Chapter 3

Methods for Generating Random Variables

3.1 Introduction

One of the fundamental tools required in computational statistics is the ability to simulate random variables from specified probability distributions. On this topic many excellent references are available. On the general subject of methods for generating random variates from specified probability distributions, readers are referred to [72, 97, 117, 119, 120, 160, 234, 240, 251, 255]. On specific topics, also see [3, 4, 33, 43, 71, 101, 161, 165, 195].

In the simplest case, to simulate drawing an observation at random from a finite population, a method of generating random observations from the discrete uniform distribution is required. Therefore, a suitable generator of uniform pseudo-random numbers is essential. Methods for generating random variates from other probability distributions all depend on the uniform random number generator.

In this text we assume that a suitable uniform pseudo-random number generator is available. Refer to the help topic for `.Random.seed` or `RNGkind` for details about the default random number generator in R. For reference about different types of random number generators and their properties see Gentle [117, 119] and Knuth [169].

The uniform pseudo-random number generator in R is `runif`. To generate a vector of n (pseudo) random numbers between 0 and 1, use `runif(n)`. Throughout this text, whenever computer generated random numbers are mentioned, it is understood that these are pseudo-random numbers. To generate n random Uniform(a, b) numbers, use `runif(n, a, b)`. To generate an n by m matrix of random numbers between 0 and 1, use `matrix(runif(n*m), nrow=n, ncol=m)` or `matrix(runif(n*m), n, m)`.

In the examples of this chapter, several functions are given for generating random variates from continuous and discrete probability distributions. Generators for many of these distributions are available in R (e.g., `rbeta`, `rgeom`, `rchisq`, etc.), but the methods presented below are general and apply to many other types of distributions. These methods are also applicable for external libraries, stand-alone programs, or nonstandard simulation problems.

Most of the examples include a comparison of the generated sample with the theoretical distribution of the sampled population. In some examples, histograms, density curves, or QQ plots are constructed. In other examples summary statistics such as sample moments, sample percentiles, or the empirical distribution are compared with the corresponding theoretical values. These are informal approaches to check the implementation of an algorithm for simulating a random variable.

Example 3.1 (Sampling from a finite population)**.** The sample function can be used to sample from a finite population, with or without replacement.

```
> #toss some coins
> sample(0:1, size = 10, replace = TRUE)
[1] 0 1 1 1 0 1 1 1 1 0

> #choose some lottery numbers
> sample(1:100, size = 6, replace = FALSE)
[1] 51 89 26 99 74 73

> #permuation of letters a-z
> sample(letters)
[1] "d" "n" "k" "x" "s" "p" "j" "t" "e" "b" "g"
    "a" "m" "y" "i" "v" "l" "r" "w" "q" "z"
[22] "u" "h" "c" "f" "o"

> #sample from a multinomial distribution
> x <- sample(1:3, size = 100, replace = TRUE,
              prob = c(.2, .3, .5))
> table(x)
x
 1  2  3
17 35 48
```

◇

Random Generators of Common Probability Distributions in R

In the sections that follow, various methods of generating random variates from specified probability distributions are presented. Before discussing those methods, however, it is useful to summarize some of the probability functions available in R. The probability mass function (pmf) or density (pdf), cumulative distribution function (cdf), quantile function, and random generator of many commonly used probability distributions are available. For example, four functions are documented in the help topic `Binomial`:

```
dbinom(x, size, prob, log = FALSE)
pbinom(q, size, prob, lower.tail = TRUE, log.p = FALSE)
qbinom(p, size, prob, lower.tail = TRUE, log.p = FALSE)
rbinom(n, size, prob)
```

The same pattern is applied to other probability distributions. In each case, the abbreviation for the name of the distribution is combined with first letter `d` for density or pmf, `p` for cdf, `q` for quantile, or `r` for random generation from the distribution.

A partial list of available probability distributions and parameters is given in Table 3.1. For a complete list, refer to the R documentation [294, Ch. 8]. In addition to the parameters listed, some of the functions take optional `log`, `lower.tail`, or `log.p` arguments, and some take an optional `ncp` (noncentrality) parameter.

TABLE 3.1: Selected Univariate Probability Functions Available in R

Distribution	cdf	Generator	Parameters
beta	pbeta	rbeta	shape1, shape2
binomial	pbinom	rbinom	size, prob
chi-squared	pchisq	rchisq	df
exponential	pexp	rexp	rate
F	pf	rf	df1, df2
gamma	pgamma	rgamma	shape, rate or scale
geometric	pgeom	rgeom	prob
lognormal	plnorm	rlnorm	meanlog, sdlog
negative binomial	pnbinom	rnbinom	size, prob
normal	pnorm	rnorm	mean, sd
Poisson	ppois	rpois	lambda
Student's t	pt	rt	df
uniform	punif	runif	min, max

3.2 The Inverse Transform Method

The inverse transform method of generating random variables is based on the following well-known result (see [21, p. 201] or [244, p. 203]).

Theorem 3.1 (Probability Integral Transformation). *If X is a continuous random variable with cdf $F_X(x)$, then $U = F_X(X) \sim$ Uniform(0, 1).*

The inverse transform method of generating random variables applies the probability integral transformation. Define the inverse transformation

$$F_X^{-1}(u) = \inf\{x : \; F_X(x) = u\}, \qquad 0 < u < 1.$$

If $U \sim$ Uniform(0, 1), then for all $x \in \mathbb{R}$

$$\begin{aligned} P(F_X^{-1}(U) \le x) &= P(\inf\{t : \; F_X(t) = U\} \le x) \\ &= P(U \le F_X(x)) \\ &= F_U(F_X(x)) = F_X(x), \end{aligned}$$

and therefore $F_X^{-1}(U)$ has the same distribution as X. Thus, to generate a random observation X, first generate a Uniform(0,1) variate u and deliver the inverse value $F_X^{-1}(u)$. The method is easy to apply, provided that F_X^{-1} is easy to compute. The method can be applied for generating continuous or discrete random variables. The method can be summarized as follows.

1. Derive the inverse function $F_X^{-1}(u)$.
2. Write a command or function to compute $F_X^{-1}(u)$.
3. For each random variate required:
   (a) Generate a random u from Uniform(0,1).
   (b) Deliver $x = F_X^{-1}(u)$.

3.2.1 Inverse Transform Method, Continuous Case

Example 3.2 (Inverse transform method, continuous case)**.** This example uses the inverse transform method to simulate a random sample from the distribution with density $f_X(x) = 3x^2,\ 0 < x < 1$.

Here $F_X(x) = x^3$ for $0 < x < 1$, and $F_X^{-1}(u) = u^{1/3}$. Generate all n required random uniform numbers as vector `u`. Then `u^(1/3)` is a vector of length `n` containing the sample $x_1, \dots, x_n$.

```
n <- 1000
u <- runif(n)
x <- u^(1/3)
hist(x, prob = TRUE) #density histogram of sample
y <- seq(0, 1, .01)
lines(y, 3*y^2)    #density curve f(x)
```

The histogram and density plot in Figure 3.1 suggests that the empirical and theoretical distributions approximately agree. ◇

R Note 3.1

In Figure 3.1, the title includes a math expression. This title is obtained by specifying the main title using the `expression` function as follows:

```
hist(x, prob = TRUE, main = expression(f(x)==3*x^2))
```

Alternately, `main = bquote(f(x)==3*x^2))` produces the same title. Math annotation is covered in the help topic for `plotmath`. Also see the help topics for `text` and `axis`.

Example 3.3 (Exponential distribution)**.** This example applies the inverse transform method to generate a random sample from the exponential distribution with mean $1/\lambda$.

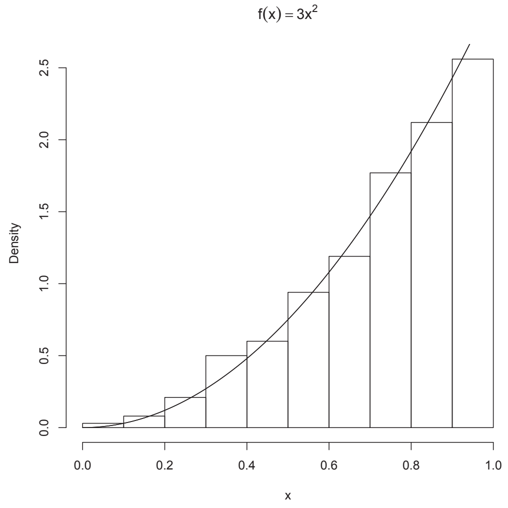

FIGURE 3.1: Probability density histogram of a random sample generated by the inverse transform method in Example 3.2, with the theoretical density $f(x) = 3x^2$ superimposed.

If $X \sim \text{Exp}(\lambda)$, then for $x > 0$ the cdf of X is $F_X(x) = 1 - e^{-\lambda x}$. The inverse transformation is $F_X^{-1}(u) = -\frac{1}{\lambda}\log(1-u)$. Note that U and $1-U$ have the same distribution and it is simpler to set $x = -\frac{1}{\lambda}\log(u)$. To generate a random sample of size `n` with parameter `lambda`:

```
-log(runif(n)) / lambda
```

A generator `rexp` is available in R. However, this algorithm is very useful for implementation in other situations, such as a C program. ◇

3.2.2 Inverse Transform Method, Discrete Case

The inverse transform method can also be applied to discrete distributions. If X is a discrete random variable and

$$\ldots < x_{i-1} < x_i < x_{i+1} < \ldots$$

are the points of discontinuity of $F_X(x)$, then the inverse transformation is $F_X^{-1}(u) = x_i$, where $F_X(x_{i-1}) < u \leq F_X(x_i)$.

