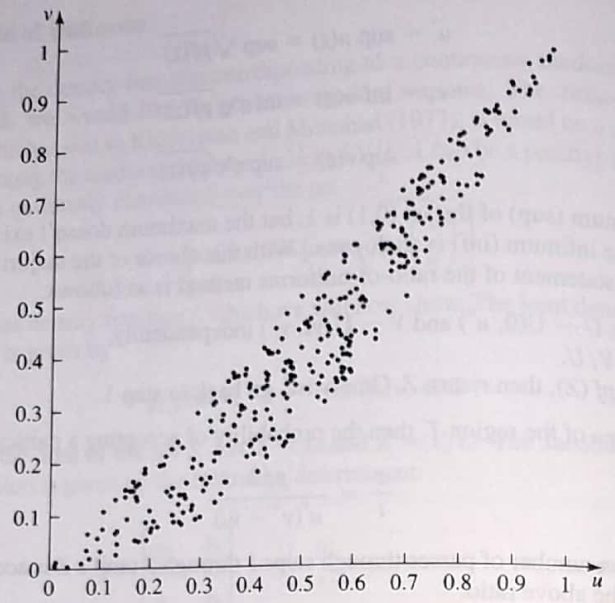


FIGURE 8.13
Accepted points for the ratio-of-uniforms method.

discrete distributions, while Wakefield et al. (1991) and Stefănescu and Văduva (1987) consider multivariate distributions.

8.2.6 Special Properties

Although most methods for generating random variates can be classified into one of the five approaches discussed so far in Sec. 8.2, some techniques simply rely on some *special property* of the desired distribution function F or the random variable X. Frequently, the special property will take the form of representing X in terms of other random variables that are more easily generated; in this sense the method of convolution is a "special" special property. The following four examples are based on normal-theory random variables (see Secs. 8.3 and 8.4 for other examples that have nothing to do with the normal distribution).

EXAMPLE 8.9. If $Y \sim \mathrm{N}(0, 1)$ (the standard normal distribution), then Y^2 has a chi-square distribution with 1 df. (We write $X \sim \chi^2_k$ to mean that X has a chi-square distribution with k df.) Thus, to generate $X \sim \chi^2_1$, generate $Y \sim \mathrm{N}(0, 1)$ (see Sec. 8.3.6), and return $X = Y^2$.

EXAMPLE 8.10. If $Z_1, Z_2, \ldots, Z_k$ are IID χ^2_1 random variables, then $X = Z_1 + Z_2 + \cdots + Z_k \sim \chi^2_k$. Thus, to generate $X \sim \chi^2_k$, first generate $Y_1, Y_2, \ldots, Y_k$ as IID N(0, 1) random variates, and then return $X = Y_1^2 + Y_2^2 + \cdots + Y_k^2$ (see Example 8.9). Since for large k this may be quite slow, we might want to exploit the fact that the χ^2_k distribution is a gamma distribution with shape parameter $\alpha = k/2$ and scale parameter $\beta = 2$. Then X can be obtained directly from the gamma-generation methods discussed in Sec. 8.3.4.

EXAMPLE 8.11. If $Y \sim \mathrm{N}(0, 1)$, $Z \sim \chi^2_k$, and Y and Z are independent, then $X = Y/\sqrt{Z/k}$ is said to have *Student's t distribution* with k df, which we denote $X \sim t_k$. Thus, to generate $X \sim t_k$, we generate $Y \sim \mathrm{N}(0, 1)$ and $Z \sim \chi^2_k$ independently of Y (see Example 8.10), and return $X = Y/\sqrt{Z/k}$.

EXAMPLE 8.12. If $Z_1 \sim \chi^2_{k_1}$, $Z_2 \sim \chi^2_{k_2}$, and Z_1 and Z_2 are independent, then

$$X = \frac{Z_1/k_1}{Z_2/k_2}$$

is said to have an *F distribution* with (k_1, k_2) df, denoted $X \sim F_{k_1,k_2}$. We thus generate $Z_1 \sim \chi^2_{k_1}$ and $Z_2 \sim \chi^2_{k_2}$ independently, and return $X = (Z_1/k_1)/(Z_2/k_2)$.

For some continuous distributions, it is possible to transform the density function so that it is easy to construct majorizing functions for use with the acceptance-rejection method. In particular, Hörmann (1995) suggested the *transformed density rejection method* for generating random variates from a continuous distribution with density f. The idea is to transform f by a strictly increasing function T so that $T(f(x))$ is concave, in which case we say that f is *T-concave*. [A function g is said to be *concave* if

$$\frac{g(x_1) + g(x_2)}{2} < g\left(\frac{x_1 + x_2}{2}\right)$$

for $x_1 < x_2$.] Since $T(f(x))$ is a concave function, a majorizing function for $T(f(x))$ can be constructed easily as the minimum of several tangents. Then T^{-1} is used to transform the majorizing function back to the original scale. This results in a majorizing function for the density f, and random variates can be generated from f by the acceptance-rejection method. If $T(x) = -1/\sqrt{x}$, then a large number of distributions are T-concave, including the beta, exponential, gamma, lognormal, normal, and Weibull distributions. (For some distributions, there are restrictions on the values of the parameters.) Since transformed density rejection is applicable to a large class of distributions, it is sometimes called a *universal* method [see Hörmann et al. (2004)].

8.3 GENERATING CONTINUOUS RANDOM VARIATES

In this section we discuss particular algorithms for generating random variates from several commonly occurring continuous distributions; Sec. 8.4 contains a similar treatment for discrete random variates. Although there may be several different algorithms for generating from a given distribution, we explicitly present

only one technique in each case and provide references for other algorithms that may be better in some sense, e.g., in terms of speed at the expense of increased setup cost and greater complexity. In deciding which algorithm to present, we have tried to choose those that are simple to describe and implement, and are reasonably efficient as well. We also give only exact (up to machine accuracy) methods, as opposed to approximations. If speed is critically important, however, we urge the reader to pursue the various references given for the desired distribution. For definitions of density functions, mass functions, and distribution functions, see Secs. 6.2.2 and 6.2.3.

8.3.1 Uniform

The distribution function of a U(a, b) random variable is easily inverted by solving $u = F(x)$ for x to obtain, for $0 \le u \le 1$,

$$x = F^{-1}(u) = a + (b - a)u$$

Thus, we can use the inverse-transform method to generate X:

1. Generate $U \sim$ U(0, 1).
2. Return $X = a + (b - a)U$.

If many X values are to be generated, the constant $b - a$ should, of course, be computed beforehand and stored for use in the algorithm.

8.3.2 Exponential

The exponential random variable with mean $\beta > 0$ was considered in Example 8.1, where we derived the following inverse-transform algorithm:

1. Generate $U \sim$ U(0, 1).
2. Return $X = -\beta \ln U$.

[Recall that the U in step 2 would be $1 - U$ instead if we wanted a literal version of $X = F^{-1}(U)$ in order to make the correlation between the X's and U's positive.] This is certainly a simple technique and has all the advantages of the inverse-transform method discussed in Sec. 8.2.1. It is also reasonably fast, with most of the computing time's being taken up in evaluating the logarithm. In the experiments of Ahrens and Dieter (1972), this method was the fastest of the four algorithms considered if programming in FORTRAN, with some 72 percent of the time taken up by the logarithm evaluation. If one is willing to program in a lower-level language, however, there are other methods that avoid the logarithm and are faster, although considerably more complex and involving various amounts of preliminary setup; see von Neumann (1951), Marsaglia (1961), and MacLaren, Marsaglia, and Bray (1964). We refer the interested reader to Ahrens and Dieter (1972) and to Fishman (1978, pp. 402–410) for further discussion.

