \mu\, t + \sigma W_t$$

and is called a *Brownian motion* with drift μ and diffusion coefficient σ^2.

A simple technique for approximately simulating diffusion processes is *Euler's method*; see, for example, [4]. The idea is to replace the SDE with the stochastic difference equation

$$Y_{k+1} = Y_k + a(Y_k, kh)\, h + b(Y_k, kh)\sqrt{h}\, Z_{k+1} \;, \quad k = 0, 1, 2, \ldots \;, \tag{2.37}$$

where $Z_1, Z_2, \ldots$ are independent $\mathsf{N}(0,1)$-distributed random variables. For a small step size h, the process $\{Y_k, k = 0, 1, 2, \ldots\}$ approximates the process $\{X_t, t \geqslant 0\}$ in the sense that $Y_k \approx X_{kh}$, $k = 0, 1, 2, \ldots$.

Algorithm 2.9.1: Euler's Method for SDEs

input : Step size h, sample size N, functions a and b.
output: SDE diffusion process approximation at times $0, h, 2h, \ldots, Nh$.

Generate Y_0 from the distribution of X_0.
for $k = 1$ **to** N **do**
  Draw $Z \sim \mathsf{N}(0,1)$
  $Y_{k+1} \leftarrow Y_k + a(Y_k, kh)\, h + b(Y_k, kh)\sqrt{h}\, Z$
return $Y_0, \ldots, Y_N$

EXAMPLE 2.14 Geometric Brownian Motion

Geometric Brownian motion is often used in financial engineering to model the price of a risky asset; see, for example, [3]. The corresponding SDE is

$$\mathrm{d}X_t = \mu\, X_t\, \mathrm{d}t + \sigma X_t\, \mathrm{d}W_t \;,$$

with initial value X_0. The Euler approximation is

$$Y_{k+1} = Y_k \left(1 + \mu\, h + \sigma\sqrt{h}\, Z_{k+1}\right), \quad k = 0, 1, 2, \ldots \;,$$

where $Z_1, Z_2, \ldots$ are independent standard normal random variables.

Figure 2.12 shows a typical path of the geometric Brownian motion starting at $X_0 = 1$, with parameters $\mu = 1$ and $\sigma = 0.2$, generated via the Euler approximation with step size $h = 10^{-4}$. The dashed line is the graph of $t \mapsto \exp[t(\mu - \sigma^2/2)]$ along which the process fluctuates.

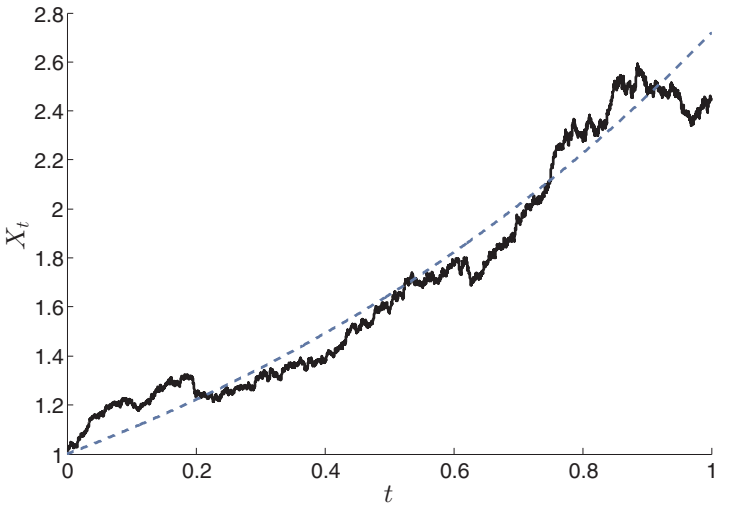

Figure 2.12: Geometric Brownian motion.

More elaborate approximation methods for SDEs can be found, for example, in [4].

2.10 GENERATING RANDOM PERMUTATIONS

Many Monte Carlo algorithms involve generating random permutations, that is, random ordering of the numbers $1, 2, \ldots, n$, for some fixed n. For examples of interesting problems associated with the generation of random permutations, see the traveling salesman problem in Chapter 6, the permanent problem in Chapter 9, and Example 2.15 below.

Suppose that we want to generate each of the $n!$ possible orderings with equal probability. We present two algorithms to achieve this. The first is based on the ordering of a sequence of n uniform random numbers. In the second, we choose the components of the permutation consecutively. The second algorithm is faster than the first.

Algorithm 2.10.1: First Algorithm for Generating Random Permutations

input : Integer $n \geqslant 1$.
output: Random permutation $(X_1, \ldots, X_n)$ of $(1, \ldots, n)$.
1 Generate $U_1, U_2, \ldots, U_n \sim \mathsf{U}(0, 1)$ independently.
2 Arrange these in increasing order.
3 Let $(X_1, \ldots, X_n)$ be the indices of the successive ordered values.
4 **return** $(X_1, \ldots, X_n)$

For example, let $n = 4$ and assume that the generated numbers (U_1, U_2, U_3, U_4) are $(0.7, 0.3, 0.5, 0.4)$. Since $(U_2, U_4, U_3, U_1) = (0.3, 0.4, 0.5, 0.7)$ is the ordered sequence, the resulting permutation is $(2, 4, 3, 1)$. The drawback of this algorithm

is that it requires ordering a sequence of n random numbers, which requires $n \ln n$ comparisons.

As we mentioned, the second algorithm is based on the idea of generating the components of the random permutation one by one. The first component is chosen randomly (with equal probability) from $1, \ldots, n$. The next component is randomly chosen from the remaining numbers, and so on. For example, let $n = 4$. We draw component 1 from the discrete uniform distribution on $\{1, 2, 3, 4\}$. Say we obtain 2. Our permutation is thus of the form $(2, \cdot, \cdot, \cdot)$. We next generate from the three-point uniform distribution on $\{1, 3, 4\}$. Now, say 1 is chosen. Thus our intermediate result for the permutation is $(2, 1, \cdot, \cdot)$. Last, for the third component, we can choose either 3 or 4 with equal probability. Suppose that we draw 4. The resulting permutation is $(2, 1, 4, 3)$. Generating a random variable X from a discrete uniform distribution on $\{x_1, \ldots, x_k\}$ is done efficiently by first generating $I = \lfloor k\,U \rfloor + 1$, with $U \sim \mathsf{U}(0, 1)$ and returning $X = x_I$. Thus we have the following algorithm:

Algorithm 2.10.2: Second Algorithm for Generating Random Permutations

input : Integer $n \geqslant 1$.
output: Random permutation $(X_1, \ldots, X_n)$ of $(1, \ldots, n)$.
Set $\mathscr{P} = \{1, \ldots, n\}$.
for $i = 1$ **to** n **do**
  Generate X_i from the discrete uniform distribution on $\mathscr{P}$.
  Remove X_i from $\mathscr{P}$.
return $(X_1, \ldots, X_n)$

Remark 2.10.1 To further raise the efficiency of the second random permutation algorithm, we can implement it as follows: Let $\mathbf{p} = (p_i, \ldots, p_n)$ be a vector that stores the intermediate results of the algorithm at the i-th step. Initially, let $\mathbf{p} = (1, \ldots, n)$. Draw X_1 by uniformly selecting an index $I \in \{1, \ldots, n\}$, and return $X_1 = p_I$. Then, *swap* X_1 and $p_n = n$. In the second step, draw X_2 by uniformly selecting I from $\{1, \ldots, n-1\}$, return $X_2 = p_I$ and swap it with p_{n-1}, and so on. In this way, the algorithm requires the generation of only n uniform random numbers (for drawing from $\{1, 2, \ldots, k\}, k = n, n-1, \ldots, 2$) and n swap operations.