For each random variate required:

1. Generate a random u from Uniform(0,1).

2. Deliver x_i where $F(x_{i-1}) < u \leq F(x_i)$.

The solution of $F(x_{i-1}) < u \leq F(x_i)$ in Step (2) may be difficult for some distributions. See Devroye [72, Ch. III] for several different methods of implementing the inverse transform method in the discrete case.

Example 3.4 (Two point distribution)**.** This example applies the inverse transform to generate a random sample of Bernoulli($p = 0.4$) variates. Although there are simpler methods to generate a two point distribution in R, this example illustrates computing the inverse cdf of a discrete random variable in the simplest case.

In this example, $F_X(0) = f_X(0) = 1-p$ and $F_X(1) = 1$. Thus, $F_X^{-1}(u) = 1$ if $u > 0.6$ and $F_X^{-1}(u) = 0$ if $u \leq 0.6$. The generator should therefore deliver the numerical value of the logical expression $u > 0.6$.

```
n <- 1000
p <- 0.4
u <- runif(n)
x <- as.integer(u > 0.6)   #(u > 0.6) is a logical vector

> mean(x)
[1] 0.41
> var(x)
[1] 0.2421421
```

Compare the sample statistics with the theoretical moments. The sample mean of a generated sample should be approximately $p = 0.4$ and the sample variance should be approximately $p(1-p) = 0.24$. Our sample statistics are $\bar{x} = 0.41$ ($se = \sqrt{0.24/1000} \doteq 0.0155$) and $s^2 \doteq 0.242$. ◇

R Note 3.2

In R one can use the `rbinom` (random binomial) function with `size=1` to generate a Bernoulli sample. Another method is to sample from the vector (0,1) with probabilities ($1-p$, p).

```
rbinom(n, size = 1, prob = p)
sample(c(0,1), size = n, replace = TRUE, prob = c(.6,.4))
```

Also see Example 3.1.

Example 3.5 (Geometric distribution)**.** Use the inverse transform method to generate a random geometric sample with parameter $p = 1/4$.

The pmf is $f(x) = pq^x,\ x = 0, 1, 2, \ldots$, where $q = 1-p$. At the points of discontinuity $x = 0, 1, 2, \ldots$, the cdf is $F(x) = 1 - q^{x+1}$. For each sample element we need to generate a random uniform u and solve

$$1 - q^x < u \leq 1 - q^{x+1}.$$

This inequality simplifies to $x < \log(1-u)/\log(q) \leq x + 1$. The solution is $x + 1 = \lceil \log(1-u)/\log(q) \rceil$, where $\lceil t \rceil$ denotes the ceiling function (the smallest integer not less than t).

```
n <- 1000
p <- 0.25
u <- runif(n)
k <- ceiling(log(1-u) / log(1-p)) - 1
```

Here again there is a simplification, because U and $1 - U$ have the same distribution. Also, the probability that $\log(1-u)/\log(1-p)$ equals an integer is zero. The last step can therefore be simplified to

```
k <- floor(log(u) / log(1-p))
```

$\diamond$

The geometric distribution was particularly easy to simulate by the inverse transform method because it was easy to solve the inequality

$$F(x-1) < u \leq F(x)$$

rather than compare each u to all the possible values $F(x)$. The same method applied to the Poisson distribution is more complicated because we do not have an explicit formula for the value of x such that $F(x-1) < u \leq F(x)$.

The R function `rpois` generates random Poisson samples. The basic method to generate a Poisson(λ) variate (see e.g. [250]) is to generate and store the cdf via the recursive formula

$$f(x+1) = \frac{\lambda f(x)}{x+1}; \qquad F(x+1) = F(x) + f(x+1).$$

For each Poisson variate required, a random uniform u is generated, and the cdf vector is searched for the solution to $F(x-1) < u \leq F(x)$.

To illustrate the main idea of the inverse transform method for generating Poisson variates, here is a similar example for which there is no R generator available: the logarithmic distribution. The logarithmic distribution is a one-parameter discrete distribution supported on the positive integers.

Example 3.6 (Logarithmic distribution)**.** This example implements a function to simulate a Logarithmic(θ) random sample by the inverse transform method. A random variable X has the logarithmic distribution (see [164], Ch. 7) if

$$f(x) = P(X = x) = \frac{a\,\theta^x}{x}, \qquad x = 1, 2, \ldots \tag{3.1}$$

where $0 < \theta < 1$ and $a = (-\log(1-\theta))^{-1}$. A recursive formula for $f(x)$ is

$$f(x+1) = \frac{\theta^x}{x+1} f(x), \qquad x = 1, 2, \ldots. \tag{3.2}$$

Theoretically, the pmf can be evaluated recursively using (3.2), but the calculation is not sufficiently accurate for large values of x and ultimately produces $f(x) = 0$ with $F(x) < 1$. Instead we compute the pmf from (3.1) as $\exp(\log a + x \log \theta - \log x)$. In generating a large sample, there will be many repetitive calculations of the same values $F(x)$. It is more efficient to store the cdf values. Initially choose a length N for the cdf vector, and compute $F(x)$, $x = 1, 2, \dots, N$. If necessary, N will be increased.

To solve $F(x-1) < u \leq F(x)$ for a particular u, it is necessary to count the number of values x such that $F(x-1) < u$. If F is a vector and u_i is a scalar, then the expression $F < u_i$ produces a logical vector; that is, a vector the same length as F containing logical values `TRUE` or `FALSE`. In an arithmetic expression, `TRUE` has value 1 and `FALSE` has value 0. Notice that the sum of the logical vector $(u_i > F)$ is exactly $x - 1$.

```
rlogarithmic <- function(n, theta) {
    #returns a random logarithmic(theta) sample size n
    u <- runif(n)
    #set the initial length of cdf vector
    N <- ceiling(-16 / log10(theta))
    k <- 1:N
    a <- -1/log(1-theta)
    fk <- exp(log(a) + k * log(theta) - log(k))
    Fk <- cumsum(fk)
    x <- integer(n)
    for (i in 1:n) {
        x[i] <- as.integer(sum(u[i] > Fk)) #F^{-1}(u)-1
        while (x[i] == N) {
            #if x==N we need to extend the cdf
            #very unlikely because N is large
            logf <- log(a) + (N+1)*log(theta) - log(N+1)
            fk <- c(fk, exp(logf))
            Fk <- c(Fk, Fk[N] + fk[N+1])
            N <- N + 1
            x[i] <- as.integer(sum(u[i] > Fk))
        }
    }
    x + 1
}
```

Generate random samples from a Logarithmic(0.5) distribution.

```
n <- 1000
theta <- 0.5
x <- rlogarithmic(n, theta)
#compute density of logarithmic(theta) for comparison
k <- sort(unique(x))
p <- -1 / log(1 - theta) * theta^k / k
se <- sqrt(p*(1-p)/n)   #standard error
```

In the following results, the relative frequencies of the sample (first line) match the theoretical distribution (second line) of the Logarithmic(0.5) distribution within two standard errors.

```
> round(rbind(table(x)/n, p, se),3)
       1     2     3     4     5     6     7
   0.741 0.169 0.049 0.026 0.008 0.003 0.004
p  0.721 0.180 0.060 0.023 0.009 0.004 0.002
se 0.014 0.012 0.008 0.005 0.003 0.002 0.001
```

◇

Remark 3.1. A more efficient generator for the Logarithmic(θ) distribution is implemented in Example 3.9 of Section 3.4.

3.3 The Acceptance-Rejection Method

Suppose that X and Y are random variables with density or pmf f and g, respectively, and there exists a constant c such that

$$\frac{f(t)}{g(t)} \leq c$$

for all t such that $f(t) > 0$. Then the acceptance-rejection method (or rejection method) can be applied to generate the random variable X.

The Acceptance-Rejection Method

1. Find a random variable Y with density g satisfying $f(t)/g(t) \leq c$, for all t such that $f(t) > 0$. Provide a method to generate random Y.
2. For each random variate required:
   (a) Generate a random y from the distribution with density g.
   (b) Generate a random u from the Uniform(0, 1) distribution.
   (c) If $u < f(y)/(cg(y))$, accept y and deliver $x = y$; otherwise reject y and repeat from Step 2a.

Note that in Step 2c,

$$P(\text{accept}|Y) = P\big(U < \frac{f(Y)}{cg(Y)} \,\big|Y\big) = \frac{f(Y)}{cg(Y)}.$$

The last equality is simply evaluating the cdf of U. The total probability of acceptance for any iteration is therefore

$$\sum_y P(\text{accept}|y)P(Y = y) = \sum_y \frac{f(y)}{cg(y)}g(y) = \frac{1}{c},$$

and the number of iterations until acceptance has the geometric distribution with mean c. Hence, on average each sample value of X requires c iterations. For efficiency, Y should be easy to simulate and c small.

To see that the accepted sample has the same distribution as X, apply Bayes' Theorem. In the discrete case, for each k such that $f(k) > 0$,

$$P(k \mid \text{accepted}) = \frac{P(\text{accepted} \mid k) g(k)}{P(\text{accepted})} = \frac{[f(k)/(cg(k))]\, g(k)}{1/c} = f(k).$$

The continuous case is similar.

Example 3.7 (Acceptance-rejection method)**.** This example illustrates the acceptance-rejection method for the beta distribution. On average, how many random numbers must be simulated to generate 1000 variates from the Beta($\alpha = 2$, $\beta = 2$) distribution by this method? It depends on the upper bound c of $f(x)/g(x)$, which depends on the choice of the function $g(x)$.