8.3.3 m-Erlang

As discussed in Example 8.5, if X is an m-Erlang random variable with mean β, we can write $X = Y_1 + Y_2 + \cdots + Y_m$, where the Y_i's are IID exponential random variables, each with mean β/m. This led to the convolution algorithm described in Example 8.5. Its efficiency can be improved, however, as follows. If we use the inverse-transform method of Sec. 8.3.2 to generate the exponential Y_i's [$Y_i = (-\beta/m) \ln U_i$, where $U_1, U_2, \ldots, U_m$ are IID U(0, 1) random variates], then

$$X = \sum_{i=1}^{m} Y_i = \sum_{i=1}^{m} \frac{-\beta}{m} \ln U_i = \frac{-\beta}{m} \ln\left(\prod_{i=1}^{m} U_i\right)$$

so that we need to evaluate only one logarithm (rather than m logarithms). Then the statement of the algorithm is as follows:

1. Generate $U_1, U_2, \ldots, U_m$ as IID U(0, 1).
2. Return $X = \dfrac{-\beta}{m} \ln\left(\prod_{i=1}^{m} U_i\right)$.

(Again, one should compute β/m beforehand and store it for repeated use.) This algorithm is really a combination of the composition and inverse-transform methods.

Since we must generate m random numbers and perform m multiplications, the execution time of the algorithm is approximately proportional to m. Therefore, one might look for an alternative method when m is large. Fortunately, the m-Erlang distribution is a special case of the gamma distribution (with shape parameter α equal to the integer m), so that we can use one of the methods for generating gamma random variates here as well (see Sec. 8.3.4 for discussion of gamma generation). The precise threshold for m beyond which one should switch to general gamma generation will depend on the method used for generating a gamma random variate as well as on languages, compilers, and hardware; preliminary experimentation in one's particular situation might prove worthwhile. [For the gamma generator of Sec. 8.3.4 in the case $\alpha > 1$, timing experiments in Cheng (1977) indicate that using his general gamma generator becomes faster than the above m-Erlang algorithm for $m \ge 10$, approximately.] Another potential problem with using the above algorithm, especially for large m, is that $\prod_{i=1}^{m} U_i$ might get close to zero, which could lead to numerical difficulties when its logarithm is taken.

8.3.4 Gamma

General gamma random variates are more complicated to generate than the three types of random variates considered so far in this section, since the distribution function has no simple closed form for which we could try to find an inverse. First note that given $X \sim$ gamma(α, 1), we can obtain, for any $\beta > 0$, a gamma(α, β) random variate X' by letting $X' = \beta X$, so that it is sufficient to restrict attention

to generating from the gamma(α, 1) distribution. Furthermore, recall that the gamma(1, 1) distribution is just the exponential distribution with mean 1, so that we need consider only $0 < \alpha < 1$ and $\alpha > 1$. Since the available algorithms for generating gamma random variates are for the most part valid in only one of these ranges of α, we shall discuss them separately. [Tadikamalla and Johnson (1981) provide a comprehensive review of gamma variate generation methods that were available at that time.]

We first consider the case $0 < \alpha < 1$. (Note that if $\alpha = 0.5$, we have a rescaled χ_1^2 distribution and X can be easily generated using Example 8.9; the algorithm stated below is nevertheless valid for $\alpha = 0.5$.) Atkinson and Pearce (1976) tested three alterative algorithms for this case, and we present one of them, due to Ahrens and Dieter (1974). [The algorithm of Forsythe (1972) was usually the fastest in the comparisons in Atkinson and Pearce (1976), but it is considerably more complicated.] This algorithm, denoted GS in Ahrens and Dieter (1974), is an acceptance-rejection technique, with majorizing function

$$t(x) = \begin{cases} 0 & \text{if } x \le 0 \\ \dfrac{x^{\alpha-1}}{\Gamma(\alpha)} & \text{if } 0 < x \le 1 \\ \dfrac{e^{-x}}{\Gamma(\alpha)} & \text{if } 1 < x \end{cases}$$

Thus, $c = \int_0^\infty t(x)\,dx = b/[\alpha\Gamma(\alpha)]$, where $b = (e + \alpha)/e > 1$, which yields the density $r(x) = t(x)/c$ as

$$r(x) = \begin{cases} 0 & \text{if } x \le 0 \\ \dfrac{\alpha x^{\alpha-1}}{b} & \text{if } 0 < x \le 1 \\ \dfrac{\alpha e^{-x}}{b} & \text{if } 1 < x \end{cases}$$

Generating a random variate Y with density $r(x)$ can be done by the inverse-transform method; the distribution function corresponding to r is

$$R(x) = \int_0^x r(y)\,dy = \begin{cases} \dfrac{x^\alpha}{b} & \text{if } 0 \le x \le 1 \\ 1 - \dfrac{\alpha e^{-x}}{b} & \text{if } 1 < x \end{cases}$$

which can be inverted to obtain

$$R^{-1}(u) = \begin{cases} (bu)^{1/\alpha} & \text{if } u \le \dfrac{1}{b} \\ -\ln \dfrac{b(1-u)}{\alpha} & \text{otherwise} \end{cases}$$

Thus, to generate Y with density r, we first generate $U_1 \sim \text{U}(0, 1)$. If $U_1 \le 1/b$, we set $Y = (bU_1)^{1/\alpha}$; in this case, $Y \le 1$. Otherwise, if $U_1 > 1/b$, set $Y = -\ln[b(1 - U_1)/\alpha]$, which will be greater than 1. Noting that

$$\frac{f(Y)}{t(Y)} = \begin{cases} e^{-Y} & \text{if } 0 \le Y \le 1 \\ Y^{\alpha-1} & \text{if } 1 < Y \end{cases}$$

we obtain the final algorithm [$b = (e + \alpha)/e$ must be computed beforehand]:

1. Generate $U_1 \sim \text{U}(0, 1)$, and let $P = bU_1$. If $P > 1$, go to step 3. Otherwise, proceed to step 2.
2. Let $Y = P^{1/\alpha}$, and generate $U_2 \sim \text{U}(0, 1)$. If $U_2 \le e^{-Y}$, return $X = Y$. Otherwise, go back to step 1.
3. Let $Y = -\ln[(b - P)/\alpha]$ and generate $U_2 \sim \text{U}(0, 1)$. If $U_2 \le Y^{\alpha-1}$, return $X = Y$. Otherwise, go back to step 1.

We now consider the case $\alpha > 1$, where there are several good algorithms. In view of timing experiments by Schmeiser and Lal (1980) and Cheng and Feast (1979), we will present a modified acceptance-rejection method due to Cheng (1977), who calls this the GB algorithm. This algorithm has a "capped" execution time; i.e., its execution time is bounded as $\alpha \to \infty$ and in fact appears to become faster as α grows. (The modification of the general acceptance-rejection method consists of adding a faster pretest for acceptance.) To obtain a majorizing function $t(x)$, first let $\lambda = \sqrt{2\alpha - 1}$, $\mu = \alpha^\lambda$, and $c = 4\alpha^\alpha e^{-\alpha}/[\lambda\Gamma(\alpha)]$. Then define $t(x) = cr(x)$, where

$$r(x) = \begin{cases} \dfrac{\lambda\mu x^{\lambda-1}}{(\mu + x^\lambda)^2} & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

The distribution function corresponding to the density $r(x)$ is

$$R(x) = \begin{cases} \dfrac{x^\lambda}{\mu + x^\lambda} & \text{if } x \ge 0 \\ 0 & \text{otherwise} \end{cases}$$

which is easily inverted to obtain

$$R^{-1}(u) = \left(\frac{\mu u}{1 - u}\right)^{1/\lambda} \qquad \text{for } 0 < u < 1$$