■ EXAMPLE 2.15 Generating a Random Tour in a Graph

Consider a weighted graph G with n nodes, labeled $1, 2, \ldots, n$. The nodes represent cities, and the edges represent the roads between the cities. The problem is to randomly generate a *tour* that visits all the cities exactly once except for the starting city, which is also the terminating city. Without loss of generality, let us assume that the graph is complete, that is, all cities are connected. We can represent each tour via a permutation of the numbers $1, \ldots, n$. For example, for $n = 4$, the permutation $(1, 3, 2, 4)$ represents the tour $1 \to 3 \to 2 \to 4 \to 1$.

More generally, we represent a tour via a permutation $\mathbf{x} = (x_1, \ldots, x_n)$ with $x_1 = 1$, that is, we assume without loss of generality that we start the tour at city number 1. To generate a random tour uniformly on $\mathscr{X}$, we can

simply apply Algorithm 2.10.2. Note that the number of all possible tours of elements in the set of all possible tours $\mathscr{X}$ is

$$|\mathscr{X}| = (n-1)! \tag{2.38}$$

PROBLEMS

2.1 Apply the inverse-transform method to generate a random variable from the discrete uniform distribution with pdf

$$f(x) = \begin{cases} \frac{1}{n+1}, & x = 0, 1, \ldots, n, \\ 0 & \text{otherwise.} \end{cases}$$

2.2 Explain how to generate from the $\mathsf{Beta}(1, \beta)$ distribution using the inverse-transform method.

2.3 Explain how to generate from the $\mathsf{Weib}(\alpha, \lambda)$ distribution using the inverse-transform method.

2.4 Explain how to generate from the $\mathsf{Pareto}(\alpha, \lambda)$ distribution using the inverse-transform method.

2.5 Many families of distributions are of *location-scale* type. That is, the cdf has the form

$$F(x) = F_0\left(\frac{x-\mu}{\sigma}\right),$$

where μ is called the *location* parameter and σ the *scale* parameter, and F_0 is a fixed cdf that does not depend on μ and σ. The $\mathsf{N}(\mu, \sigma^2)$ family of distributions, where F_0 is the standard normal cdf, is a basic example. Write $F(x; \mu, \sigma)$ for $F(x)$. Let $X \sim F_0$ (i.e., $X \sim F(x; 0, 1)$). Prove that $Y = \mu + \sigma X \sim F(x; \mu, \sigma)$. Hence to sample from any cdf in a location-scale family, it suffices to know how to sample from F_0.

2.6 Apply the inverse-transform method to generate random variables from a *Laplace distribution* (i.e., a shifted two-sided exponential distribution) with pdf

$$f(x) = \frac{\lambda}{2}\,\mathrm{e}^{-\lambda|x-\theta|}\,, \quad -\infty < x < \infty \quad (\lambda > 0).$$

2.7 Apply the inverse-transform method to generate a random variable from the *extreme value distribution*, which has cdf

$$F(x) = 1 - \mathrm{e}^{-\exp(\frac{x-\mu}{\sigma})}\,, \quad -\infty < x < \infty \quad (\sigma > 0).$$

2.8 Consider the triangular random variable with pdf

$$f(x) = \begin{cases} 0 & \text{if } x < 2a \text{ or } x \geqslant 2b\,, \\ \dfrac{x-2a}{(b-a)^2} & \text{if } 2a \leqslant x < a+b\,, \\ \dfrac{(2b-x)}{(b-a)^2} & \text{if } a+b \leqslant x < 2b\,. \end{cases}$$

a) Derive the corresponding cdf F.

b) Show that applying the inverse-transform method yields

$$X = \begin{cases} 2a + (b-a)\sqrt{2U} & \text{if } 0 \leqslant U < \frac{1}{2}\,, \\ 2b + (a-b)\sqrt{2(1-U)} & \text{if } \frac{1}{2} \leqslant U < 1\,. \end{cases}$$

2.9 Present an inverse-transform algorithm for generating a random variable from the piecewise-constant pdf

$$f(x) = \begin{cases} C_i, & x_{i-1} \leqslant x \leqslant x_i,\ i = 1, 2, \ldots, n, \\ 0 & \text{otherwise,} \end{cases}$$

where $C_i \geqslant 0$ and $x_0 < x_1 < \cdots < x_{n-1} < x_n$.

2.10 Let

$$f(x) = \begin{cases} C_i\, x, & x_{i-1} \leqslant x < x_i, \quad i = 1, \ldots, n, \\ 0 & \text{otherwise,} \end{cases}$$

where $C_i \geqslant 0$ and $x_0 < x_1 < \cdots < x_{n-1} < x_n$.

a) Let $F_i = \sum_{j=1}^{i} \int_{x_{j-1}}^{x_j} C_j\, u \,\mathrm{d}u$, $i = 1, \ldots, n$. Show that the cdf F satisfies

$$F(x) = F_{i-1} + \frac{C_i}{2}\left(x^2 - x_{i-1}^2\right), \qquad x_{i-1} \leqslant x < x_i, \quad i = 1, \ldots, n\,.$$

b) Describe an inverse-transform algorithm for random variable generation from $f(x)$.

2.11 A random variable is said to have a *Cauchy* distribution if its pdf is given by

$$f(x) = \frac{1}{\pi}\,\frac{1}{1+x^2}, \quad x \in \mathbb{R}\,. \tag{2.39}$$

Explain how one can generate Cauchy random variables using the inverse-transform method.

2.12 If X and Y are independent standard normal random variables, then $Z = X/Y$ has a Cauchy distribution. Show this. [Hint: First show that if U and $V > 0$ are continuous random variables with joint pdf $f_{U,V}$, then the pdf of $W = U/V$ is given by $f_W(w) = \int_0^\infty f_{U,V}(w\,v, v)\, v \,\mathrm{d}v$.]

2.13 Verify the validity of the composition Algorithm 2.3.4.

2.14 Using the composition method, formulate and implement an algorithm for generating random variables from the following normal (Gaussian) mixture pdf:

$$f(x) = \sum_{i=1}^{3} p_i\, \frac{1}{b_i}\, \varphi\left(\frac{x - a_i}{b_i}\right),$$

where φ is the pdf of the standard normal distribution and $(p_1, p_2, p_3) = (1/2, 1/3, 1/6)$, $(a_1, a_2, a_3) = (-1, 0, 1)$, and $(b_1, b_2, b_3) = (1/4, 1, 1/2)$.

2.15 Verify that $C = \sqrt{2\mathrm{e}/\pi}$ in Figure 2.5.

2.16 Prove that if $X \sim \mathsf{Gamma}(\alpha, 1)$, then $X/\lambda \sim \mathsf{Gamma}(\alpha, \lambda)$.

2.17 Let $X \sim \mathsf{Gamma}(1+\alpha, 1)$ and $U \sim \mathsf{U}(0,1)$ be independent. If $\alpha < 1$, then $XU^{1/a} \sim \mathsf{Gamma}(\alpha, 1)$. Prove this.

2.18 If $Y_1 \sim \mathsf{Gamma}(\alpha, 1)$, $Y_2 \sim \mathsf{Gamma}(\beta, 1)$, and Y_1 and Y_2 are independent, then

$$X = \frac{Y_1}{Y_1 + Y_2}$$

is $\mathsf{Beta}(\alpha, \beta)$ distributed. Prove this.