The Beta(2,2) density is $f(x) = 6x(1 - x),\ 0 < x < 1$. Let $g(x)$ be the Uniform(0,1) density. Then $f(x)/g(x) \leq 6$ for all $0 < x < 1$, so $c = 6$. A random x from $g(x)$ is accepted if

$$\frac{f(x)}{cg(x)} = \frac{6x(1-x)}{6(1)} = x(1-x) > u.$$

On average, $cn = 6000$ iterations (12000 random numbers) will be required for a sample size 1000. In the following simulation, the counter `j` for iterations is not necessary, but included to record how many iterations were actually needed to generate the 1000 beta variates.

```
n <- 1000
k <- 0      #counter for accepted
j <- 0      #iterations
y <- numeric(n)

while (k < n) {
    u <- runif(1)
    j <- j + 1
    x <- runif(1)  #random variate from g
    if (x * (1-x) > u) {
        #we accept x
        k <- k + 1
        y[k] <- x
    }
}

> j
[1] 5873
```

In this simulation, 5873 iterations (11746 random numbers) were required to generate the 1000 beta variates. Compare the empirical and theoretical percentiles.

```
#compare empirical and theoretical percentiles
p <- seq(.1, .9, .1)
Qhat <- quantile(y, p)   #quantiles of sample
Q <- qbeta(p, 2, 2)      #theoretical quantiles
se <- sqrt(p * (1-p) / (n * dbeta(Q, 2, 2)^2)) #see Ch. 2
```

The sample percentiles (first line) approximately match the Beta(2,2) percentiles computed by `qbeta` (second line), most closely near the center of the distribution. Larger numbers of replicates are required for estimation of percentiles where the density is close to zero.

```
> round(rbind(Qhat, Q, se), 3)
       10%   20%   30%   40%   50%   60%   70%   80%   90%
Qhat 0.189 0.293 0.365 0.449 0.519 0.589 0.665 0.741 0.830
Q    0.196 0.287 0.363 0.433 0.500 0.567 0.637 0.713 0.804
se   0.010 0.010 0.010 0.011 0.011 0.011 0.010 0.010 0.010
```

Repeating the simulation with $n = 10000$ produces more precise estimates.

```
>  round(rbind(Qhat, Q, se), 3)
       10%   20%   30%   40%   50%   60%   70%   80%   90%
Qhat 0.194 0.292 0.368 0.436 0.504 0.572 0.643 0.716 0.804
Q    0.196 0.287 0.363 0.433 0.500 0.567 0.637 0.713 0.804
se   0.003 0.004 0.004 0.004 0.004 0.004 0.004 0.004 0.003
```

◇

Remark 3.2. See Example 3.8 for a more efficient beta generator based on the ratio of gammas method.

3.4 Transformation Methods

Many types of transformations other than the probability inverse transformation can be applied to simulate random variables. Some examples are

1. If $Z \sim$ N(0,1), then $V = Z^2 \sim \chi^2(1)$.
2. If $U \sim \chi^2(m)$ and $V \sim \chi^2(n)$ are independent, then $F = \frac{U/m}{V/n}$ has the F distribution with (m, n) degrees of freedom.
3. If $Z \sim$ N(0,1) and $V \sim \chi^2(n)$ are independent, then $T = \frac{Z}{\sqrt{V/n}}$ has the Student t distribution with n degrees of freedom.
4. If $U, V \sim$ Unif(0,1) are independent, then

$$Z_1 = \sqrt{-2\log U}\ \cos(2\pi V),$$
$$Z_2 = \sqrt{-2\log U}\ \sin(2\pi V)$$

are independent standard normal variables [255, p. 86].

5. If $U \sim \text{Gamma}(r, \lambda)$ and $V \sim \text{Gamma}(s, \lambda)$ are independent, then $X = \frac{U}{U+V}$ has the $\text{Beta}(r, s)$ distribution.

6. If $U, V \sim \text{Unif}(0,1)$ are independent, then
$$X = \left\lfloor 1 + \frac{\log(V)}{\log(1-(1-\theta)^U)} \right\rfloor$$
has the $\text{Logarithmic}(\theta)$ distribution, where $\lfloor x \rfloor$ denotes the integer part of x.

Generators based on transformations (5) and (6) are implemented in Examples 3.8 and 3.9. Sums and mixtures are special types of transformations that are discussed in Section 3.5. Example 3.21 uses a multivariate transformation to generate points uniformly distributed on the unit sphere.

Example 3.8 (Beta distribution)**.** The following relation between beta and gamma distributions provides another beta generator.

If $U \sim \text{Gamma}(r, \lambda)$ and $V \sim \text{Gamma}(s, \lambda)$ are independent, then
$$X = \frac{U}{U+V}$$
has the $\text{Beta}(r, s)$ distribution [255, p.64]. This transformation determines an algorithm for generating random $\text{Beta}(a, b)$ variates.

1. Generate a random u from $\text{Gamma}(a, 1)$.
2. Generate a random v from $\text{Gamma}(b, 1)$.
3. Deliver $x = \frac{u}{u+v}$.

This method is applied below to generate a random Beta(3, 2) sample.

```
n <- 1000
a <- 3
b <- 2
u <- rgamma(n, shape=a, rate=1)
v <- rgamma(n, shape=b, rate=1)
x <- u / (u + v)
```

The sample data can be compared with the Beta(3, 2) distribution using a quantile-quantile (QQ) plot. If the sampled distribution is Beta(3, 2), the QQ plot should be nearly linear.

```
q <- qbeta(ppoints(n), a, b)
qqplot(q, x, cex=0.25, xlab="Beta(3, 2)", ylab="Sample")
abline(0, 1)
```

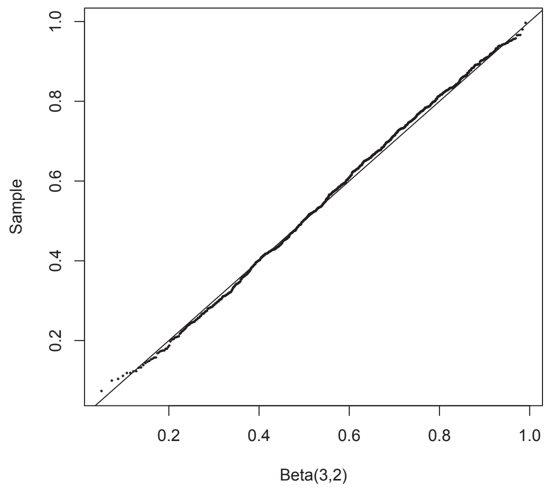

FIGURE 3.2: QQ Plot comparing the Beta(3, 2) distribution with a simulated random sample generated by the ratio of gammas method in Example 3.8.

The line $x = q$ is added for reference. The QQ plot of the ordered sample vs. the Beta(3, 2) quantiles in Figure 3.2 is very nearly linear, as it should be if the generated sample is in fact a Beta(3, 2) sample. $\diamond$

Example 3.9 (Logarithmic distribution, version 2)**.** This example provides another, more efficient generator for the logarithmic distribution (see Example 3.6). If U, V are independent Uniform(0,1) random variables, then

$$X = \left\lfloor 1 + \frac{\log(V)}{\log(1-(1-\theta)^U)} \right\rfloor \tag{3.3}$$

has the Logarithmic(θ) distribution ([72, pp. 546-8], [165]). This transformation provides a simple and efficient generator for the logarithmic distribution.

1. Generate u from Unif(0,1).
2. Generate v from Unif(0,1).
3. Deliver $x = \lfloor 1 + \log(v)/\log(1-(1-\theta)^u) \rfloor$.

Below is a comparison of the Logarithmic(0.5) distribution with a sample generated using transformation (3.3). The empirical probabilities `p.hat` are within two standard errors of the theoretical probabilities `p`.

```
n <- 1000
theta <- 0.5
```

```
u <- runif(n)  #generate logarithmic sample
v <- runif(n)
x <- floor(1 + log(v) / log(1 - (1 - theta)^u))
k <- 1:max(x)  #calc. logarithmic probs.
p <- -1 / log(1 - theta) * theta^k / k
se <- sqrt(p*(1-p)/n)
p.hat <- tabulate(x)/n

> print(round(rbind(p.hat, p, se), 3))
       [,1]  [,2]  [,3]  [,4]  [,5]  [,6]  [,7]
p.hat 0.740 0.171 0.052 0.018 0.010 0.006 0.003
p     0.721 0.180 0.060 0.023 0.009 0.004 0.002
se    0.014 0.012 0.008 0.005 0.003 0.002 0.001
```

The following function is a simple replacement for `rlogarithmic` in Example 3.6.

```
rlogarithmic <- function(n, theta) {
    stopifnot(all(theta > 0 & theta < 1))
    th <- rep(theta, length=n)
    u <- runif(n)
    v <- runif(n)
    x <- floor(1 + log(v) / log(1 - (1 - th)^u))
    return(x)
}
```

◇

R Note 3.3

The `tabulate` function bins positive integers, so it can be used on the logarithmic sample. For other types of data, recode the data to positive integers or use `table`. If the data are not positive integers, `tabulate` will truncate real numbers and ignore without warning integers less than 1.

R Note 3.4

In the `rlogarithmic` function above, notice the `&` operator in `stopifnot(all(theta > 0 & theta < 1))`. Here we must use `&` rather than `&&`. The `&` operator performs an elementwise AND comparison, returning a logical vector. The `&&` operator, however, evaluates from left to right until a single logical result is obtained. For example

```
x <- 1:5
> 1 < x & x < 5
[1] FALSE  TRUE  TRUE  TRUE FALSE
> 1 < x && x < 5
[1] FALSE
> any( 1 < x & x < 5 )
[1] TRUE
> any( 1 < x && x < 5 )
[1] FALSE
> any(1 < x) && any(x < 5)
[1] TRUE
> all(1 < x) && all(x < 5)
[1] FALSE
```

Similarly, `|` performs elementwise an OR comparison, and `||` evaluates from left to right.

3.5 Sums and Mixtures

Sums and mixtures of random variables are special types of transformations. In this section we focus on sums of independent random variables (convolutions) and several examples of discrete and continuous mixtures.