To verify that $t(x)$ indeed majorizes $f(x)$, see Cheng (1977). Note that this is an example of obtaining a majorizing function by first specifying a known distribution [$R(x)$ is actually the log-logistic distribution function (see Sec. 8.3.11) with shape parameter λ, scale parameter $\mu^{1/\lambda}$, and location parameter 0] and then rescaling the density $r(x)$ to majorize $f(x)$. Thus, we use the inverse-transform method to generate Y with density r. After adding an advantageous pretest for acceptance and streamlining for computational efficiency, Cheng (1977) recommends the

following algorithm (the prespecified constants are $a = 1/\sqrt{2\alpha - 1}$, $b = \alpha - \ln 4$, $q = \alpha + 1/a$, $\theta = 4.5$, and $d = 1 + \ln \theta$):

1. Generate U_1 and U_2 as IID U(0, 1).
2. Let $V = a \ln [U_1/(1 - U_1)]$, $Y = \alpha e^V$, $Z = U_1^2 U_2$, and $W = b + qV - Y$.
3. If $W + d - \theta Z \geq 0$, return $X = Y$. Otherwise, proceed to step 4.
4. If $W \geq \ln Z$, return $X = Y$. Otherwise, go back to step 1.

Step 3 is the added pretest, which (if passed) avoids computing the logarithm in the regular acceptance-rejection test in step 4. (If step 3 were removed, the algorithm would still be valid and would just be the literal acceptance-rejection method.)

As mentioned above, there are several other good algorithms that could be used when $\alpha > 1$. Schmeiser and Lal (1980) present another acceptance-rejection method with $t(x)$ piecewise linear in the "body" of $f(x)$ and exponential in the tails; their algorithm was roughly twice as fast as the one we chose to present above, for α ranging from 1.0001 through 1000. However, their algorithm is more complicated and requires additional time to set up the necessary constants for a given value of α. This is typical of the tradeoffs the analyst must consider in choosing among alternative variate-generation algorithms.

Finally, we consider direct use of the inverse-transform method to generate gamma random variates. Since neither the gamma distribution function nor its inverse has a simple closed form, we must resort to numerical methods. Best and Roberts (1975) give a numerical procedure for inverting the distribution function of a chi-square random variable with degrees of freedom that need not be an integer, so is applicable for gamma generation for any $\alpha > 0$. [If $Y \sim \chi^2_\nu$ where $\nu > 0$ need not be an integer, then $Y \sim$ gamma(ν/2, 2). If we want $X \sim$ gamma(α, 1), first generate $Y \sim \chi^2_{2\alpha}$, and then return $X = Y/2$.] An IMSL routine [Visual Numerics, Inc. (2004)] is available to invert the chi-square distribution function. Press et al. (1992, sec. 6.2) give C codes to evaluate the chi-square distribution function (a reparameterization of what's known as the *incomplete gamma function*), which would then have to be numerically inverted by a root-finding algorithm, which they discuss in their chap. 9.

8.3.5 Weibull

The Weibull distribution function is easily inverted to obtain

$$F^{-1}(u) = \beta[-\ln(1-u)]^{1/\alpha}$$

which leads to the following inverse-transform algorithm:

1. Generate $U \sim$ U(0, 1).
2. Return $X = \beta(-\ln U)^{1/\alpha}$.

Again we are exploiting the fact that U and $1 - U$ have the same U(0, 1) distribution, so that in step 2, U should be replaced by $1 - U$ if the literal inverse-transform method is desired. This algorithm can also be justified by noting that if Y has an exponential distribution with mean β^α, then $Y^{1/\alpha} \sim$ Weibull(α, β); see Sec. 6.2.2.

8.3.6 Normal

First note that given $X \sim$ N(0, 1), we can obtain $X' \sim$ N(μ, σ^2) by setting $X' = \mu + \sigma X$, so that we can restrict attention to generating standard normal random variates. Efficiency is important, since the normal density has often been used to provide majorizing functions for acceptance-rejection generation of random variates from other distributions, e.g., Ahrens and Dieter's (1974) gamma and beta generators. Normal random variates can also be transformed directly into random variates from other distributions, e.g., the lognormal. Also, statisticians seeking to estimate empirically, in a Monte Carlo study, the null distribution of a test statistic for normality will need an efficient source of normal random variates. [See, for example, Filliben (1975), Lilliefors (1967), or Shapiro and Wilk (1965).]

One of the early methods for generating N(0, 1) random variates, due to Box and Muller (1958), is evidently still in use despite the availability of much faster algorithms. It does have the advantage, however, of maintaining a one-to-one correspondence between the random numbers used and the N(0, 1) random variates produced; it may thus prove useful for maintaining synchronization in the use of common random numbers or antithetic variates as a variance-reduction technique (see Secs. 11.2 and 11.3). The method simply says to generate U_1 and U_2 as IID U(0, 1), then set $X_1 = \sqrt{-2 \ln U_1} \cos 2\pi U_2$ and $X_2 = \sqrt{-2 \ln U_1} \sin 2\pi U_2$. Then X_1 and X_2 are IID N(0, 1) random variates. Since we obtain the desired random variates in pairs, we could, on odd-numbered calls to the subprogram, actually compute X_1 and X_2 as just described, but return only X_1, saving X_2 for immediate return on the next (even-numbered) call. Thus, we use *two* random numbers to produce *two* N(0, 1) random variates. While this method is valid in principle, i.e., if U_1 and U_2 are truly IID U(0, 1) random variables, there is a serious difficulty if U_1 and U_2 are actually adjacent random numbers produced by a linear congruential generator (see Sec. 7.2), as they might be in practice. Due to the fact that U_2 would depend on U_1 according to the recursion in Eq. (7.1) in Sec. 7.2, it can be shown that the generated variates X_1 and X_2 must fall on a spiral in (X_1, X_2) space, rather than being truly independently normally distributed; see, for example, Bratley, Fox, and Schrage (1987, pp. 223–224). Thus, the Box-Muller method should not be used with a single stream of a linear congruential generator; it might be possible to use separate streams or a composite generator instead, e.g., the combined multiple recursive generator in App. 7B, but one of the methods described below for normal variate generation should probably be used instead.

An improvement to the Box and Muller method, which eliminates the trigonometric calculations and was described in Marsaglia and Bray (1964), has become known as the *polar method*. It relies on a special property of the normal distribution and was found by Atkinson and Pearce (1976) to be between 9 and 31 percent

faster in FORTRAN programming than the Box and Muller method, depending on the machine used. [Ahrens and Dieter (1972) experienced a 27 percent reduction in time.] The polar method, which also generates N(0, 1) random variates in pairs, is as follows:

1. Generate U_1 and U_2 as IID U(0, 1); let $V_i = 2U_i - 1$ for $i = 1, 2$; and let $W = V_1^2 + V_2^2$.
2. If $W > 1$, go back to step 1. Otherwise, let $Y = \sqrt{(-2 \ln W)/W}$, $X_1 = V_1 Y$, and $X_2 = V_2 Y$. Then X_1 and X_2 are IID N(0, 1) random variates.

Since a "rejection" of U_1 and U_2 can occur in step 2 (with probability $1 - \pi/4$, by Prob. 8.12), the polar method will require a random number of U(0, 1) random variates to generate each pair of N(0, 1) random variates. More recently, a very fast algorithm for generating N(0, 1) random variates was developed by Kinderman and Ramage (1976), which is more complicated but required 30 percent less time than the polar method in their FORTRAN experiments.

For direct use of the inverse-transform method in normal generation, one must use a numerical method, since neither the normal distribution function nor its inverse has a simple closed-form expression. Such a method is given by Moro (1995). Also, the IMSL [Visual Numerics, Inc. (2004)] library has routines to invert the standard normal distribution function.