2.19 Devise an acceptance–rejection algorithm for generating a random variable from the pdf f given in (2.17) using an $\mathsf{Exp}(\lambda)$ proposal distribution. Which λ gives the largest acceptance probability?

2.20 The pdf of the truncated exponential distribution with parameter $\lambda = 1$ is given by

$$f(x) = \frac{\mathrm{e}^{-x}}{1 - \mathrm{e}^{-a}}, \quad 0 \leqslant x \leqslant a \ .$$

a) Devise an algorithm for generating random variables from this distribution using the inverse-transform method.

b) Construct a generation algorithm that uses the acceptance–rejection method with an $\mathsf{Exp}(\lambda)$ proposal distribution.

c) Find the efficiency of the acceptance–rejection method for the cases $a = 1$, and a approaching zero and infinity.

2.21 Let the random variable X have pdf

$$f(x) = \begin{cases} \frac{1}{4}\,, & 0 < x < 1\ , \\ x - \frac{3}{4}\,, & 1 \leqslant x \leqslant 2\ . \end{cases}$$

Generate a random variable from $f(x)$, using

a) the inverse-transform method,

b) the acceptance–rejection method, using the proposal density

$$g(x) = \frac{1}{2}\,, \quad 0 \leqslant x \leqslant 2\ .$$

2.22 Let the random variable X have pdf

$$f(x) = \begin{cases} \frac{1}{2}\,x\ , & 0 < x < 1\ , \\ \frac{1}{2}\,, & 1 \leqslant x \leqslant \frac{5}{2}\ . \end{cases}$$

Generate a random variable from $f(x)$, using

a) the inverse-transform method

b) the acceptance–rejection method, using the proposal density

$$g(x) = \frac{8}{25}\,x\,, \quad 0 \leqslant x \leqslant \frac{5}{2}\ .$$

2.23 Let X have a truncated geometric distribution, with pdf

$$f(x) = c\,p(1-p)^{x-1},\ x = 1, \ldots, n\ ,$$

where c is a normalization constant. Generate a random variable from $f(x)$, using

a) the inverse-transform method,

b) the acceptance–rejection method, with $\mathsf{G}(p)$ as the proposal distribution. Find the efficiency of the acceptance–rejection method for $n = 2$ and $n = \infty$.

2.24 Generate a random variable $Y = \min_{i=1,\ldots,m} \max_{j=1,\ldots,r}\{X_{ij}\}$, assuming that the variables $X_{ij},\ i = 1,\ldots,m,\ j = 1,\ldots,r$, are iid with common cdf $F(x)$, using the inverse-transform method. [Hint: Use the results for the distribution of order statistics in Example 2.5.]

2.25 Generate 100 $\mathsf{Ber}(0.2)$ random variables three times and produce bar graphs similar to those in Figure 2.6. Repeat for $\mathsf{Ber}(0.5)$.

2.26 Generate a homogeneous Poisson process with rate 100 on the interval $[0, 1]$. Use this to generate a nonhomogeneous Poisson process on the same interval, with rate function

$$\lambda(t) = 100 \sin^2(10\,t), \quad t \geqslant 0\ .$$

2.27 Generate and plot a realization of the points of a two-dimensional Poisson process with rate $\lambda = 2$ on the square $[0, 5] \times [0, 5]$. How many points fall in the square $[1, 3] \times [1, 3]$? How many do you expect to fall in this square?

2.28 Write a program that generates and displays 100 random vectors that are uniformly distributed within the ellipse

$$5\,x^2 + 21\,x\,y + 25\,y^2 = 9\ .$$

2.29 Implement both random permutation algorithms in Section 2.10. Compare their performance.

2.30 Consider a random walk on the undirected graph in Figure 2.13. For example, if the random walk at some time is in state 5, it will jump to 3, 4, or 6 at the next transition, each with probability 1/3.

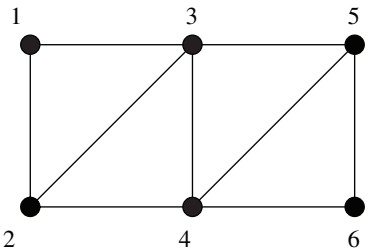

Figure 2.13: A graph.

a) Find the one-step transition matrix for this Markov chain.

b) Show that the stationary distribution is given by $\boldsymbol{\pi} = (\frac{1}{9}, \frac{1}{6}, \frac{2}{9}, \frac{2}{9}, \frac{1}{6}, \frac{1}{9})$.

c) Simulate the random walk on a computer and verify that in the long run, the proportion of visits to the various nodes is in accordance with the stationary distribution.

2.31 Generate various sample paths for the random walk on the integers for $p = 1/2$ and $p = 2/3$.

2.32 Consider the $M/M/1$ queueing system of Example 1.13. Let X_t be the number of customers in the system at time t. Write a computer program to simulate the stochastic process $X = \{X_t\}$ by viewing X as a Markov jump process, and applying Algorithm 2.7.2. Present sample paths of the process for the cases $\lambda = 1$, $\mu = 2$ and $\lambda = 10$, $\mu = 11$.

Further Reading

Classical references on random number generation and random variable generation are [5] and [2]. Other references include [8], [14], and [18] and the tutorial in [17]. A good reference is [1]. The simulation of spatial processes is discussed in [6].

REFERENCES

1. S. Asmussen and P. W. Glynn. *Stochastic Simulation*. Springer-Verlag, New York, 2007.
2. L. Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, New York, 1986.
3. P. Glasserman. *Monte Carlo Methods in Financial Engineering*. Springer-Verlag, New York, 2004.
4. P. E. Kloeden and E. Platen. *Numerical Solution of Stochastic Differential Equations*. Springer-Verlag, New York, 1992. corrected third printing.
5. D. E. Knuth. *The Art of Computer Programming*, volume 2: *Seminumerical Algorithms*. Addison-Wesley, Reading, MA, 2nd edition, 1981.
6. D. P. Kroese and Z. I. Botev. Spatial process simulation. In V. Schmidt, editor, *Lectures on Stochastic Geometry, Spatial Statistics and Random Fields*, volume II: Analysis, Modeling and Simulation of Complex Structures. Springer, Berlin, 2014.
7. D. P. Kroese, T. Taimre, and Z. I. Botev. *Handbook of Monte Carlo Methods*. John Wiley & Sons, 2011.
8. A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, New York, 3rd edition, 2000.
9. P. L'Ecuyer. Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47(1):159 – 164, 1999.
10. P. L'Ecuyer and F. Panneton. $\mathbb{F}_2$-linear random number generators. In C. Alexopoulos, D. Goldsman, and J. R. Wilson, editors, *Advancing the Frontiers of Simulation: A Festschrift in Honor of George Samuel Fishman*, pages 175–200, New York, 2009. Springer-Verlag.
11. P. L'Ecuyer and R. Simard. TestU01: A C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*, 33(4), 2007. Article 22.
12. D. H. Lehmer. Mathematical methods in large-scale computing units. *Annals of the Computation Laboratory of Harvard University*, 26:141–146, 1951.

13. P. A. Lewis, A. S. Goodman, and J. M. Miller. A pseudo-random number generator for the system/360. *IBM Systems Journal*, 8(2):136–146, 1969.