Convolutions

Let $X_1, \dots, X_n$ be independent and identically distributed with distribution $X_j \sim X$, and let $S = X_1 + \cdots + X_n$. The distribution function of the sum S is called the n-fold convolution of X and denoted $F_X^{*(n)}$. It is straightforward to simulate a convolution by directly generating $X_1, \dots, X_n$ and computing the sum.

Several distributions are related by convolution. If $\nu > 0$ is an integer, the chisquare distribution with ν degrees of freedom is the convolution of ν iid squared standard normal variables. The negative binomial distribution NegBin(r, p) is the convolution of r iid Geom(p) random variables. The convolution of r independent Exp(λ) random variables has the Gamma(r, λ) distribution. See Bean [26] for an introductory level presentation of these and many other interesting relationships between families of distributions.

In R it is of course easier to use the functions `rchisq`, `rgeom` and `rnbinom` to generate chisquare, geometric and negative binomial random samples. The following example is presented to illustrate a general method that can be applied whenever distributions are related by convolutions.

Example 3.10 (Chisquare)**.** This example generates a chisquare $\chi^2(\nu)$ random variable as the convolution of ν squared normals. If $Z_1, \dots, Z_\nu$ are iid N(0,1) random variables, then $V = Z_1^2 + \cdots + Z_\nu^2$ has the $\chi^2(\nu)$ distribution. Steps to generate a random sample of size n from $\chi^2(\nu)$ are as follows:

1. Fill an $n \times \nu$ matrix with $n\nu$ random N(0,1) variates.
2. Square each entry in the matrix (1).
3. Compute the row sums of the squared normals. Each row sum is one random observation from the $\chi^2(\nu)$ distribution.
4. Deliver the vector of row sums.

An example with $n = 1000$ and $\nu = 2$ is shown below.

```
n <- 1000
nu <- 2
X <- matrix(rnorm(n*nu), n, nu)^2 #matrix of sq. normals
#sum the squared normals across each row: method 1
y <- rowSums(X)
#method 2
y <- apply(X, MARGIN=1, FUN=sum)  #a vector length n
> mean(y)
[1] 2.027334
> mean(y^2)
[1] 7.835872
```

A $\chi^2(\nu)$ random variable has mean ν and variance 2ν. Our sample statistics below agree very closely with the theoretical moments $E[Y] = \nu = 2$ and $E[Y^2] = 2\nu + \nu^2 = 8$. Here the standard errors of the sample moments are 0.063 and 0.089, respectively. ◇

R Note 3.5

This example introduces the `apply` function. The `apply` function applies a function to the margins of an array. To sum across the rows of matrix `X`, the function (`FUN=sum`) is applied to the rows (`MARGIN=1`). Notice that a loop is not used to compute the row sums. In general for efficient programming in R, avoid unnecessary loops. (For row and column sums it is easier to use `rowSums` and `colSums`.)

Mixtures

A random variable X is a discrete mixture if the distribution of X is a weighted sum $F_X(x) = \sum \theta_i F_{X_i}(x)$ for some sequence of random variables $X_1, X_2, \dots$ and $\theta_i > 0$ such that $\sum_i \theta_i = 1$. The constants θ_i are called the mixing weights or mixing probabilities. Although the notation is similar for sums and mixtures, the distributions represented are different.

A random variable X is a continuous mixture if the distribution of X is $F_X(x) = \int_{-\infty}^{\infty} F_{X|Y=y}(x) f_Y(y)\, dy$ for a family $X|Y = y$ indexed by the real numbers y and weighting function f_Y such that $\int_{-\infty}^{\infty} f_Y(y)\, dy = 1$.

Compare the methods for simulation of a convolution and a mixture of normal variables. Suppose $X_1 \sim N(0,1)$ and $X_2 \sim N(3,1)$ are independent. The notation $S = X_1 + X_2$ denotes the *convolution* of X_1 and X_2. The distribution of S is normal with mean $\mu_1 + \mu_2 = 3$ and variance $\sigma_1^2 + \sigma_2^2 = 2$. To

simulate the *convolution*:

1. Generate $x_1 \sim$ N(0, 1).
2. Generate $x_2 \sim$ N(3, 1).
3. Deliver $s = x_1 + x_2$.

We can also define a 50% normal *mixture* X, denoted $F_X(x) = 0.5F_{X_1}(x) + 0.5F_{X_2}(x)$. Unlike the convolution above, the distribution of the mixture X is distinctly non-normal; it is bimodal.

To simulate the *mixture*:

1. Generate an integer $k \in \{1, 2\}$, where $P(1) = P(2) = 0.5$.
2. If $k = 1$ deliver random x from N(0, 1);
   if $k = 2$ deliver random x from N(3, 1).

In the following example we will compare simulated distributions of a convolution and a mixture of gamma random variables.

Example 3.11 (Convolutions and mixtures)**.** Let $X_1 \sim$ Gamma(2, 2) and $X_2 \sim$ Gamma(2, 4) be independent. Compare the histograms of the samples generated by the convolution $S = X_1 + X_2$ and the mixture $F_X = 0.5F_{X_1} + 0.5F_{X_2}$.

```
n <- 1000
x1 <- rgamma(n, 2, 2)
x2 <- rgamma(n, 2, 4)
s <- x1 + x2              #the convolution
u <- runif(n)
k <- as.integer(u > 0.5)  #vector of 0's and 1's
x <- k * x1 + (1-k) * x2  #the mixture
```

```
par(mfcol=c(1,2))         #two graphs per page
hist(s, prob=TRUE, xlim=c(0,5), ylim=c(0,1))
hist(x, prob=TRUE, xlim=c(0,5), ylim=c(0,1))
par(mfcol=c(1,1))         #restore display
```

The histograms shown in Figure 3.3, of the convolution S and mixture X, are clearly different. ◇

R Note 3.6

The `par` function can be used to set (or query) certain graphical parameters. A list of all graphical parameters is returned by `par()`. The command `par(mfcol=c(n,m))` configures the graphical device to display nm graphs per screen, in n rows and m columns.

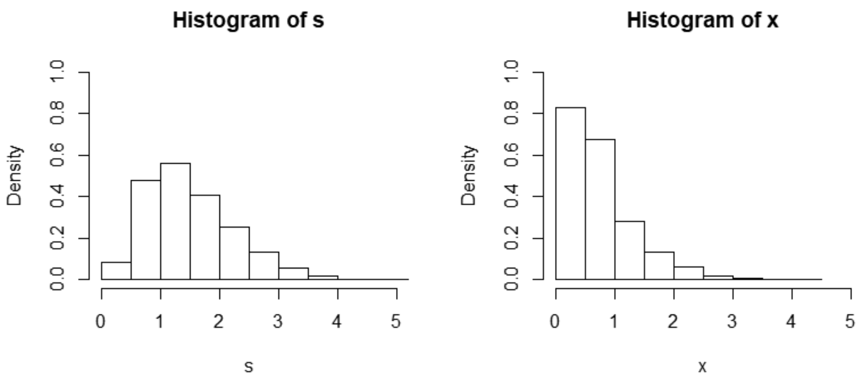

FIGURE 3.3: Histogram of a simulated convolution of Gamma(2, 2) and Gamma(2, 4) random variables (left), and a 50% mixture of the same variables (right), from Example 3.11.

The method of generating the mixture in this example is simple for a mixture of two distributions, but not for arbitrary mixtures. The next example illustrates how to generate a mixture of several distributions with arbitrary mixing probabilities.

Example 3.12 (Mixture of several gamma distributions). This example is similar to the previous one, but there are several components to the mixture and the mixing weights are not uniform. The mixture is

$$F_X = \sum_{i=1}^{5} \theta_j F_{X_j},$$

where $X_j \sim \text{Gamma}(r = 3, \lambda_j = 1/j)$ are independent and the mixing probabilities are $\theta_j = j/15$, $j = 1, \ldots, 5$.

To simulate one random variate from the mixture F_X:

1. Generate an integer $k \in \{1, 2, 3, 4, 5\}$, where $P(k) = \theta_k$, $k = 1, \ldots, 5$.

2. Deliver a random Gamma(r, λ_k) variate.

To generate a sample size n, Steps (1) and (2) are repeated n times. Notice that the algorithm stated above suggests using a `for` loop, but `for` loops are really inefficient in R. The algorithm can be translated into a vectorized approach.

1. Generate a random sample $k_1, \ldots, k_n$ of integers in a vector `k`, where $P(k) = \theta_k$, $k = 1, \ldots, 5$. Then `k[i]` indicates which of the five gamma distributions will be sampled to get the i^{th} element of the sample (use `sample`).

2. Set `rate` equal to the length n vector $\lambda = (\lambda_k)$.

3. Generate a gamma sample size n, with shape parameter r and rate vector `rate` (use `rgamma`).

Then an efficient way to implement this in R is shown by the following example.

```
n <- 5000
k <- sample(1:5, size=n, replace=TRUE, prob=(1:5)/15)
rate <- 1/k
x <- rgamma(n, shape=3, rate=rate)

#plot the density of the mixture
#with the densities of the components
plot(density(x), xlim=c(0,40), ylim=c(0,.3),
    lwd=3, xlab="x", main="")
for (i in 1:5)
    lines(density(rgamma(n, 3, 1/i)))
```

The plot in Figure 3.4 shows the density of each X_j and the density of the mixture (thick line). The density curves in Figure 3.4 are actually density estimates, which will be discussed in Chapter 12. ◇

Example 3.13 (Mixture of several gamma distributions). Let

$$F_X = \sum_{j=1}^{5} \theta_j F_{X_j}$$

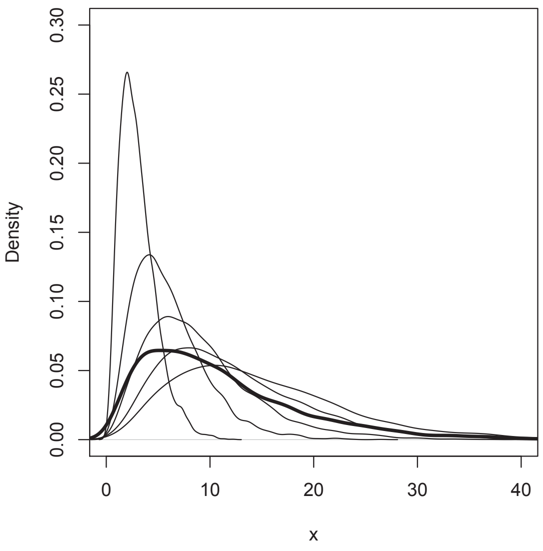

FIGURE 3.4: Density estimates from Example 3.12: A mixture (thick line) of several gamma densities (thin lines).

where $X_j \sim \text{Gamma}(3, \lambda_j)$ are independent, with rates $\lambda = (1, 1.5, 2, 2.5, 3)$, and mixing probabilities $\theta = (0.1, 0.2, 0.2, 0.3, 0.2)$.