8.3.7 Lognormal

A special property of the lognormal distribution, namely, that if $Y \sim \text{N}(\mu, \sigma^2)$, then $e^Y \sim \text{LN}(\mu, \sigma^2)$, is used to obtain the following algorithm:

1. Generate $Y \sim \text{N}(\mu, \sigma^2)$.
2. Return $X = e^Y$.

To accomplish step 1, any method discussed in Sec. 8.3.6 for normal generation can be used.

Note that μ and σ^2 are *not* the mean and variance of the $\text{LN}(\mu, \sigma^2)$ distribution. In fact, if $X \sim \text{LN}(\mu, \sigma^2)$ and we let $\mu' = E(X)$ and $\sigma'^2 = \text{Var}(X)$, then it turns out that $\mu' = e^{\mu + \sigma^2/2}$ and $\sigma'^2 = e^{2\mu + \sigma^2}(e^{\sigma^2} - 1)$. Thus, if we want to generate a lognormal random variate X with *given* mean $\mu' = E(X)$ and *given* variance $\sigma'^2 = \text{Var}(X)$, we should solve for μ and σ^2 in terms of μ' and σ'^2 first, *before* generating the intermediate normal random variate Y. The formulas are easily obtained as

$$\mu = E(Y) = \ln \frac{\mu'^2}{\sqrt{\mu'^2 + \sigma'^2}}$$

and

$$\sigma^2 = \text{Var}(Y) = \ln\left(1 + \frac{\sigma'^2}{\mu'^2}\right)$$

8.3.8 Beta

First note that we can obtain $X' \sim \text{beta}(\alpha_1, \alpha_2)$ on the interval $[a, b]$ for $a < b$ by setting $X' = a + (b - a)X$, where $X \sim \text{beta}(\alpha_1, \alpha_2)$ on the interval $[0, 1]$, so that it is sufficient to consider only the latter case, which we henceforth call *the* beta(α_1, α_2) distribution.

Some properties of the beta(α_1, α_2) distribution for certain (α_1, α_2) combinations facilitate generating beta random variates. First, if $X \sim \text{beta}(\alpha_1, \alpha_2)$, then $1 - X \sim \text{beta}(\alpha_2, \alpha_1)$, so that we can readily generate a beta(α_2, α_1) random variate if we can obtain a beta(α_1, α_2) random variate easily. One such situation occurs when either α_1 or α_2 is equal to 1. If $\alpha_2 = 1$, for example, then for $0 \le x \le 1$ we have $f(x) = \alpha_1 x^{\alpha_1 - 1}$, so the distribution function is $F(x) = x^{\alpha_1}$, and we can easily generate $X \sim \text{beta}(\alpha_1, 1)$ by the inverse-transform method, i.e., by returning $X = U^{1/\alpha_1}$, for $U \sim \text{U}(0, 1)$. Finally, the beta(1, 1) distribution is simply U(0, 1).

A general method for generating a beta(α_1, α_2) random variate for any $\alpha_1 > 0$ and $\alpha_2 > 0$ is a result of the fact that if $Y_1 \sim \text{gamma}(\alpha_1, 1)$, $Y_2 \sim \text{gamma}(\alpha_2, 1)$, and Y_1 and Y_2 are independent, then $Y_1/(Y_1 + Y_2) \sim \text{beta}(\alpha_1, \alpha_2)$. This leads to the following algorithm:

1. Generate $Y_1 \sim \text{gamma}(\alpha_1, 1)$ and $Y_2 \sim \text{gamma}(\alpha_2, 1)$ independent of Y_1.
2. Return $X = Y_1/(Y_1 + Y_2)$.

Generating the two gamma random variates Y_1 and Y_2 can be done by any appropriate algorithm for gamma generation (see Sec. 8.3.4), so that we must take care to check whether α_1 and α_2 are less than or greater than 1.

This method is quite convenient, in that it is essentially done provided that we have gamma$(\alpha, 1)$ generators for all $\alpha > 0$; its efficiency will, of course, depend on the speed of the chosen gamma generators. There are, however, considerably faster (and more complicated, as usual) algorithms for generating from the beta distribution directly. For $\alpha_1 > 1$ and $\alpha_2 > 1$, Schmeiser and Babu (1980) present a very fast acceptance-rejection method, where the majorizing function is piecewise linear over the center of $f(x)$ and exponential over the tails; a fast acceptance pretest is specified by a piecewise-linear function $b(x)$ that minorizes (i.e., is always below) $f(x)$. If $\alpha_1 < 1$ or $\alpha_2 < 1$ (or both), algorithms for generating beta(α_1, α_2) random variates directly are given by Atkinson and Whittaker (1976, 1979), Cheng (1978), and Jöhnk (1964). Cheng's (1978) method BA is quite simple and is valid as well for any $\alpha_1 > 0, \alpha_2 > 0$ combination; the same is true for the algorithms of Atkinson (1979a) and Jöhnk (1964).

The inverse-transform method for generating beta random variates must rely on numerical methods, as was the case for the gamma and normal distributions. Cran, Martin, and Thomas (1977) give such a method with a FORTRAN program, and IMSL [Visual Numerics, Inc. (2004)] routines are also available. Press et al. (1992, sec. 6.4) give C codes to evaluate the beta distribution function (also known as the *incomplete beta function*), which would then have to be numerically inverted by a root-finding algorithm, which they discuss in their chap. 9.

8.3.9 Pearson Type V

As noted in Sec. 6.2.2, $X \sim \text{PT5}(\alpha, \beta)$ if and only if $1/X \sim \text{gamma}(\alpha, 1/\beta)$, which leads to the following special-property algorithm:

1. Generate $Y \sim \text{gamma}(\alpha, 1/\beta)$.
2. Return $X = 1/Y$.

Any method from Sec. 8.3.4 for gamma generation could be used, taking care to note whether $\alpha < 1$, $\alpha = 1$, or $\alpha > 1$. To use the inverse-transform method, we note from Sec. 6.2.2 that the PT5(α, β) distribution function is $F(x) = 1 - F_G(1/x)$ for $x > 0$, where F_G is the gamma$(\alpha, 1/\beta)$ distribution function. Setting $F(X) = U$ thus leads to $X = 1/F_G^{-1}(1 - U)$ as the literal inverse-transform method, or to $X = 1/F_G^{-1}(U)$ if we want to exploit the fact that $1 - U$ and U have the same U(0, 1) distribution. In any case, we would generally have to use a numerical method to evaluate F_G^{-1}, as discussed in Sec. 8.3.4.

8.3.10 Pearson Type VI

From Sec. 6.2.2, we note that if $Y_1 \sim \text{gamma}(\alpha_1, \beta)$ and $Y_2 \sim \text{gamma}(\alpha_2, 1)$, and Y_1 and Y_2 are independent, then $Y_1/Y_2 \sim \text{PT6}(\alpha_1, \alpha_2, \beta)$; this leads directly to:

1. Generate $Y_1 \sim \text{gamma}(\alpha_1, \beta)$ and $Y_2 \sim \text{gamma}(\alpha_2, 1)$ independent of Y_1.
2. Return $X = Y_1/Y_2$.

Any method from Sec. 8.3.4 for gamma generation could be used, checking whether $\alpha < 1$, $\alpha = 1$, or $\alpha > 1$. To use the inverse-transform method, note from Sec. 6.2.2 that the PT6$(\alpha_1, \alpha_2, \beta)$ distribution function is $F(x) = F_B(x/(x + \beta))$ for $x > 0$, where F_B is the beta(α_1, α_2) distribution function. Setting $F(X) = U$ thus leads to $X = \beta F_B^{-1}(U)/[1 - F_B^{-1}(U)]$, where $F_B^{-1}(U)$ would generally have to be evaluated by a numerical method, as discussed in Sec. 8.3.8.