14. N. N. Madras. *Lectures on Monte Carlo Methods*. American Mathematical Society, 2002.

15. G. Marsaglia and W. Tsang. A simple method for generating gamma variables. *ACM Transactions on Mathematical Software*, 26(3):363–372, 2000.

16. M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.

17. B. D. Ripley. Computer generation of random variables: A tutorial. *International Statistical Review*, 51:301–319, 1983.

18. S. M. Ross. *Simulation*. Academic Press, New York, 3rd edition, 2002.

19. A. J. Walker. An efficient method for generating discrete random variables with general distributions. *ACM Transactions on Mathematical Software*, 3:253–256, 1977.

CHAPTER 3

SIMULATION OF DISCRETE-EVENT SYSTEMS

3.1 INTRODUCTION

Computer simulation has long served as an important tool in a wide variety of disciplines: engineering, operations research and management science, statistics, mathematics, physics, economics, biology, medicine, engineering, chemistry, and the social sciences. Through computer simulation, one can study the behavior of real-life systems that are too difficult to examine analytically. Examples can be found in supersonic jet flight, telephone communications systems, wind tunnel testing, large-scale battle management (e.g., to evaluate defensive or offensive weapons systems), or maintenance operations (e.g., to determine the optimal size of repair crews), to mention a few. Recent advances in simulation methodologies, software availability, sensitivity analysis, and stochastic optimization have combined to make simulation one of the most widely accepted and used tools in system analysis and operations research. The sustained growth in size and complexity of emerging real-world systems (e.g., high-speed communication networks and biological systems) will undoubtedly ensure that the popularity of computer simulation continues to grow.

The aim of this chapter is to provide a brief introduction to the art and science of computer simulation, in particular with regard to discrete-event systems. The chapter is organized as follows: Section 3.2 describes basic concepts such as systems, models, simulation, and Monte Carlo methods. Section 3.3 deals with the most fundamental ingredients of discrete-event simulation, namely, the simulation clock and the event list. Section 3.4 explains the ideas behind discrete-event simulation via a number of worked examples.

Simulation and the Monte Carlo Method, Third Edition. By R. Y. Rubinstein and D. P. Kroese

3.2 SIMULATION MODELS

By a *system* we mean a collection of related entities, sometimes called *components* or *elements*, forming a complex whole. For instance, a hospital may be considered a system, with doctors, nurses, and patients as elements. The elements possess certain characteristics or *attributes* that take on logical or numerical values. In our example, an attribute may be the number of beds, the number of X-ray machines, skill level, and so on. Typically, the activities of individual components interact over time. These activities cause changes in the system's state. For example, the state of a hospital's waiting room might be described by the number of patients waiting for a doctor. When a patient arrives at the hospital or leaves it, the system jumps to a new state.

We will be solely concerned with *discrete-event systems*, to wit, those systems in which the state variables change instantaneously through jumps at discrete points in time, as opposed to *continuous systems*, where the state variables change continuously with respect to time. Examples of discrete and continuous systems are, respectively, a bank serving customers and a car moving on the freeway. In the former case, the number of waiting customers is a piecewise constant state variable that changes only when either a new customer arrives at the bank or a customer finishes transacting his business and departs from the bank; in the latter case, the car's velocity is a state variable that can change continuously over time.

The first step in studying a system is to build a model from which to obtain predictions concerning the system's behavior. By a *model* we mean an abstraction of some real system that can be used to obtain predictions and formulate control strategies. Often such models are mathematical (formulas, relations) or graphical in nature. Thus, the actual physical system is translated — through the model — into a mathematical system. In order to be useful, a model must necessarily incorporate elements of two conflicting characteristics: realism and simplicity. On the one hand, the model should provide a reasonably close approximation to the real system and incorporate most of the important aspects of the real system. On the other hand, the model must not be so complex as to preclude its understanding and manipulation.

There are several ways to assess the validity of a model. Usually, we begin testing a model by reexamining the formulation of the problem and uncovering possible flaws. Another check on the validity of a model is to ascertain that all mathematical expressions are dimensionally consistent. A third useful test consists of varying input parameters and checking that the output from the model behaves in a plausible manner. The fourth test is the so-called *retrospective* test. It involves using historical data to reconstruct the past and then determining how well the resulting solution would have performed if it had been used. Comparing the effectiveness of this hypothetical performance with what actually happens then indicates how well the model predicts reality. However, a disadvantage of retrospective testing is that it uses the same data as the model. Unless the past is a representative replica of the future, it is better not to resort to this test at all.

Once a model for the system at hand has been constructed, the next step is to derive a solution from this model. To this end, both *analytical* and *numerical* solutions methods may be invoked. An analytical solution is usually obtained directly from its mathematical representation in the form of formulas. A numerical solution is generally an approximation via a suitable approximation procedure.

Much of this book deals with numerical solution and estimation methods obtained via computer simulation. More precisely, we use *stochastic computer simulation* — often called *Monte Carlo simulation* — which includes some randomness in the underlying model, rather than deterministic computer simulation. The term *Monte Carlo* was used by von Neumann and Ulam during World War II as a code word for secret work at Los Alamos on problems related to the atomic bomb. That work involved simulation of random neutron diffusion in nuclear materials.

Naylor et al. [7] define *simulation* as follows:

> *Simulation is a numerical technique for conducting experiments on a digital computer, which involves certain types of mathematical and logical models that describe the behavior of business or economic systems (or some component thereof) over extended period of real time.*

The following list of typical situations should give the reader some idea of where simulation would be an appropriate tool.

- The system may be so complex that a formulation in terms of a simple mathematical equation may be impossible. Most economic systems fall into this category. For example, it is often virtually impossible to describe the operation of a business firm, an industry, or an economy in terms of a few simple equations. Another class of problems that leads to similar difficulties is that of large-scale, complex queueing systems. Simulation has been an extremely effective tool for dealing with problems of this type.

- Even if a mathematical model can be formulated that captures the behavior of some system of interest, it may not be possible to obtain a solution to the problem embodied in the model by straightforward analytical techniques. Again, economic systems and complex queueing systems exemplify this type of difficulty.

- Simulation may be used as a pedagogical device for teaching both students and practitioners basic skills in systems analysis, statistical analysis, and decision making. Among the disciplines in which simulation has been used successfully for this purpose are business administration, economics, medicine, and law.

- The formal exercise of designing a computer simulation model may be more valuable than the actual simulation itself. The knowledge obtained in designing a simulation study serves to crystallize the analyst's thinking and often suggests changes in the system being simulated. The effects of these changes can then be tested via simulation before implementing them in the real system.

- Simulation can yield valuable insights into the problem of identifying which variables are important and which have negligible effects on the system, and can shed light on how these variables interact; see Chapter 7.

- Simulation can be used to experiment with new scenarios so as to gain insight into system behavior under new circumstances.

- Simulation provides an *in silico* lab, allowing the analyst to discover better control of the system under study.

- Simulation makes it possible to study dynamic systems in either real, compressed, or expanded time horizons.
- Introducing randomness in a system can actually help solve many optimization and counting problems; see Chapters 6 – 10.

As a modeling methodology, simulation is by no means ideal. Some of its shortcomings and various caveats are: Simulation provides *statistical estimates* rather than *exact* characteristics and performance measures of the model. Thus, simulation results are subject to uncertainty and contain experimental errors. Moreover, simulation modeling is typically time-consuming and consequently expensive in terms of analyst time. Finally, simulation results, no matter how precise, accurate, and impressive, provide consistently useful information about the actual system *only* if the model is a valid representation of the system under study.