This example is similar to the previous one. Sample from `1:5` with probability weights θ to get a vector length n. The i^{th} position in this vector indicates which of the five gamma distributions is sampled to get the i^{th} element of the sample. This vector is used to select the correct rate parameter from the vector λ.

```
n <- 5000
p <- c(.1,.2,.2,.3,.2)
lambda <- c(1,1.5,2,2.5,3)
k <- sample(1:5, size=n, replace=TRUE, prob=p)
rate <- lambda[k]
x <- rgamma(n, shape=3, rate=rate)
```

Note that `lambda[k]` is a vector the same length as `k`, containing the elements of `lambda` indexed by the vector `k`. In mathematical notation, `lambda[k]` is equal to $(\lambda_{k_1}, \lambda_{k_2}, \ldots, \lambda_{k_n})$.

Compare the first few entries of `k` and the corresponding values of `rate` with λ.

```
> k[1:8]
[1] 5 1 4 2 1 3 2 3
> rate[1:8]
[1] 3.0 1.0 2.5 1.5 1.0 2.0 1.5 2.0
```

◇

Example 3.14 (Plot density of mixture)**.** Plot the densities (not density estimates) of the gamma distributions and the mixture in Example 3.13. (This example is a programming exercise that involves vectors of parameters and repeated use of the `apply` function.)

The density of the mixture is

$$f(x) = \sum_{j=1}^{5} \theta_j f_j(x), \quad x > 0, \tag{3.4}$$

where f_j is the Gamma(3, λ_j) density. To produce the plot, we need a function to compute the density $f(x)$ of the mixture.

```
f <- function(x, lambda, theta) {
    #density of the mixture at the point x
    sum(dgamma(x, 3, lambda) * theta)
}
```

The function `f` computes the density of the mixture (3.4) for a single value of `x`. If `x` has length 1, `dgamma(x, 3, lambda)` is a vector the same length as `lambda`; in this case $(f_1(x), \ldots, f_5(x))$. Then `dgamma(x, 3, lambda)*theta` is the vector $(\theta_1 f_1(x), \ldots, \theta_5 f_5(x))$. The sum of this vector is the density of the mixture (3.3) evaluated at the point `x`.

```
x <- seq(0, 8, length=200)
dim(x) <- length(x)  #need for apply

#compute density of the mixture f(x) along x
y <- apply(x, 1, f, lambda=lambda, theta=p)
```

The density of the mixture is computed by function `f` applied to the vector `x`. The function `f` takes several arguments, so the additional arguments `lambda=lambda, theta=prob` are supplied after the name of the function, `f`.

A plot of the five densities with the mixture is shown in Figure 3.5. The code to produce the plot is listed below. The densities f_k can be computed by the `dgamma` function. A sequence of points `x` is defined and each of the densities is computed along `x`.

```
#plot the density of the mixture
plot(x, y, type="l", ylim=c(0,.85), lwd=3, ylab="Density")

for (j in 1:5) {
    #add the j-th gamma density to the plot
    y <- apply(x, 1, dgamma, shape=3, rate=lambda[j])
    lines(x, y)
}
```

◇

R Note 3.7

The `apply` function requires a dimension attribute for `x`. Since `x` is a vector, it does not have a dimension attribute by default. The dimension of `x` is assigned by `dim(x) <- length(x)`. Alternately, `x <- as.matrix(x)` converts `x` to a matrix (a column vector), which has a dimension attribute.

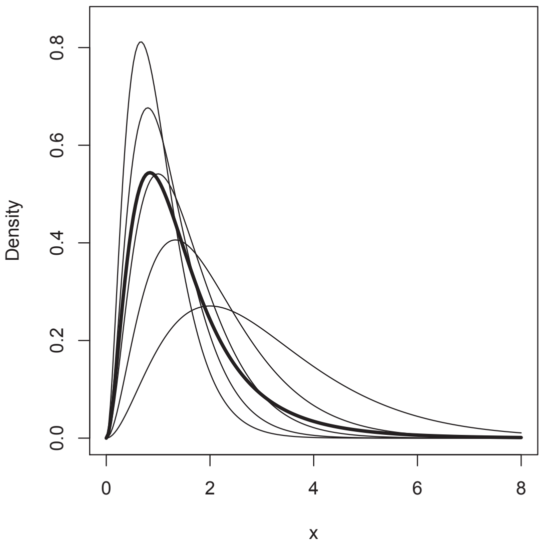

FIGURE 3.5: Densities from Example 3.14: A mixture (thick line) of several gamma densities (thin lines).

Example 3.15 (Poisson-Gamma mixture)**.** This is an example of a continuous mixture. The negative binomial distribution is a mixture of Poisson(Λ) distributions, where Λ has a gamma distribution. Specifically, if $(X|\Lambda = \lambda) \sim$ Poisson(λ) and $\Lambda \sim$ Gamma(r, β), then X has the negative binomial distribution with parameters r and $p = \beta/(1 + \beta)$ (see, e.g., [26]). This example illustrates a method of sampling from a Poisson-Gamma mixture and compares the sample with the negative binomial distribution.

```
#generate a Poisson-Gamma mixture
n <- 1000
r <- 4
beta <- 3
lambda <- rgamma(n, r, beta) #lambda is random

#now supply the sample of lambda's as the Poisson mean
```

```
x <- rpois(n, lambda)       #the mixture

#compare with negative binomial
mix <- tabulate(x+1) / n
negbin <- round(dnbinom(0:max(x), r, beta/(1+beta)), 3)
se <- sqrt(negbin * (1 - negbin) / n)
```

The empirical distribution (first line below) of the mixture agrees very closely with the pmf of NegBin$(4, 3/4)$ (second line).

```
> round(rbind(mix, negbin, se), 3)
        [,1]  [,2]  [,3]  [,4]  [,5]  [,6]  [,7]  [,8]  [,9]
mix    0.334 0.305 0.201 0.091 0.042 0.018 0.005 0.003 0.001
negbin 0.316 0.316 0.198 0.099 0.043 0.017 0.006 0.002 0.001
se     0.015 0.015 0.013 0.009 0.006 0.004 0.002 0.001 0.001
```

$\diamond$

3.6 Multivariate Distributions

Generators for the multivariate normal distribution, multivariate normal mixtures, Wishart distribution, and uniform distribution on the sphere in $\mathbb{R}^d$ are presented in this section.

3.6.1 Multivariate Normal Distribution

A random vector $X = (X_1, \dots, X_d)$ has a d-dimensional multivariate normal (MVN) distribution denoted $N_d(\mu, \Sigma)$ if the density of X is

$$f(x) = \frac{1}{(2\pi)^{d/2}\,|\Sigma|^{1/2}} \exp\{-(1/2)(x-\mu)^T \Sigma^{-1}(x-\mu)\}, \quad x \in \mathbb{R}^d, \qquad (3.5)$$

where $\mu = (\mu_1, \dots, \mu_d)^T$ is the mean vector and Σ is a $d \times d$ symmetric positive definite matrix

$$\Sigma = \begin{bmatrix} \sigma_{11} & \sigma_{12} & \dots & \sigma_{1d} \\ \sigma_{21} & \sigma_{22} & \dots & \sigma_{2d} \\ \vdots & \vdots & & \vdots \\ \sigma_{d1} & \sigma_{d2} & \dots & \sigma_{dd} \end{bmatrix}$$

with entries $\sigma_{ij} = Cov(X_i, X_j)$. Here Σ^{-1} is the inverse of Σ, and $|\Sigma|$ is the determinant of Σ. The bivariate normal distribution is the special case $N_2(\mu, \Sigma)$.

A random $N_d(\mu, \Sigma)$ variate can be generated in two steps. First generate $Z = (Z_1, \dots, Z_d)$, where $Z_1, \dots, Z_d$ are iid standard normal variates. Then transform the random vector Z so that it has the desired mean vector μ and covariance structure Σ. The transformation requires factoring the covariance matrix Σ.

Recall that if $Z \sim N_d(\mu, \Sigma)$, then the linear transformation $CZ + b$ is multivariate normal with mean $C\mu + b$ and covariance $C\Sigma C^T$. If Z is $N_d(0, I_d)$, then

$$CZ + b \sim N_d(b, CC^T).$$

Suppose that Σ can be factored so that $\Sigma = CC^T$ for some matrix C. Then

$$CZ + \mu \sim N_d(\mu, \Sigma),$$

and $CZ + \mu$ is the required transformation.

The required factorization of Σ can be obtained by the spectral decomposition method (eigenvector decomposition), Choleski factorization, or singular value decomposition (svd). The corresponding R functions are `eigen`, `chol`, and `svd`.