8.3.11 Log-Logistic

The log-logistic distribution function can be inverted to obtain

$$F^{-1}(u) = \beta\left(\frac{u}{1-u}\right)^{1/\alpha}$$

which leads to the inverse-transform algorithm:

1. Generate $U \sim \text{U}(0, 1)$.
2. Return $X = \beta[U/(1 - U)]^{1/\alpha}$.

8.3.12 Johnson Bounded

$X \sim \text{JSB}(\alpha_1, \alpha_2, a, b)$ if and only if $Z = \alpha_1 + \alpha_2 \ln[(X - a)/(b - X)] \sim \text{N}(0, 1)$, and we can solve this equation for X in terms of Z to get the following special-property algorithm:

1. Generate $Z \sim \text{N}(0, 1)$.
2. Let $Y = \exp[(Z - \alpha_1)/\alpha_2]$.
3. Return $X = (a + bY)/(Y + 1)$.

Any method from Sec. 8.3.6 for standard normal generation can be used to generate Z in step 1.

8.3.13 Johnson Unbounded

$X \sim \text{JSU}(\alpha_1, \alpha_2, \gamma, \beta)$ if and only if

$$Z = \alpha_1 + \alpha_2 \ln\left[\frac{X-\gamma}{\beta} + \sqrt{\left(\frac{X-\gamma}{\beta}\right)^2 + 1}\right] \sim \text{N}(0, 1)$$

and we can solve this equation for X in terms of Z to get the following special-property algorithm:

1. Generate $Z \sim \text{N}(0, 1)$.
2. Let $Y = \exp[(Z - \alpha_1)/\alpha_2]$.
3. Return $X = \gamma + (\beta/2)(Y - 1/Y)$.

Any method from Sec. 8.3.6 for standard normal generation can be used to generate Z in step 1. An alternative statement of the algorithm is $X = \gamma + \beta \sinh[(Z - \alpha_1)/\alpha_2]$ where Z is as in step 1.

8.3.14 Bézier

Random variates from fitted Bézier distributions, as discussed in Sec. 6.9, can be generated by a numerical inverse-transform method given by Wagner and Wilson (1996b), which requires a root-finding algorithm as part of its operation.

8.3.15 Triangular

First notice that if we have $X \sim \text{triang}[0, 1, (m - a)/(b - a)]$, then $X' = a + (b - a)\,X \sim \text{triang}(a, b, m)$, so we can restrict attention to triang$(0, 1, m)$ random variables, where $0 < m < 1$. (For the limiting cases $m = 0$ or $m = 1$, giving rise to a left or right triangle, see Prob. 8.7.) The distribution function is easily inverted to obtain, for $0 \le u \le 1$,

$$F^{-1}(u) = \begin{cases} \sqrt{mu} & \text{if } 0 \le u \le m \\ 1 - \sqrt{(1-m)(1-u)} & \text{if } m < u \le 1 \end{cases}$$

Therefore, we can state the following inverse-transform algorithm for generating $X \sim \text{triang}(0, 1, m)$:

1. Generate $U \sim \text{U}(0, 1)$.
2. If $U \le m$, return $X = \sqrt{mU}$. Otherwise, return $X = 1 - \sqrt{(1-m)(1-U)}$.

(Note that if $U > c$ in step 2, we *cannot* replace the $1 - U$ in the formula for X by U. Why?) For an alternative method of generating a triangular random variate (by composition), see Prob. 8.13.

8.3.16 Empirical Distributions

In this section we give algorithms for generating random variates from the continuous empirical distribution functions F and G defined in Sec. 6.2.4. In both cases, the inverse-transform approach can be used.

First suppose that we have the original individual observations, which we use to define the empirical distribution function $F(x)$ given in Sec. 6.2.4 (see also Fig. 6.24). Although an inverse-transform algorithm might at first appear to involve some kind of a search, the fact that the "corners" of F occur precisely at levels $0, 1/(n-1), 2/(n-1), \ldots, (n-2)/(n-1)$, and 1 allows us to avoid an explicit search. We leave it to the reader to verify that the following algorithm *is* the inverse-transform method:

1. Generate $U \sim \mathrm{U}(0, 1)$, let $P = (n-1)U$, and let $I = \lfloor P \rfloor + 1$.
2. Return $X = X_{(I)} + (P - I + 1)(X_{(I+1)} - X_{(I)})$.

Note that the $X_{(i)}$'s must be stored and that storing a separate array containing the values of $X_{(I+1)} - X_{(I)}$ would eliminate a subtraction in step 2. Also, the values of X generated will always be between $X_{(1)}$ and $X_{(n)}$; this limitation is a possible disadvantage of specifying an empirical distribution in this way. The lack of a search makes the marginal execution time of this algorithm essentially independent of n, although large n entails more storage and setup time for sorting the X_i's.

Now suppose that our data are grouped; that is, we have k adjacent intervals $[a_0, a_1), [a_1, a_2), \ldots, [a_{k-1}, a_k]$, and the jth interval contains n_j observations, with $n_1 + n_2 + \cdots + n_k = n$. In this case, we defined an empirical distribution function $G(x)$ in Sec. 6.2.4 (see also Fig. 6.25), and the following inverse-transform algorithm generates a random variate with this distribution:

1. Generate $U \sim \mathrm{U}(0, 1)$.
2. Find the nonnegative integer J $(0 \le J \le k-1)$ such that $G(a_J) \le U < G(a_{J+1})$, and return $X = a_J + [U - G(a_J)](a_{J+1} - a_J)/[G(a_{J+1}) - G(a_J)]$.

Note that the J found in step 2 satisfies $G(a_J) < G(a_{J+1})$, so that no X can be generated in an interval for which $n_j = 0$. (Also, it is clear that $a_0 \le X \le a_k$.) Determining J in step 2 could be done by a straightforward left-to-right search or by a search starting with the value of j for which $G(a_{j+1}) - G(a_j)$ is largest, then next largest, etc. As an alternative that avoids the search entirely (at the expense of extra storage), we could initially define a vector $(m_1, m_2, \ldots, m_n)$ by setting the first n_1 m_i's to 0, the next n_2 m_i's to 1, etc., with the last n_k m_i's being set to $k-1$. (If some n_j is 0, no m_i's are set to $j-1$. For example, if $k \ge 3$ and $n_1 > 0$, $n_2 = 0$, and $n_3 > 0$, the first n_1 m_i's are set to 0 and the *next* n_3 m_i's are set to 2.) Then the value of J in step 2 can be determined by setting $L = \lfloor nU \rfloor + 1$ and letting $J = m_L$. Whether or not this is worthwhile depends on the particular characteristics of the data and on the importance of any computational speed that might be gained relative to the extra storage and programming effort. Finally, Chen and Asau (1974) give another method for determining J in step 2, based on preliminary calculations that reduce the range of search for a given U; it requires only 10 extra memory locations. (Their treatment is for a discrete empirical distribution function but can also be applied to the present case.)

The empirical/exponential distribution mentioned briefly in Sec. 6.2.4 can also be inverted so that the inverse-transform method can be used; an explicit algorithm is given in Bratley, Fox, and Schrage (1987, p. 151).

8.4 GENERATING DISCRETE RANDOM VARIATES

This section discusses particular algorithms for generating random variates from various discrete distributions that might be useful in a simulation study. As in Sec. 8.3, we usually present for each distribution one algorithm that is fairly simple to implement and reasonably efficient. References will be made to alternative algorithms that might be faster, usually at the expense of greater complexity.