3.2.1 Classification of Simulation Models

Computer simulation models can be classified in several ways:

1. *Static versus Dynamic Models.* Static models are those that do not evolve over time and therefore do not represent the passage of time. In contrast, dynamic models represent systems that evolve over time (for example, traffic light operation).
2. *Deterministic versus Stochastic Models.* If a simulation model contains *only* deterministic (i.e., nonrandom) components, it is called *deterministic*. In a deterministic model, all mathematical and logical relationships between elements (variables) are fixed in advance and not subject to uncertainty. A typical example is a complicated and analytically unsolvable system of standard differential equations describing, say, a chemical reaction. In contrast, a model with at least one random input variable is called a *stochastic* model. Most queueing and inventory systems are modeled stochastically.
3. *Continuous versus Discrete Simulation Models.* In discrete simulation models the state variable changes instantaneously at discrete points in time, whereas in continuous simulation models the state changes continuously over time. A mathematical model aiming to calculate a numerical solution for a system of differential equations is an example of continuous simulation, while queueing models are examples of discrete simulation.

This chapter deals with discrete simulation and in particular with *discrete-event simulation* (DES) models. The associated systems are driven by the occurrence of discrete events, and their state typically changes over time. We shall further distinguish between so-called *discrete-event static systems* (DESS) and *discrete-event dynamic systems* (DEDS). The fundamental difference between DESS and DEDS is that the former do not evolve over time, whereas the latter do. A queueing network is a typical example of a DEDS. A DESS usually involves evaluating (estimating) complex multidimensional integrals or sums via Monte Carlo simulation.

Remark 3.2.1 (Parallel Computing) Recent advances in computer technology have enabled the use of *parallel* or *distributed* simulation, where discrete-event simulation is carried out on multiple linked (networked) computers, operating simultaneously in a cooperative manner. Such an environment allows simultaneous distribution of different computing tasks among the individual processors, thus reducing the overall simulation time.

3.3 SIMULATION CLOCK AND EVENT LIST FOR DEDS

Recall that DEDS evolve over time. In particular, these systems change their state only at a countable number of time points. State changes are triggered by the execution of simulation events occurring at the corresponding time points. Here, an *event* is a collection of attributes (values, types, flags, etc.), chief among which are the *event occurrence time* — or simply *event time* — the *event type*, and an associated algorithm to execute state changes.

Because of their dynamic nature, DEDS require a time-keeping mechanism to advance the simulation time from one event to another as the simulation evolves over time. The mechanism recording the current simulation time is called the *simulation clock*. To keep track of events, the simulation maintains a list of all pending events. This list is called the *event list*, and its task is to maintain all pending events in *chronological* order. That is, events are ordered by their time of occurrence. In particular, the most imminent event is always located at the head of the event list.

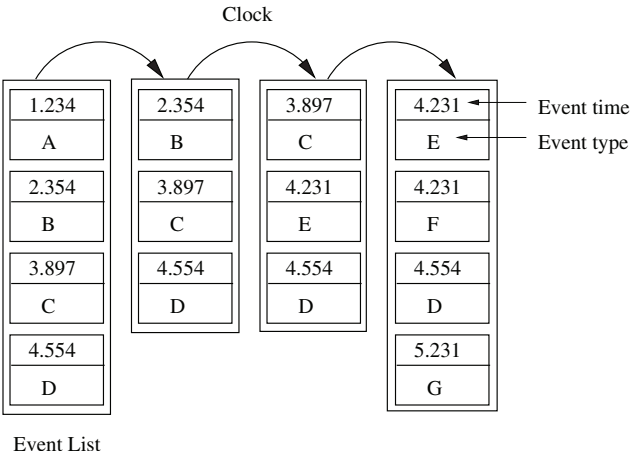

Figure 3.1: The advancement of the simulation clock and event list.

The situation is illustrated in Figure 3.1. The simulation starts by loading the initial events into the event list (chronologically ordered), in this case four events. Next, the most imminent event is unloaded from the event list for execution, and the simulation clock is advanced to its occurrence time, 1.234. After this event is processed and removed, the clock is advanced to the next event, which occurs at time 2.354. In the course of executing a current event, based on its type, the state of the system is updated, and future events are possibly generated and loaded into (or deleted from) the event list. In the example above, the third event — of type C, occurring at time 3.897 — schedules a new event of type E at time 4.231.

The process of unloading events from the event list, advancing the simulation clock, and executing the next most imminent event terminates when some specific stopping condition is met — say, as soon as a prescribed number of customers departs from the system. The following example illustrates this *next-event time advance* approach.

EXAMPLE 3.1

Money enters a certain bank account in two ways: via frequent small payments and occasional large payments. Suppose that the times between subsequent frequent payments are independent and uniformly distributed on the continuous interval $[7, 10]$ (in days); and, similarly, the times between subsequent occasional payments are independent and uniformly distributed on $[25, 35]$. Each frequent payment is exponentially distributed with a mean of 16 units (e.g., one unit is \$1000), whereas occasional payments are always of size 100. It is assumed that all payment intervals and sizes are independent. Money is debited from the account at times that form a Poisson process with rate 1 (per day), and the amount debited is normally distributed with mean 5 and standard deviation 1. Suppose that the initial amount of money in the bank account is 150 units.

Note that the state of the system — the account balance — changes only at discrete times. To simulate this DEDS, one need only keep track of when the next frequent and occasional payments occur, as well as the next withdrawal. Denote these three event types by 1, 2, and 3, respectively. We can now implement the event list simply as a 3×2 matrix, where each row contains the event time and the event type. After each advance of the clock, the current event time t and event type i are recorded and the current event is erased. Next, for each event type $i = 1, 2, 3$, the same type of event is scheduled using its corresponding interval distribution. For example, if the event type is 2, then another event of type 2 is scheduled at a time $t + 25 + 10\,U$, where $U \sim \mathsf{U}[0, 1]$. Note that this event can be stored in the same location as the current event that was just erased. However, it is crucial that the event list is then resorted to put the events in chronological order.

A realization of the stochastic process $\{X_t, 0 \leqslant t \leqslant 400\}$, where X_t is the account balance at time t, is given in Figure 3.2, followed by a simple Matlab implementation.

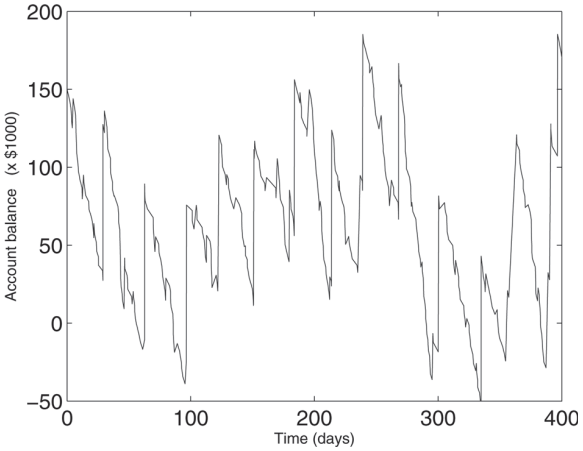

Figure 3.2: A realization of the simulated account balance process.