Usually, one does not apply a linear transformation to the random vectors of a sample one at a time. Typically, one applies the transformation to a data matrix and transforms the entire sample. Suppose that $Z = (Z_{ij})$ is an $n \times d$ matrix where Z_{ij} are iid N(0,1). Then the rows of Z are n random observations from the d-dimensional standard MVN distribution. The required transformation applied to the data matrix is

$$X = ZQ + J\mu^T, \tag{3.6}$$

where $Q^TQ = \Sigma$ and J is a column vector of ones. The rows of X are n random observations from the d-dimensional MVN distribution with mean vector μ and covariance matrix Σ.

Method for generating multivariate normal samples

To generate a random sample of size n from the $N_d(\mu, \Sigma)$ distribution:

1. Generate an $n \times d$ matrix Z containing nd random $N(0, 1)$ variates (n random vectors in $\mathbb{R}^d$).
2. Compute a factorization $\Sigma = Q^TQ$.
3. Apply the transformation $X = ZQ + J\mu^T$.
4. Deliver the $n \times d$ matrix X.
   Each row of X is a random variate from the $N_d(\mu, \Sigma)$ distribution.

The $X = ZQ + J\mu^T$ transformation can be coded in R as follows. Recall that the matrix multiplication operator is `%*%`.

```
Z <- matrix(rnorm(n*d), nrow = n, ncol = d)
X <- Z %*% Q + matrix(mu, n, d, byrow = TRUE)
```

The matrix product $J\mu^T$ is equal to `matrix(mu, n, d, byrow = TRUE)`. This saves a matrix multiplication. The argument `byrow = TRUE` is necessary here; the default is `byrow = FALSE`. The matrix is filled row by row with the entries of the mean vector `mu`.

In this section, each method of generating MVN random samples is illustrated with examples. Also note that there are functions provided in R packages for generating multivariate normal samples. See the `mvrnorm` function in the `MASS` package [293], and `rmvnorm` in the `mvtnorm` package [121]. In all of the examples below, the `rnorm` function is used to generate standard normal random variates.

Spectral decomposition method for generating $N_d(\mu, \Sigma)$ samples

The square root of the covariance is $\Sigma^{1/2} = P\Lambda^{1/2}P^{-1}$, where Λ is the diagonal matrix with the eigenvalues of Σ along the diagonal and P is the matrix whose columns are the eigenvectors of Σ corresponding to the eigenvalues in Λ. This method can also be called the eigen-decomposition method. In the eigen-decomposition we have $P^{-1} = P^T$ and therefore $\Sigma^{1/2} = P\Lambda^{1/2}P^T$. The matrix $Q = \Sigma^{1/2}$ is a factorization of Σ such that $Q^TQ = \Sigma$.

Example 3.16 (Spectral decomposition method)**.** This example provides a function `rmvn.eigen` to generate a multivariate normal random sample. It is applied to generate a bivariate normal sample with zero mean vector and

$$\Sigma = \begin{bmatrix} 1.0 & 0.9 \\ 0.9 & 1.0 \end{bmatrix}.$$

```
# mean and covariance parameters
mu <- c(0, 0)
Sigma <- matrix(c(1, .9, .9, 1), nrow = 2, ncol = 2)
```

The `eigen` function returns the eigenvalues and eigenvectors of a matrix.

```
rmvn.eigen <-
function(n, mu, Sigma) {
    # generate n random vectors from MVN(mu, Sigma)
    # dimension is inferred from mu and Sigma
    d <- length(mu)
    ev <- eigen(Sigma, symmetric = TRUE)
    lambda <- ev$values
    V <- ev$vectors
    R <- V %*% diag(sqrt(lambda)) %*% t(V)
    Z <- matrix(rnorm(n*d), nrow = n, ncol = d)
    X <- Z %*% R + matrix(mu, n, d, byrow = TRUE)
    X
}
```

Print summary statistics and display a scatterplot as a check on the results of the simulation.

```
# generate the sample
X <- rmvn.eigen(1000, mu, Sigma)

plot(X, xlab = "x", ylab = "y", pch = 20)

> print(colMeans(X))
[1] -0.001628189  0.023474775

> print(cor(X))
          [,1]      [,2]
[1,] 1.0000000 0.8931007
[2,] 0.8931007 1.0000000
```

Output from Example 3.16 shows the sample mean vector is $(-0.002, 0.023)$ and sample correlation is 0.893, which agree closely with the specified parameters. The scatter plot of the sample data shown in Figure 3.6 exhibits the elliptical symmetry of multivariate normal distributions. ◇

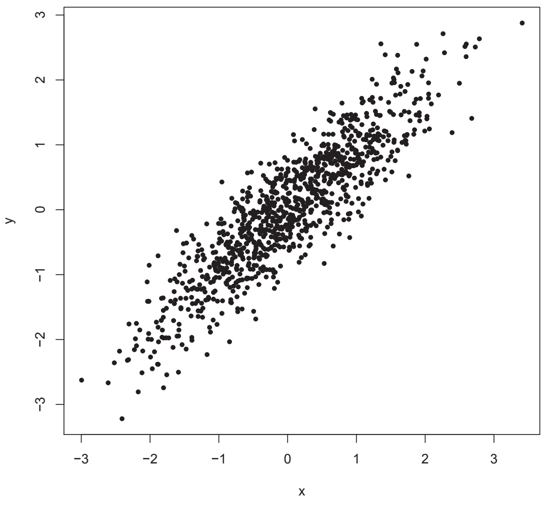

FIGURE 3.6: Scatterplot of a random bivariate normal sample with mean vector zero, variances $\sigma_1^2 = \sigma_2^2 = 1$ and correlation $\rho = 0.9$, from Example 3.16.

SVD Method of generating $N_d(\mu, \Sigma)$ samples

The singular value decomposition (svd) generalizes the idea of eigenvectors to rectangular matrices. The svd of a matrix X is $X = UDV^T$, where D is a vector containing the singular values of X, U is a matrix whose columns contain the left singular vectors of X, and V is a matrix whose columns contain the right singular vectors of X. The matrix X in this case is the population covariance matrix Σ, and $UV^T = I$. The svd of a symmetric positive definite matrix Σ gives $U = V = P$ and $\Sigma^{1/2} = UD^{1/2}V^T$. Thus the svd method for this application is equivalent to the spectral decomposition method, but is less efficient because the svd method does not take advantage of the fact that the matrix Σ is square symmetric.

Example 3.17 (SVD method)**.** This example provides a function `rmvn.svd` to generate a multivariate normal sample, using the svd method to factor Σ.

```
rmvn.svd <-
function(n, mu, Sigma) {
    # generate n random vectors from MVN(mu, Sigma)
    # dimension is inferred from mu and Sigma
    d <- length(mu)
    S <- svd(Sigma)
    R <- S$u %*% diag(sqrt(S$d)) %*% t(S$v) #sq. root Sigma
    Z <- matrix(rnorm(n*d), nrow=n, ncol=d)
    X <- Z %*% R + matrix(mu, n, d, byrow=TRUE)
    X
}
```

This function is applied in Example 3.19. ◇

Choleski factorization method of generating $N_d(\mu, \Sigma)$ samples

The Choleski factorization of a real symmetric positive-definite matrix is $X = Q^TQ$, where Q is an upper triangular matrix. The Choleski factorization is implemented in the R function `chol`. The basic syntax is `chol(X)` and the return value is an upper triangular matrix R such that $R^TR = X$.

Example 3.18 (Choleski factorization method)**.** The Choleski factorization method is applied to generate 200 random observations from a four-dimensional multivariate normal distribution.

```
rmvn.Choleski <-
function(n, mu, Sigma) {
    # generate n random vectors from MVN(mu, Sigma)
    # dimension is inferred from mu and Sigma
    d <- length(mu)
    Q <- chol(Sigma) # Choleski factorization of Sigma
    Z <- matrix(rnorm(n*d), nrow=n, ncol=d)
    X <- Z %*% Q + matrix(mu, n, d, byrow=TRUE)
    X
}
```

In this example, we will generate the samples according to the same mean and covariance structure as the four-dimensional iris virginica data.

```
y <- subset(x=iris, Species=="virginica")[, 1:4]
mu <- colMeans(y)
```

```
Sigma <- cov(y)

> mu
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
       6.588        2.974        5.552        2.026
> Sigma
             Sepal.Length Sepal.Width Petal.Length Petal.Width
Sepal.Length   0.40434286  0.09376327   0.30328980  0.04909388
Sepal.Width    0.09376327  0.10400408   0.07137959  0.04762857
Petal.Length   0.30328980  0.07137959   0.30458776  0.04882449
Petal.Width    0.04909388  0.04762857   0.04882449  0.07543265

#now generate MVN data with this mean and covariance
X <- rmvn.Choleski(200, mu, Sigma)
pairs(X)
```

The pairs plot of the data in Figure 3.7 gives a 2-D view of the bivariate distribution of each pair of marginal distributions. The joint distribution of each pair of marginal distributions is theoretically bivariate normal. The plot can be compared with Figure 5.1, which displays the iris virginica data. (The iris virginica data are not multivariate normal, but means and correlation for each pair of variables should be similar to the simulated data.) ◇

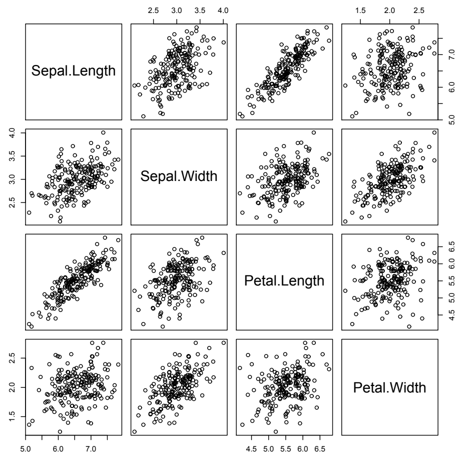

FIGURE 3.7: Pairs plot of the bivariate marginal distributions of a simulated multivariate normal random sample in Example 3.18. The parameters match the mean and covariance of the *iris virginica* data.