The discrete inverse-transform method, as described in Sec. 8.2.1, can be used for any discrete distribution, whether the range of possible values is finite or (countably) infinite. Many of the algorithms presented in this section *are* the discrete inverse-transform method, although in some cases this fact is very well disguised due to the particular way the required search is performed, which often takes advantage of the special form of the probability mass function. As was the case for continuous random variates, however, the inverse-transform method may not be the most efficient way to generate a random variate from a given distribution.

One other general approach should be mentioned here, which can be used for generating *any* discrete random variate having a *finite* range of values. This is the *alias method,* developed by Walker (1977) and refined by Kronmal and Peterson (1979); it is very general and efficient, but it does require some initial setup as well as extra storage. We discuss the alias method in greater detail in Sec. 8.4.3, but the reader should keep in mind that it is applicable to *any* discrete distribution with a finite range (such as the binomial). For an infinite range, the alias method can be used indirectly in conjunction with the general composition approach (see Sec. 8.2.2); this is also discussed in Sec. 8.4.3.

In addition to the alias method, there are some other general discrete-variate generation ideas; see, for example, Shanthikumar (1985) and Peterson and Kronmal (1983).

A final comment concerns the apparent loss of generality in considering below only distributions that have range $S_n = \{0, 1, 2, \ldots, n\}$ or $S = \{0, 1, 2, \ldots\}$, which may appear to be more restrictive than our original definition of a discrete random variable having general range $T_n = \{x_1, x_2, \ldots, x_n\}$ or $T = \{x_1, x_2, \ldots\}$. However, no generality is actually lost. If we really want a random variate X with mass function $p(x_i)$ and general range T_n (or T), we can first generate a random variate I with range S_{n-1} (or S) such that $P(I = i - 1) = p(x_i)$ for $i = 1, 2, \ldots, n$

(or $i = 1, 2, \ldots$). Then the random variate $X = x_{I+1}$ is returned and has the desired distribution. (Given I, x_{I+1} could be determined from a stored table of the x_i's or from a formula that computes x_i as a function of i.)

8.4.1 Bernoulli

The following algorithm is quite intuitive and is equivalent to the inverse-transform method (if the roles of U and $1 - U$ are reversed):

1. Generate $U \sim \mathrm{U}(0, 1)$.
2. If $U \le p$, return $X = 1$. Otherwise, return $X = 0$.

8.4.2 Discrete Uniform

Again, the straightforward intuitive algorithm given below is (exactly) the inverse-transform method:

1. Generate $U \sim \mathrm{U}(0, 1)$.
2. Return $X = i + \lfloor (j - i + 1)U \rfloor$.

Note that no search is required. The constant $j - i + 1$ should, of course, be computed ahead of time and stored.

8.4.3 Arbitrary Discrete Distribution

Consider the very general situation in which we have *any* probability mass function $p(0), p(1), p(2), \ldots$ on the nonnegative integers S, and we want to generate a discrete random variate X with the corresponding distribution. The $p(i)$'s could have been specified theoretically by some distributional form or empirically from a data set directly. The case of finite range S_n is included here by setting $p(i) = 0$ for all $i \ge n + 1$. (Note that this formulation includes *every* special discrete distribution form.)

The direct inverse-transform method, for either the finite- or infinite-range case, is as follows (define the empty sum to be 0):

1. Generate $U \sim \mathrm{U}(0, 1)$.
2. Return the nonnegative integer $X = I$ satisfying

$$\sum_{j=0}^{I-1} p(j) \le U < \sum_{j=0}^{I} p(j)$$

Note that this algorithm will never return a value $X = i$ for which $p(i) = 0$, since the strict inequality between the two summations in step 2 would be impossible. Step 2 does require a search, which may be time consuming. As an alternative, we could initially sort the $p(i)$'s into decreasing order so that the search would be likely to terminate after a smaller number of comparisons; see Prob. 8.2 for an example.

Due to the generality of the present situation, we present three other methods that are useful when the desired random variable has *finite* range S_n. The first of these methods assumes that each $p(i)$ is exactly equal to a q-place decimal; for example we take the case $q = 2$, so that $p(i)$ is of the form $0.01k_i$ for some integer $k_i \in \{0, 1, \ldots, 100\}$ $(i = 0, 1, 2, \ldots, n)$, and $\sum_{i=0}^{n} k_i = 100$. We initialize a vector $(m_1, m_2, \ldots, m_{100})$ by setting the first k_0 m_j's to 0, the next k_1 m_j's to 1, etc., and the last k_n m_j's to n. (If $k_i = 0$ for some i, no m_j's are set to i.) Then an algorithm for generating the desired random variate X is as follows:

1. Generate $J \sim \mathrm{DU}(1, 100)$.
2. Return $X = m_J$.

(See Sec. 8.4.2 to accomplish step 1.) Note that this method requires 10^q extra storage locations and an array reference in step 2; it *is*, however, the inverse-transform method provided that J is generated by the algorithm in Sec. 8.4.2. If three or four decimal places are needed to specify the $p(i)$'s exactly, the value of 100 in step 1 would be replaced by 1000 or 10,000, respectively, and the storage requirements would also grow by one or two orders of magnitude. Even if the $p(i)$'s are not *exactly* q-place decimals for some small value of q, the analyst might be able to obtain sufficient accuracy by rounding the $p(i)$'s to the nearest hundredth or thousandth; this is an attractive alternative especially when the $p(i)$'s are proportions obtained directly from data, and may not be accurate beyond two or three decimal places anyway. When rounding the $p(i)$'s, however, it is important to remember that they must sum exactly to 1.

The above idea is certainly fast, but it could require large tables if we need high precision in the probabilities. Marsaglia (1963) proposed another kind of table-based algorithm requiring less storage and only slightly more time. For example, consider the distribution

$$p(0) = 0.15, \qquad p(1) = 0.20, \qquad p(2) = 0.37, \qquad p(3) = 0.28$$

Then the idea of the preceding paragraph would require a vector of length 100 to store 15 0s, 20 1s, 37 2s, and 28 3s. Instead, define *two* vectors—one for the "tenths" place and the other for the "hundredths" place. To fill up the tenths vector, look only at the tenths place in the probabilities, and put in that many copies of the associated i (for $i = 0, 1, 2, 3$), so we would take one 0, two 1s, three 2s, and two 3s to get

0 1 1 2 2 2 3 3

Similarly, the hundredths vector is

0 0 0 0 0 2 2 2 2 2 2 2 3 3 3 3 3 3 3

corresponding to the hundredths place in the probabilities; thus, there are 28 storage locations in all (as opposed to 100 for the earlier table method). To generate an X, pick the tenths vector with probability equal to one-tenth the sum of the digits in the probabilities' tenths places, i.e., with probability

$$\frac{1 + 2 + 3 + 2}{10} = 0.8$$

and then choose one of the eight members of the tenths vector at random (equiprobably) as the returned X. On the other hand, we choose the hundredths vector with probability equal to $\frac{1}{100}$ of the sum of the digits in the hundredths place in the original probabilities, i.e., with probability

$$\frac{5 + 0 + 7 + 8}{100} = 0.2$$

and then choose one of the 20 entries in the hundredths vector at random to return as the value of X. It is easy to see that this method is valid; for example,

$$\begin{aligned} P(X = 2) &= P(X = 2 \mid \text{choose tenths vector})P(\text{choose tenths vector}) \\ &\quad + P(X = 2 \mid \text{choose hundredths vector})P(\text{choose hundredths vector}) \\ &= \tfrac{3}{8}(0.8) + \tfrac{7}{20}(0.2) \\ &= 0.37 \end{aligned}$$

as required. The storage advantage of Marsaglia's tables becomes more marked as the number of decimal places in the probabilities increases; in his original example there were three-place decimals, so the direct table method of the preceding paragraph would require 1000 storage locations; the three vectors (tenths, hundredths, and thousandths) in this example required only 91 locations.