Matlab Program

```
clear all
T = 400;
x = 150; %initial amount of money.
xx = [150]; tt = [0];
t=0;
ev_list = inf*ones(3,2);          %record time, type
ev_list(1,:) = [7 + 3*rand, 1];  %schedule type 1 event
ev_list(2,:) = [25 + 10*rand,2]; %schedule type 2 event
ev_list(3,:) = [-log(rand),3];   %schedule type 3 event
ev_list = sortrows(ev_list,1);   % sort event list
while t < T
    t = ev_list(1,1);
    ev_type = ev_list(1,2);
    switch ev_type
        case 1
            x = x + 16*-log(rand);
            ev_list(1,:) = [7 + 3*rand + t, 1];
        case 2
            x = x + 100;
            ev_list(1,:) = [25 + 10*rand + t, 2];
        case 3
            x = x - (5 + randn);
            ev_list(1,:) = [-log(rand) + t, 3];
    end
    ev_list = sortrows(ev_list,1); % sort event list
    xx = [xx,x];
    tt = [tt,t];
end
plot(tt,xx)
```

3.4 DISCRETE-EVENT SIMULATION

As mentioned, DES is the standard framework for the simulation of a large class of models in which the system *state* (one or more quantities that describe the condition of the system) needs to be observed only at certain critical epochs (event times). Between these epochs, the system state either stays the same or changes in a predictable fashion. We further explain the ideas behind DES via two more examples.

3.4.1 Tandem Queue

Figure 3.3 depicts a simple queueing system, consisting of two queues in tandem, called a (Jackson) *tandem queue*. Customers arrive at the first queue according to a Poisson process with rate λ. The service time of a customer at the first queue is exponentially distributed with rate μ_1. Customers who leave the first queue enter the second one. The service time in the second queue has an exponential distribution with rate μ_2. All interarrival and service times are independent.

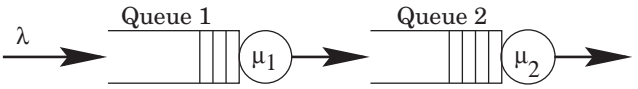

Figure 3.3: A Jackson tandem queue.

Suppose that we are interested in the number of customers, X_t and Y_t, in the first and second queues, respectively, where we regard a customer who is being served as part of the queue. Figure 3.4 depicts a typical realization of the queue length processes $\{X_t,\ t \geqslant 0\}$ and $\{Y_t,\ t \geqslant 0\}$ obtained via DES.

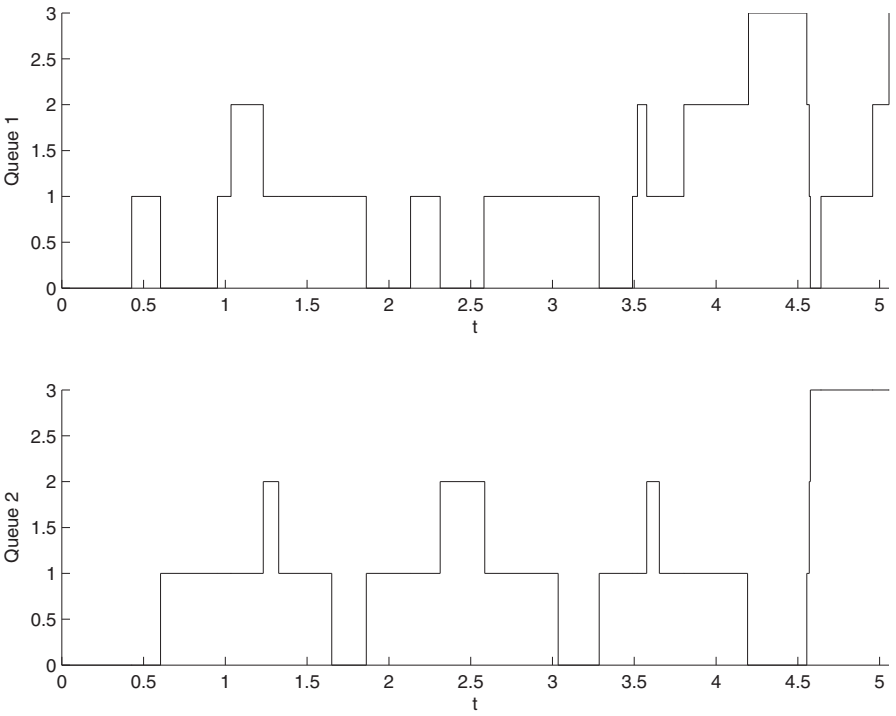

Figure 3.4: A realization of the queue length processes $(X_t, t \geqslant 0)$ and $(Y_t, t \geqslant 0)$.

Before we discuss how to simulate the queue length processes via DES, observe that the system evolves via a sequence of discrete events, as illustrated in Figure 3.5. Specifically, the system state (X_t, Y_t) changes only at times of an arrival at the first queue (indicated by A), a departure from the first queue (indicated by D1), and a departure from the second queue (D2).

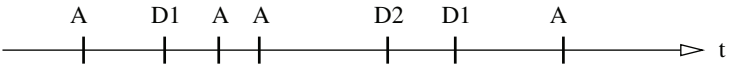

Figure 3.5: Sequence of discrete events (A = arrival, D1 = departure from the first queue, D2 = departure from the second queue).

There are two fundamental approaches to DES, called the *event-oriented* and *process-oriented* approaches. The pseudocode for an event-oriented implementation of the tandem queue is given in Figure 3.6. The program consists of a main subroutine and separate subroutines for each event. In addition, the program maintains an ordered list of scheduled current and future events, the so-called *event list*. Each event in the event list has an event *type* ('A', 'D1', and 'D2') and an event *time* (the time at which the arrival or departure will occur). The role of the main subroutine is primarily to progress through the event list and to call the subroutines that are associated with each event type.

Main

```
1  initialize: t ← 0, x ← 0, y ← 0
2  Schedule 'A' at t + Exp(λ).
3  while true do
4      Get the first event in the event list.
5      Let t be the time of this (now current) event.
6      switch current event type do
7          case 'A' : Call Arrival
8          case 'D1' : Call Departure1
9          case 'D2' : Call Departure2
10     Remove the current event from the event list and sort the event list.
```

Figure 3.6: Main subroutine of an event-oriented simulation program.

The role of the event subroutines is to update the system state and to schedule new events into the event list. For example, an arrival event at time t will trigger another arrival event at time $t+Z$, with $Z \sim \mathsf{Exp}(\lambda)$. We write this, as in the Main routine, in shorthand as $t + \mathsf{Exp}(\lambda)$. Moreover, if the first queue is empty, it will also trigger a departure event from the first queue at time $t + \mathsf{Exp}(\mu_1)$.

Arrival

```
1  Schedule 'A'
       at t + Exp(λ).
2  if x = 0 then
3      Schedule 'D1'
           at t + Exp(μ1).
4  x ← x + 1
```

Departure1

```
1  x ← x − 1
2  if x ≠ 0 then
3      Schedule 'D1'
           at t + Exp(μ1).
4  if y = 0 then
5      Schedule 'D2'
           at t + Exp(μ2).
6  y ← y + 1
```

Departure2

```
1  y ← y − 1
2  if y ≠ 0 then
3      Schedule 'D2'
           at t + Exp(μ2).
```

Figure 3.7: Event subroutines of an event-oriented simulation program.