Remark 3.3. To standardize a multivariate normal sample, we invert the procedure above, substituting the sample mean vector and sample covariance matrix if the parameters are unknown. The transformed d-dimensional sample then has zero mean vector and covariance I_d. This is not the same as scaling the columns of the data matrix. ◇

Comparing Performance of Generators

We have discussed several methods for generating random samples from specified probability distributions. When several methods are available, which method is preferred? One consideration may be the computational time required (the time complexity). Another important consideration, if the purpose of the simulation is to estimate one or more parameters, is the variance of the estimator. The latter topic is considered in Chapter 6. To compare the empirical performance with respect to computing time, we can time each procedure.

R provides the `system.time` function, which times the evaluation of its argument. This function can be used as a rough benchmark to compare the performance of different algorithms. In the next example, the `system.time` function is used to compare the CPU time required for several different methods of generating multivariate normal samples.

Example 3.19 (Comparing performance of MVN generators)**.** This example generates multivariate normal samples in a higher dimension ($d = 30$) and compares the timing of each of the methods presented in Section 3.6.1 and two generators available in R packages. This example uses a function `rmvnorm` in the package `mvtnorm` [121]. This package is not part of the standard R distribution, but can be installed from CRAN. The `MASS` package [293] is one of the recommended packages included with the R distribution.

```
library(MASS)
library(mvtnorm)
n <- 100          #sample size
d <- 30           #dimension
N <- 2000         #iterations
mu <- numeric(d)

set.seed(100)
system.time(for (i in 1:N)
    rmvn.eigen(n, mu, cov(matrix(rnorm(n*d), n, d))))
set.seed(100)
system.time(for (i in 1:N)
    rmvn.svd(n, mu, cov(matrix(rnorm(n*d), n, d))))
set.seed(100)
system.time(for (i in 1:N)
    rmvn.Choleski(n, mu, cov(matrix(rnorm(n*d), n, d))))
set.seed(100)
system.time(for (i in 1:N)
    mvrnorm(n, mu, cov(matrix(rnorm(n*d), n, d))))
set.seed(100)
system.time(for (i in 1:N)
    rmvnorm(n, mu, cov(matrix(rnorm(n*d), n, d))))
```

```
set.seed(100)
system.time(for (i in 1:N)
    cov(matrix(rnorm(n*d), n, d)))
```

Most of the work involved in generating a multivariate normal sample is the factorization of the covariance matrix. The covariances used for this example are actually the sample covariances of standard multivariate normal samples. Thus, the randomly generated Σ varies with each iteration, but Σ is close to an identity matrix. In order to time each method on the same covariance matrices, the random number seed is restored before each run. The last run simply generates the covariances, for comparison with the total time.

The results below (summarized from the console output) suggest that there are differences in performance among these five methods when the covariance matrix is close to identity. The Choleski method is somewhat faster, while `rmvn.eigen` and `mvrnorm` (`MASS`) [293] appear to perform about equally well. The similar performance of `rmvn.eigen` and `mvrnorm` is not surprising, because according to the documentation for `mvrnorm`, the method of matrix decomposition is the eigendecomposition. Documentation for `mvrnorm` states that "although a Choleski decomposition might be faster, the eigendecomposition is stabler."

```
Timings of MVN generators

               user  system elapsed
rmvn.eigen     1.78    0.00    1.77
rmvn.svd       2       0       2
rmvn.choleski  1.22    0.02    1.27
mvrnorm        1.77    0.00    1.77
rmvnorm        2.17    0.00    2.17
generate Sigma 0.64    0.00    0.65
```

◇

The `system.time` function was also used to compare the methods in Examples 4.1 and 4.2. The code (not shown) is similar to the examples above.

See Chapter 15, on the topic of benchmarking, for more convenient methods of comparing running times of functions. Example 15.3 shows a simpler way to compare the timings above that automatically produces a table and relative timings.

3.6.2 Mixtures of Multivariate Normals

A multivariate normal mixture is denoted

$$pN_d(\mu_1, \Sigma_1) + (1-p)N_d(\mu_2, \Sigma_2) \tag{3.7}$$

where the sampled population is $N_d(\mu_1, \Sigma_1)$ with probability p, and $N_d(\mu_2, \Sigma_2)$ with probability $1-p$. As the mixing parameter p and other parameters are varied, the multivariate normal mixtures have a wide variety of types of departures from normality. For example, a 50% normal location mixture is symmetric with light tails, and a 90% normal location mixture is skewed with heavy tails. A normal location mixture with $p = 1 - \frac{1}{2}(1 - \frac{\sqrt{3}}{3}) \doteq 0.7887$, provides an example of a skewed distribution with normal kurtosis [147]. Parameters can be varied to generate a wide variety of distributional shapes. Johnson [160] gives many examples for the bivariate normal mixtures. Many commonly applied statistical procedures do not perform well under this type of departure from normality, so normal mixtures are often chosen to compare the properties of competing robust methods of analysis.

If X has the distribution (3.7), then a random observation from the distribution of X can be generated as follows.

To generate a random sample from $pN_d(\mu_1, \Sigma_1) + (1-p)N_d(\mu_2, \Sigma_2)$

1. Generate $U \sim$ Uniform(0,1).
2. If $U \leq p$ generate X from $N_d(\mu_1, \Sigma_1)$;
   otherwise generate X from $N_d(\mu_2, \Sigma_2)$.

The following procedure is equivalent.

1. Generate $N \sim$ Bernoulli(p).
2. If $N = 1$ generate X from $N_d(\mu_1, \Sigma_1)$;
   otherwise generate X from $N_d(\mu_2, \Sigma_2)$.

Example 3.20 (Multivariate normal mixture)**.** Write a function to generate a multivariate normal mixture with two components. The components of a location mixture differ in location only. Use the `mvrnorm(MASS)` function [293] to generate the multivariate normal observations.

First we write this generator in an inefficient loop to clearly illustrate the steps outlined above. (We will eliminate the loop later.)

```
library(MASS)  #for mvrnorm
#inefficient version loc.mix.0 with loops

loc.mix.0 <- function(n, p, mu1, mu2, Sigma) {
    #generate sample from BVN location mixture
    X <- matrix(0, n, 2)

    for (i in 1:n) {
        k <- rbinom(1, size = 1, prob = p)
        if (k)
            X[i,] <- mvrnorm(1, mu = mu1, Sigma) else
            X[i,] <- mvrnorm(1, mu = mu2, Sigma)
        }
    return(X)
}
```

Although the code above will generate the required mixture, the loop is rather inefficient. Generate n_1, the number of observations realized from the first component, from Binomial(n, p). Generate n_1 variates from component 1 and $n_2 = n - n_1$ from component 2 of the mixture. Generate a random permutation of the indices 1:n to indicate the order in which the sample observations appear in the data matrix.

```
#more efficient version

loc.mix <- function(n, p, mu1, mu2, Sigma) {
    #generate sample from BVN location mixture
    n1 <- rbinom(1, size = n, prob = p)
    n2 <- n - n1
    x1 <- mvrnorm(n1, mu = mu1, Sigma)
    x2 <- mvrnorm(n2, mu = mu2, Sigma)
    X <- rbind(x1, x2)            #combine the samples
    return(X[sample(1:n), ])      #mix them
}
```

To illustrate the normal mixture generator, we apply `loc.mix` to generate a random sample of $n = 1000$ observations from a 50% 4-dimensional normal location mixture with $\mu_1 = (0, 0, 0, 0)$ and $\mu_2 = (2, 3, 4, 5)$ and covariance I_4.

```
x <- loc.mix(1000, .5, rep(0, 4), 2:5, Sigma = diag(4))
r <- range(x) * 1.2
par(mfrow = c(2, 2))
for (i in 1:4)
    hist(x[ , i], xlim = r, ylim = c(0, .3), freq = FALSE,
    main = "", breaks = seq(-5, 10, .5))
par(mfrow = c(1, 1))
```

It is difficult to visualize data in $\mathbb{R}^4$, so we display only the histograms of the marginal distributions in Figure 3.8. All of the one-dimensional marginal distributions are univariate normal location mixtures. Methods for visualization of multivariate data are covered in Chapter 5. Also, an interesting view of a bivariate normal mixture with three components is shown in Figure 12.13. $\diamond$

3.6.3 Wishart Distribution

Suppose $M = X^T X$, where X is an $n \times d$ data matrix of a random sample from a $N_d(\mu, \Sigma)$ distribution. Then M has a Wishart distribution with scale matrix Σ and n degrees of freedom, denoted $M \sim W_d(\Sigma, n)$ (see, e.g., [13, 194]). Note that when $d = 1$, the elements of X are a univariate random sample from $N(\mu, \sigma^2)$ so $W_1(\sigma^2, n) \overset{D}{=} \sigma^2\chi^2(n)$.