The third attractive technique to use when X has range S_n is the alias method mentioned earlier. The method requires that we initially calculate two arrays of length $n + 1$ each, from the given $p(i)$'s. The first array contains what are called the *cutoff values* $F_i \in [0, 1]$ for $i = 0, 1, \ldots, n$, and the second array gives the *aliases* $L_i \in S_n$ for $i = 0, 1, \ldots, n$; two algorithms for computing valid cutoff values and aliases from the $p(i)$'s are given in App. 8B. (The cutoff values and aliases are not unique, and indeed the two algorithms in App. 8B may produce different results for the same distribution; both will result in a valid variate-generation algorithm, however.) Then the alias method is as follows:

1. Generate $I \sim \text{DU}(0, n)$ and $U \sim \text{U}(0, 1)$ independent of I.
2. If $U \le F_I$, return $X = I$. Otherwise, return $X = L_I$.

Thus, step 2 involves a kind of "rejection," but upon rejecting I we need *not* start over but only return I's alias L_I, rather than I itself. The cutoff values are seen to be the probabilities with which we return I rather than its alias. There is only one comparison needed to generate each X, and we need exactly two random numbers for each X if I is generated as in Sec. 8.4.2. (See Prob. 8.17 for a way to accomplish step 1 with only *one* random number.) Although the setup is not complicated, storing the cutoffs and aliases does require $2(n + 1)$ extra storage locations; Kronmal and Peterson (1979) discuss a way to cut the storage in half (see Prob. 8.18). In any case, storage is of order n, which is regarded as the principal weakness of the alias method if n could be very large.

EXAMPLE 8.13. Consider a random variable on $S_3 = \{0, 1, 2, 3\}$ with probability mass function $p(0) = 0.1$, $p(1) = 0.4$, $p(2) = 0.2$, and $p(3) = 0.3$. Applying the first algorithm in App. 8B leads to the following setup:

i	0	1	2	3
$p(i)$	0.1	0.4	0.2	0.3
F_i	0.4	0.0	0.8	0.0
L_i	1	1	3	3

For instance, if step 1 of the algorithm produces $I = 2$, the probability is $F_2 = 0.8$ that we would keep $X = I = 2$, and with probability $1 - F_2 = 0.2$ we would return $X = L_2 = 3$ instead. Thus, since 2 is not the alias of anything else (i.e., none of the other L_i's is equal to 2), the algorithm returns $X = 2$ if and only if $I = 2$ in step 1 and $U \le 0.8$ in step 2, so that

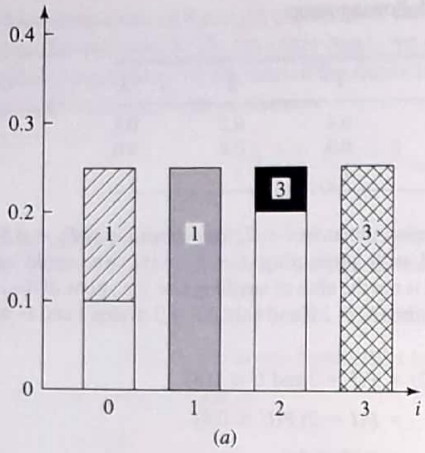
$$\begin{aligned} P(X = 2) &= P(I = 2 \text{ and } U \le 0.8) \\ &= P(I = 2)\,P(U \le 0.8) \\ &= 0.25 \times 0.8 \\ &= 0.2 \end{aligned}$$

which is equal to $p(2)$, as desired. (The second equality in the above follows since U and I are generated independently.) On the other hand, the algorithm can return $X = 3$ in two different (and mutually exclusive) ways: if $I = 3$, then since $F_3 = 0$ we will always return $X = L_3 = 3$; and if $I = 2$ we will return $X = L_2 = 3$ with probability $1 - F_2 = 0.2$. Thus,

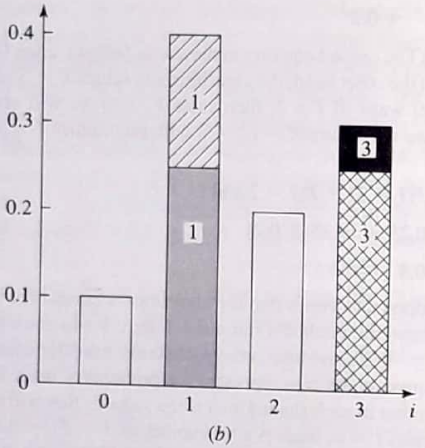
$$\begin{aligned} P(X = 3) &= P(I = 3) + P(I = 2 \text{ and } U > F_2) \\ &= 0.25 + (0.25 \times 0.2) \\ &= 0.3 \end{aligned}$$

which is $p(3)$. The reader is encouraged to verify that the algorithm is correct for $i = 0$ and 1 as well. Figure 8.14 illustrates the method's rationale. Figure 8.14*a* shows bars whose (total) height is $1/(n + 1) = 0.25$, and thus is the probability mass function of I generated in step 1. The shaded areas in the bars represent the probability mass that is moved by the method, and the number in each shaded area is the value L_i that will be returned as X. Thus, if step 1 generates $I = 0$, there is a probability of $1 - F_0 = 0.6$ that this I will be changed into its alias, $L_0 = 1$, for the returned X; the shaded area in the bar above 0 is of height $0.6 \times 0.25 = 0.15$, or 60 percent of that bar. Similarly, the fraction $1 - F_2 = 0.2$ of the 0.25-high bar (resulting in a shaded area of height $0.2 \times 0.25 = 0.05$) above 2 represents the chance that a generated $I = 2$ will be changed into $X = L_2 = 3$. Note that the entire bars above 1 and 3 are shaded, since F_1 and F_3 are both zero; however, the indicated values are their own aliases, so they do not really get moved. Figure 8.14*b* shows the probability mass function of the returned X after the shaded areas (probabilities) are moved to their destination values, and it is seen to equal the desired probabilities $p(i)$.

Although the alias method is limited to discrete random variables with a finite range, it can be used indirectly for discrete distributions with an infinite range, such as the geometric, negative binomial, or Poisson, by combining it with the general composition method. For example, if X can be any nonnegative integer, we can examine the $p(i)$'s to find an n such that $q = \sum_{i=0}^{n} p(i)$ is close to 1, so that the



(a)



(b)

FIGURE 8.14
Setup for the alias method in Example 8.13.

probability is high that $X \in S_n$. Since for any i we can write

$$p(i) = q\left[\frac{p(i)}{q} I_{S_n}(i)\right] + (1 - q)\left\{\frac{p(i)}{1 - q}[1 - I_{S_n}(i)]\right\}$$

we obtain the following general algorithm:

1. Generate $U \sim \mathrm{U}(0, 1)$. If $U \le q$, go to step 2. Otherwise, go to step 3.
2. Use the alias method to return X on S_n with probability mass function $p(i)/q$ for $i = 0, 1, \ldots, n$.
3. Use any other method to return X on $\{n + 1, n + 2, \ldots\}$ with probability mass function $p(i)/(1 - q)$ for $i = n + 1, n + 2, \ldots$.

In step 3, we could use the inverse-transform method, for example. Since n was chosen to make q close to 1, we would expect to avoid step 3 most of the time.

Finally, we note that all the table-based methods discussed above, as well as the alias method, require some effort in an initial setup stage. Thus, they could be unattractive if the probability mass function changes frequently over time as the simulation proceeds. An efficient method for general discrete-variate generation in this case was developed by Rajasekaran and Ross (1993).