The process-oriented approach to DES is much more flexible than the event-oriented approach. A process-oriented simulation program closely resembles the

actual processes that drive the simulation. Such simulation programs are invariably written in an object-oriented programming language, such as Java or C++. We illustrate the process-oriented approach via our tandem queue example. In contrast to the event-oriented approach, customers, servers, and queues are now actual entities, or *objects* in the program, that can be manipulated. The queues are passive objects that can contain various customers (or be empty), and the customers themselves can contain information such as their arrival and departure times. The servers, however, are active objects (*processes*) that can interact with each other and with the passive objects. For example, the first server takes a client out of the first queue, serves the client, and puts her into the second queue when finished, alerting the second server that a new customer has arrived if necessary. To generate the arrivals, we define a *generator* process that generates a client, puts it in the first queue, alerts the first server if necessary, holds for a random interarrival time (we assume that the interarrival times are iid), and then repeats these actions to generate the next client.

As in the event-oriented approach, there exists an event list that keeps track of the current and pending events. However, this event list now contains *processes*. The process at the top of the event list is the one that is currently active. Processes may ACTIVATE other processes by putting them at the head of the event list. Active processes may HOLD their action for a certain amount of time (such processes are put further up in the event list). Processes may PASSIVATE altogether (temporarily remove themselves from the event list). Figure 3.8 lists the typical structure of a process-oriented simulation program for the tandem queue.

Main

```
1 initialize: create the two queues, the two servers and the generator.
2 ACTIVATE the generator.
3 HOLD(duration of simulation).
4 STOP
```

Generator

```
1 while true do
2 |   Generate new client.
3 |   Put client in the first queue.
4 if server 1 is idle then
5 |   ACTIVATE server 1.
6 HOLD(interarrival time).
```

Server 1

```
1 while true do
2 |   if waiting room is empty then
3 |   |   PASSIVATE
4 |   else
5 |   |   Get first customer from
  |   |     waiting room.
6 |   |   HOLD(service time).
7 |   Put customer in queue 2.
8 |   if server 2 is idle then
9 |   |   ACTIVATE server 2.
```

Figure 3.8: The structure of a process-oriented simulation program for the tandem queue. The Server 2 process is similar to the Server 1 process, with lines 7–9 replaced with "remove customer from system".

The collection of statistics (for example, the waiting times or queue lengths), can be done by different objects and at various stages in the simulation. For example, customers can record their arrival and departure times and report or record them just before they leave the system. There are many freely available object-oriented simulation environments nowadays, such as SSJ, SimPy, and C++Sim, all inspired by the pioneering simulation language SIMULA.

3.4.2 Repairman Problem

Imagine n machines working simultaneously. The machines are unreliable and fail from time to time. There are $m < n$ identical repairmen who can each work only on one machine at a time. When a machine has been repaired, it is as good as new. Each machine has a fixed lifetime distribution and repair time distribution. We assume that the lifetimes and repair times are independent of each other. Since there are fewer repairmen than machines, it can happen that a machine fails and all repairmen are busy repairing other failed machines. In that case, the failed machine is placed in a queue to be served by the next available repairman. When upon completion of a repair job a repairman finds the failed machine queue empty, he enters the repair pool and remains idle until his service is required again. We assume that machines and repairmen enter their respective queues in a first-in-first-out (FIFO) manner. The system is illustrated in Figure 3.9 for the case of three repairmen and five machines.

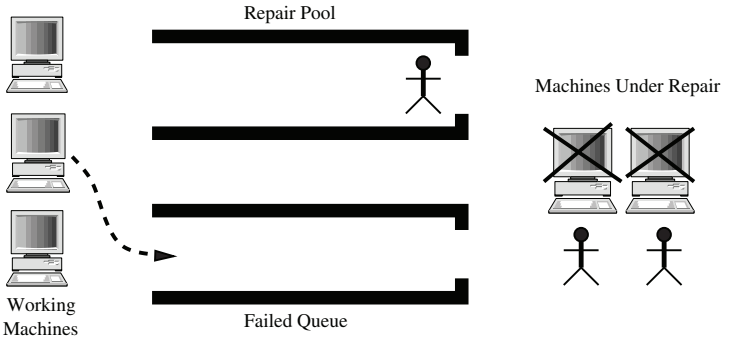

Figure 3.9: The repairman system.

For this particular model the system state could be comprised of the number of available repairmen R_t and the number of failed machines F_t at any time t. In general, the stochastic process $\{(F_t, R_t), t \geqslant 0\}$ is not a Markov process unless the service and lifetimes have exponential distributions.

As with the tandem queue, we first describe an event-oriented and then a process-oriented approach for this model.

3.4.2.1 Event-Oriented Approach There are two types of events: failure events 'F' and repair events 'R'. Each event triggers the execution of the corresponding failure or repair procedure. The task of the main program is to advance the simulation clock and to assign the correct procedure to each event. Denoting by n_f the number of failed machines and by n_r the number of available repairmen, we write the main program in the following form:

MAIN PROGRAM

```
initialize: Let t ← 0, n_r ← m and n_f ← 0. for i = 1 to n. do
    Schedule 'F' of machine i at time t+lifetime(i).
while true do
    Get the first event in the event list.
    Let t be the time of this (now current) event.
    Let i be the machine number associated with this event.
    switch current event type do
        case 'F' : Call Failure
        case 'R' : Call Repair
    Remove the current event from the event list.
```

Upon failure, a repair needs to be scheduled at a time equal to the current time plus the required repair time for the particular machine. However, this is true only if there is a repairman available to carry out the repairs. If this is not the case, the machine is placed in the "failed" queue. The number of failed machines is always increased by 1. The failure procedure is thus as follows:

FAILURE PROCEDURE

```
if n_r > 0 then
    Schedule 'R' of machine i at time t + repairtime(i).
    n_r ← n_r − 1
else
    Add the machine to the repair queue.
n_f ← n_f + 1
```

Upon repair, the number of failed machines is decreased by 1. The machine that has just been repaired is scheduled for its next failure. If the "failed" queue is not empty, the repairman takes the next machine from the queue and schedules a corresponding repair event. Otherwise, the number of idle/available repairmen is increased by 1. This gives the following repair procedure:

REPAIR PROCEDURE

```
n_f ← n_f − 1
Schedule 'F' for machine i at time t+lifetime(i).
if repair pool not empty then
    Remove the first machine from the "failed" queue; let j be its number.
    Schedule 'R' of machine j at time t+repairtime(j).
else
    n_r ← n_r + 1
```

3.4.2.2 Process-Oriented Approach To outline a process-oriented approach for any simulation, it is convenient to represent the processes by flowcharts. In this case there are two processes: the repairman process and the machine process. The flowcharts in Figure 3.10 are self-explanatory. Note that the horizontal parallel lines in the flowcharts indicate that the process PASSIVATEs, that is, the process

temporarily stops (is removed from the event list), until it is ACTIVATEd by another process. The circled letters A and B indicate how the two interact. A cross in the flowchart indicates that the process is rescheduled in the event list (E.L.). This happens in particular when the process HOLDs for an amount of time. After holding, it resumes from where it left off.

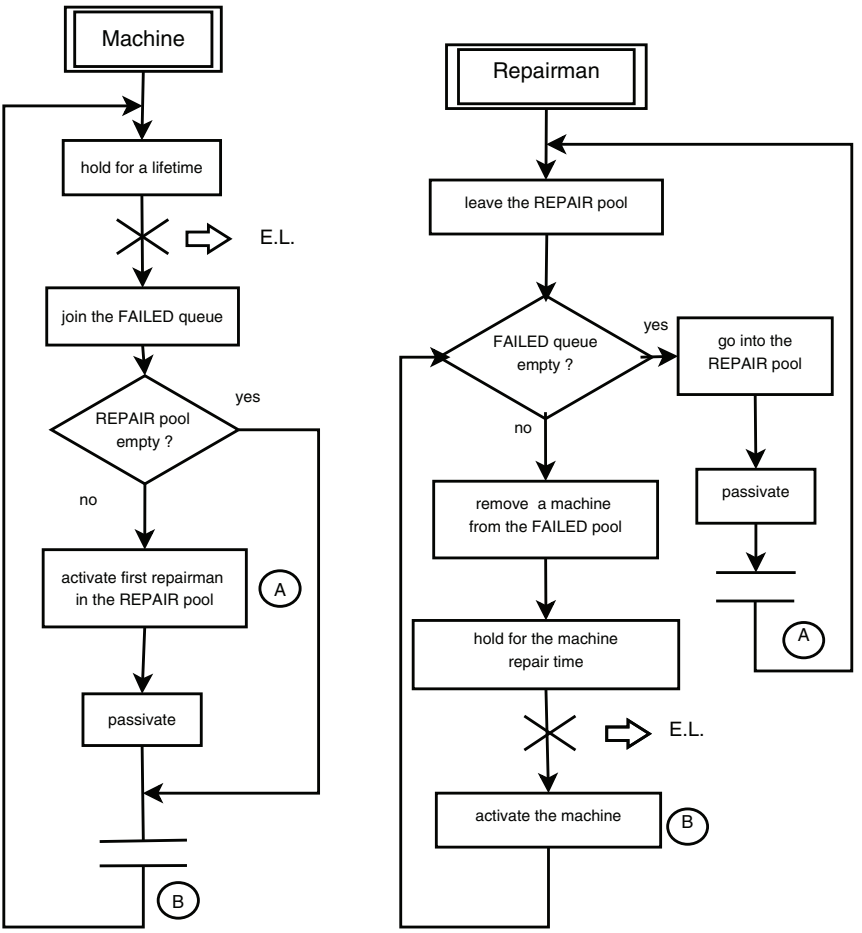

Figure 3.10: Flowcharts for the two processes in the repairman problem.

PROBLEMS

3.1 Consider the $M/M/1$ queueing system in Example 1.13. Let X_t be the number of customers in the system at time t. Write a computer program to simulate the stochastic process $X = \{X_t, t \geqslant 0\}$ using an event- or process-oriented DES approach. Present sample paths of the process for the cases $\lambda = 1$, $\mu = 2$ and $\lambda = 10$, $\mu = 11$.

3.2 Repeat the above simulation, but now assume $\mathsf{U}(0,2)$ interarrival times and $\mathsf{U}(0,1/2)$ service times (all independent).

3.3 Run the Matlab program of Example 3.1 (or implement it in the computer language of your choice). Out of 1000 runs, how many lead to a negative account balance during the first 100 days? How does the process behave for large t?

3.4 Implement an event-oriented simulation program for the tandem queue. Let the interarrivals be exponentially distributed with mean 5, and let the service times be uniformly distributed on [3,6]. Plot realizations of the queue length processes of both queues.

3.5 Consider the repairman problem with two identical machines and one repairman. We assume that the lifetime of a machine has an exponential distribution with expectation 5 and that the repair time of a machine is exponential with expectation 1. All the lifetimes and repair times are independent of each other. Let X_t be the number of failed machines at time t.

- **a)** Verify that $X = \{X_t, t \geqslant 0\}$ is a birth-and-death process, and give the corresponding birth and death rates.
- **b)** Write a program that simulates the process X according to Algorithm 2.7.2 and use this to assess the fraction of time that both machines are out of order. Simulate from $t = 0$ to $t = 100{,}000$.
- **c)** Write an event-oriented simulation program for this process.
- **d)** Let the exponential life and repair times be uniformly distributed, on $[0, 10]$ and $[0, 2]$, respectively (hence the expectations stay the same as before). Simulate from $t = 0$ to $t = 100{,}000$. How does the fraction of time that both machines are out of order change?
- **e)** Now simulate a repairman problem with the above-given life and repair times, but now with five machines and three repairmen. Run again from $t = 0$ to $t = 100{,}000$.

3.6 Draw flow diagrams, such as in Figure 3.10, for all the processes in the tandem queue; see also Figure 3.8.

3.7 Consider the following queueing system. Customers arrive at a circle, according to a Poisson process with rate λ. On the circle, which has circumference 1, a single server travels at constant speed α^{-1}. Upon arrival the customers choose their positions on the circle according to a uniform distribution. The server always moves toward the nearest customer, sometimes clockwise, sometimes counterclockwise. Upon reaching a customer, the server stops and serves the customer according to an exponential service time distribution with parameter μ. When the server is finished, the customer is removed from the circle and the server resumes his journey on the circle. Let $\eta = \lambda\,\alpha$, and let $X_t \in [0, 1]$ be the position of the server at time t. Furthermore, let N_t be the number of customers waiting on the circle at time t. Implement a simulation program for this so-called *continuous poling system with a "greedy" server*, and plot realizations of the processes $\{X_t, t \geqslant 0\}$ and $\{N_t, t \geqslant 0\}$, taking the parameters $\lambda = 1$, $\mu = 2$, for different values of α. Note that although the state space of $\{X_t, t \geqslant 0\}$ is continuous, the system is still a DEDS, since between arrival and service events the system state changes deterministically.

3.8 Consider a *continuous flow line* consisting of three machines in tandem separated by two storage areas, or buffers, through which a continuous (fluid) stream of items flows from one machine to the next; see Figure 3.11.

Figure 3.11: A flow line with three machines and two buffers (three-stage flow line).

Each machine $i = 1, 2, 3$ has a specific *machine speed* ν_i, which is the maximum rate at which it can transfer products from its upstream buffer to its downstream buffer. The lifetime of machine i has an exponential distribution with parameter λ_i. The repair of machine i starts immediately after failure and requires an exponential time with parameter μ_i. All life and repair times are assumed to be independent of each other. Failures are operation independent. In particular, the failure rate of a "starved" machine (a machine that is idle because it does not receive input from its upstream buffer) is the same as that of a fully operational machine. The first machine has an unlimited supply.

Suppose all machine speeds are 1, the buffers are of equal size b, and all machines are identical with parameters $\lambda = 1$ and $\mu = 2$.

a) Implement an event- or process-oriented simulation program for this system.

b) Assess via simulation the average *throughput* of the system (the long-run amount of fluid that enters/leaves the system per unit of time) as a function of the buffer size b.

Further Reading

One of the first books on Monte Carlo simulation is by Hammersley and Handscomb [3]. Kalos and Whitlock [4] is another classical reference. The event- and process-oriented approaches to discrete-event simulation are elegantly explained in Mitrani [6]. Among the great variety of books on DES, all focusing on different aspects of the modeling and simulation process, we mention [5], [8], [1], and [2]. The choice of computer language in which to implement a simulation program is very subjective. The simple models discussed in this chapter can be implemented in any standard computer language, even Matlab, although the latter does not provide easy event list manipulation. Commercial simulation environments such as ARENA/SIMAN and SIMSCRIPT II.5 make the implementation of larger models much easier. Alternatively, various free SIMULA-like Java packages exist that offer fast implementation of event- and process-oriented simulation programs. Examples are Pierre L'Ecuyer's SSJ `http://www.iro.umontreal.ca/~simardr/ssj/`, DSOL `http://sk-3.tbm.tudelft.nl/simulation/`, developed by the Technical University Delft, and the python-based SimPy `https://pypi.python.org/pypi/simpy`.