8.4.4 Binomial

To generate a bin(t, p) random variate, recall from Sec. 6.2.3 that the sum of t IID Bernoulli(p) random variables has the bin(t, p) distribution. This relation leads to the following convolution algorithm:

1. Generate $Y_1, Y_2, \ldots, Y_t$ as IID Bernoulli(p) random variates.
2. Return $X = Y_1 + Y_2 + \cdots + Y_t$.

Since the execution time of this algorithm is proportional to t, we might want to look for an alternative if t is large. One possibility would be the direct inverse-transform method with an efficient search. Another alternative is the alias method (see Sec. 8.4.3), since the range of X is finite. Finally, algorithms specific to the binomial distribution that are efficient for large t are discussed by Ahrens and Dieter (1974) and Kachitvichyanukul and Schmeiser (1988).

8.4.5 Geometric

The following algorithm is equivalent to the inverse-transform method if we replace U by $1 - U$ in step 2 (see Prob. 8.14):

1. Generate $U \sim \mathrm{U}(0, 1)$.
2. Return $X = \lfloor \ln U / \ln (1 - p) \rfloor$.

The constant $\ln (1 - p)$ should, of course, be computed beforehand. If p is near 0, $\ln (1 - p)$ will also be near zero, so that double-precision arithmetic should be considered to avoid excessive roundoff error in the division in step 2. For p near 1, $\ln (1 - p)$ will be a large negative number, which also could cause numerical difficulties; fortunately, for large p it is more efficient to use an altogether different algorithm based on the relationship between geometric and Bernoulli random variables described in Sec. 6.2.3 (see Prob. 8.14).

8.4.6 Negative Binomial

The relation between the negbin(s, p) and geom(p) distributions in Sec. 6.2.3 leads to the following convolution algorithm:

1. Generate $Y_1, Y_2, \ldots, Y_s$ as IID geom(p) random variates.
2. Return $X = Y_1 + Y_2 + \cdots + Y_s$.

This is simple, but its execution time is proportional to s. For large s, consideration might be given to an alternative method discussed in Fishman (1978), which makes

use of a special relationship between the negative binomial, gamma, and Poisson distributions; its efficiency depends on the ability to generate rapidly from the gamma and Poisson distributions. Other alternatives are discussed in Ahrens and Dieter (1974).

8.4.7 Poisson

Our algorithm for generating Poisson(λ) random variates is based essentially on the relationship between the Poisson(λ) and expo($1/\lambda$) distributions stated in Sec. 6.2.3. The algorithm is as follows:

1. Let $a = e^{-\lambda}$, $b = 1$, and $i = 0$.
2. Generate $U_{i+1} \sim$ U(0, 1) and replace b by bU_{i+1}. If $b < a$, return $X = i$. Otherwise, go to step 3.
3. Replace i by $i + 1$ and go back to step 2.

The algorithm is justified by noting that $X = i$ if and only if

$$\sum_{j=1}^{i} Y_j \le 1 < \sum_{j=1}^{i+1} Y_j$$

where $Y_j = (-1/\lambda) \ln U_j \sim \text{expo}(1/\lambda)$ and the Y_j's are independent. That is, $X = \max\{i: \sum_{j=1}^{i} Y_j \le 1\}$, so that $X \sim$ Poisson(λ) by the first comment in the description of the Poisson distribution in Table 6.4.

Unfortunately, this algorithm becomes slow as λ increases, since a large λ means that $a = e^{-\lambda}$ is smaller, requiring more executions of step 2 to bring the cumulative product of the U_{i+1}'s down under a. [In fact, since X is 1 less than the number of U_{i+1}'s required, the expected number of executions of step 2 is $E(X) + 1 = \lambda + 1$, so that execution time grows with λ in an essentially linear fashion.] One alternative would be to use the alias method in concert with the composition approach (since the range of X is infinite), as described in Sec. 8.4.3. Another possibility would be the inverse-transform method with an efficient search. Atkinson (1979b, 1979c) examined several such search procedures and reported that an indexed search similar to the method of Chen and Asau (1974), discussed earlier in Sec. 8.3.16, performed well. (This search procedure, called PQM by Atkinson, requires a small amount of setup and extra storage but is still quite simple to implement.) Other fast methods of generating Poisson variates are given by Devroye (1981) and by Schmeiser and Kachitvichyanukul (1981).

8.5 GENERATING RANDOM VECTORS, CORRELATED RANDOM VARIATES, AND STOCHASTIC PROCESSES

So far in this chapter we have really considered generation of only a single random variate at a time from various *univariate distributions*. Applying one of these algorithms repeatedly with independent sets of random numbers produces a sequence of IID random variates from the desired distribution.

In some simulation models, however, we may want to generate a random *vector* $\mathbf{X} = (X_1, X_2, \ldots, X_d)^T$ from a specified *joint* (or *multivariate*) *distribution*, where the individual components of the vector might not be independent. (A^T denotes the transpose of a vector or matrix A.) Even if we cannot specify the exact, full joint distribution of $X_1, X_2, \ldots, X_d$, we might want to generate them so that the individual X_i's have specified univariate distributions (called the *marginal distributions* of the X_i's) and so that the correlations, ρ_{ij}, between X_i and X_j are specified by the modeler. In Sec. 6.10 we discussed the need for modeling these situations, and in this section we give examples of methods for generating such correlated random variates and processes in some specific cases. There are several other problems related to generating correlated random variates that we do not discuss explicitly, e.g., generating from a multivariate exponential distribution; we refer the reader to Johnson (1987), Johnson, Wang, and Ramberg (1984), Fishman (1973a, 1978), Mitchell and Paulson (1979), Marshall and Olkin (1967), and Devroye (1997).

8.5.1 Using Conditional Distributions

Suppose that we have a fully specified joint distribution function $F_{X_1, X_2, \ldots, X_d}(x_1, x_2, \ldots, x_d)$ from which we would like to generate a random vector $\mathbf{X} = (X_1, X_2, \ldots, X_d)^T$. Also assume that for $i = 2, 3, \ldots, d$ we can obtain the *conditional distribution* of X_i given that $X_j = x_j$ for $j = 1, 2, \ldots, i - 1$; denote the conditional distribution function by $F_i(x_i | x_1, x_2, \ldots, x_{i-1})$. [See any probability text, such as Mood, Graybill, and Boes (1974, chap. IV) or Ross (2003, chap. 3) for a discussion of conditional distributions.] In addition, let $F_{X_i}(x_i)$ be the marginal distribution function of X_i for $i = 1, 2, \ldots, d$. Then a general algorithm for generating a random vector $\mathbf{X}$ with joint distribution function $F_{X_1, X_2, \ldots, X_d}$ is as follows:

1. Generate X_1 with distribution function F_{X_1}.
2. Generate X_2 with distribution function $F_2(\cdot | X_1)$.
3. Generate X_3 with distribution function $F_3(\cdot | X_1, X_2)$.

.
.
.

d. Generate X_d with distribution function $F_d(\cdot | X_1, X_2, \ldots, X_{d-1})$.
***d* + 1**. Return $\mathbf{X} = (X_1, X_2, \ldots, X_d)^T$.

Note that in steps 2 through d the conditional distributions used are those with the previously generated X_i's; for example, if x_1 is the value generated for X_1 in step 1, the conditional distribution function used in step 2 is $F_2(\cdot | x_1)$, etc. Proof of the validity of this algorithm relies on the definition of conditional distributions and is left to the reader.

As general as this approach may be, its practical utility is probably quite limited. Not only is specification of the entire joint distribution required, but also derivation of all the required marginal and conditional distributions must be