An obvious, but inefficient, approach to generating random variates from a Wishart distribution is to generate multivariate normal random samples and compute the matrix product $X^T X$. This method is computationally expensive

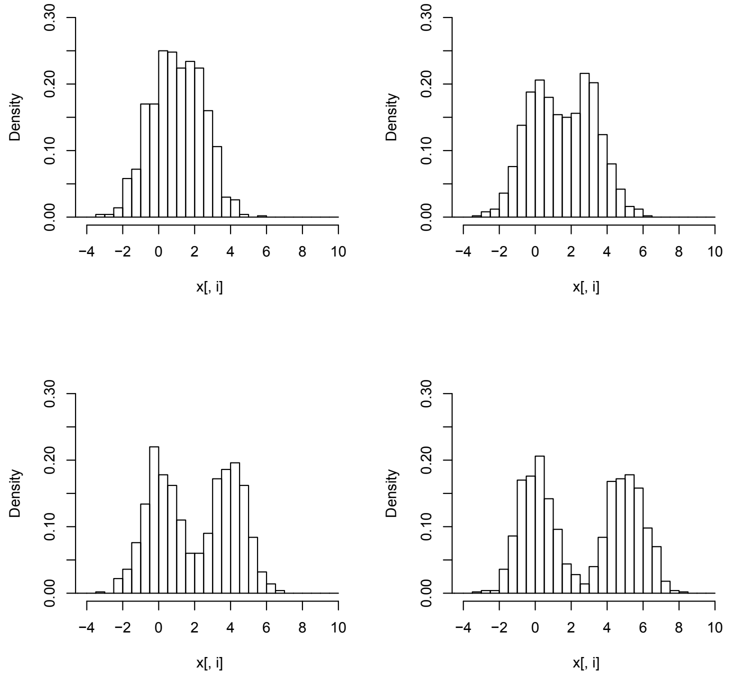

FIGURE 3.8: Histograms of the marginal distributions of multivariate normal location mixture data generated in Example 3.20.

because nd random normal variates must be generated to determine the $d(d+1)/2$ distinct entries in M.

A more efficient method based on Bartlett's decomposition [25] is summarized by Johnson [160, p. 204] as follows. Let $T = (T_{ij})$ be a lower triangular $d \times d$ random matrix with independent entries satisfying

1. $T_{ij} \overset{iid}{\sim} N(0, 1)$, $i > j$.
2. $T_{ii} \sim \sqrt{\chi^2(n - i + 1)}$, $i = 1, \dots, d$.

Then the matrix $A = TT^T$ has a $W_d(I_d, n)$ distribution. To generate $W_d(\Sigma, n)$ random variates, obtain the Choleski factorization $\Sigma = LL^T$, where L is lower triangular. Then $LAL^T \sim W_d(\Sigma, n)$ [25, 140, 218]. Implementation is left as an exercise.

3.6.4 Uniform Distribution on the d-Sphere

The d-sphere is the set of all points $x \in \mathbb{R}^d$ such that $\|x\| = (x^T x)^{1/2} = 1$. Random vectors uniformly distributed on the d-sphere have equally likely directions. A method of generating this distribution uses a property of the multivariate normal distribution (see [97, 160]). If $X_1, \dots, X_d$ are iid $N(0,1)$, then $U = (U_1, \dots, U_d)$ is uniformly distributed on the unit sphere in $\mathbb{R}^d$, where

$$U_j = \frac{X_j}{(X_1^2 + \cdots + X_d^2)^{1/2}}, \quad j = 1, \dots, d. \tag{3.8}$$

Algorithm to generate uniform variates on the d-Sphere

1. For each variate u_i, $i = 1, \dots, n$ repeat
   (a) Generate a random sample $x_{i1}, \dots, x_{id}$ from $N(0,1)$.
   (b) Compute the Euclidean norm $\|x_i\| = (x_{i1}^2 + \cdots + x_{id}^2)^{1/2}$.
   (c) Set $u_{ij} = x_{ij}/\|x_i\|$, $j = 1, \dots, d$.
   (d) Deliver $u_i = (u_{i1}, \dots, u_{id})$.

To implement these steps efficiently in R for a sample size n,

1. Generate nd univariate normals in $n \times d$ matrix `M`. The i^{th} row of `M` corresponds to the i^{th} random vector u_i.

2. Compute the denominator of (3.8) for each row, storing the n norms in vector `L`.

3. Divide each number `M[i,j]` by the norm `L[i]`, to get the matrix `U`, where `U[i,]` $= u_i = (u_{i1}, \dots, u_{id})$.

4. Deliver matrix `U` containing n random observations in rows.

Example 3.21 (Generating variates on a sphere)**.** This example provides a function to generate random variates uniformly distributed on the unit d-sphere.

```
runif.sphere <- function(n, d) {
    # return a random sample uniformly distributed
    # on the unit sphere in R ^d
    M <- matrix(rnorm(n*d), nrow = n, ncol = d)
    L <- apply(M, MARGIN = 1,
               FUN = function(x){sqrt(sum(x*x))})
    D <- diag(1 / L)
    U <- D %*% M
    U
}
```

The function `runif.sphere` is used to generate a sample of 200 points uniformly distributed on the circle.

```
#generate a sample in d=2 and plot
X <- runif.sphere(200, 2)
par(pty = "s")
plot(X, xlab = bquote(x[1]), ylab = bquote(x[2]))
par(pty = "m")
```

The circle of points is shown in Figure 3.9. ◇

R Note 3.8

The `apply` function in `runif.sphere` returns a vector containing the n norms $\|x_1\|, \|x_2\|, \ldots, \|x_n\|$ of the sample vectors in matrix `M`.

R Note 3.9

The command `par(pty = "s")` sets the square plot type so the circle is round rather than elliptical; `par(pty = "m")` restores the type to maximal plotting region. See the help topic `?par` for other plot parameters.

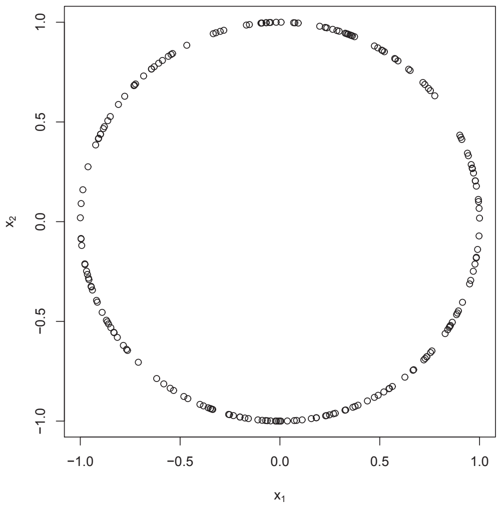

FIGURE 3.9: A random sample of 200 points from the bivariate distribution (X_1, X_2) that is uniformly distributed on the unit circle in Example 3.21.

Uniformly distributed points on a hyperellipsoid can be generated by applying a suitable linear transformation to a Uniform sample on the d-sphere. Fishman [97, 3.28] gives an algorithm for generating points in and on a simplex.

Exercises

3.1 Write a function that will generate and return a random sample of size n from the two-parameter exponential distribution $\mathrm{Exp}(\lambda, \eta)$ for arbitrary n, λ, and η. (See Examples 2.3 and 2.6.) Generate a large sample from $\mathrm{Exp}(\lambda, \eta)$ and compare the sample quantiles with the theoretical quantiles.

3.2 The standard Laplace distribution has density $f(x) = \frac{1}{2}e^{-|x|}$, $x \in \mathbb{R}$. Use the inverse transform method to generate a random sample of size 1000 from this distribution. Use one of the methods shown in this chapter to compare the generated sample to the target distribution.

3.3 The Pareto(a, b) distribution has cdf

$$F(x) = 1 - \left(\frac{b}{x}\right)^a, \qquad x \geq b > 0, a > 0.$$

Derive the probability inverse transformation $F^{-1}(U)$ and use the inverse transform method to simulate a random sample from the Pareto(2, 2) distribution. Graph the density histogram of the sample with the Pareto(2, 2) density superimposed for comparison.

3.4 The Rayleigh density [162, Ch. 18] is

$$f(x) = \frac{x}{\sigma^2} e^{-x^2/(2\sigma^2)}, \qquad x \geq 0,\ \sigma > 0.$$

Develop an algorithm to generate random samples from a Rayleigh(σ) distribution. Generate Rayleigh(σ) samples for several choices of $\sigma > 0$ and check that the mode of the generated samples is close to the theoretical mode σ (check the histogram).

3.5 A discrete random variable X has probability mass function

x	0	1	2	3	4
$p(x)$	0.1	0.2	0.2	0.2	0.3

Use the inverse transform method to generate a random sample of size 1000 from the distribution of X. Construct a relative frequency table and compare the empirical with the theoretical probabilities. Repeat using the R `sample` function.

3.6 Prove that the accepted variates generated by the acceptance-rejection sampling algorithm are a random sample from the target density f_X.

3.7 Write a function to generate a random sample of size n from the Beta(a, b) distribution by the acceptance-rejection method. Generate a random sample of size 1000 from the Beta(3,2) distribution. Graph the histogram of the sample with the theoretical Beta(3,2) density superimposed.

3.8 Write a function to generate random variates from a Lognormal(μ, σ) distribution using a transformation method, and generate a random sample of size 1000. Compare the histogram with the lognormal density curve given by the `dlnorm` function in R.

3.9 The rescaled Epanechnikov kernel [92] is a symmetric density function

$$f_e(x) = \frac{3}{4}(1 - x^2), \qquad |x| \le 1. \tag{3.9}$$

Devroye and Györfi [74, p. 236] give the following algorithm for simulation from this distribution. Generate iid $U_1, U_2, U_3 \sim$ Uniform$(-1, 1)$. If $|U_3| \ge |U_2|$ and $|U_3| \ge |U_1|$, deliver U_2; otherwise deliver U_3. Write a function to generate random variates from f_e, and construct the histogram density estimate of a large simulated random sample.

3.10 Prove that the algorithm given in Exercise 3.9 generates variates from the density f_e (3.9).

3.11 Generate a random sample of size 1000 from a normal location mixture. The components of the mixture have $N(0, 1)$ and $N(3, 1)$ distributions with mixing probabilities p_1 and $p_2 = 1 - p_1$. Graph the histogram of the sample with density superimposed, for $p_1 = 0.75$. Repeat with different values for p_1 and observe whether the empirical distribution of the mixture appears to be bimodal. Make a conjecture about the values of p_1 that produce bimodal mixtures.

3.12 Simulate a continuous Exponential-Gamma mixture. Suppose that the rate parameter Λ has Gamma(r, β) distribution and Y has Exp(Λ) distribution. That is, $(Y|\Lambda = \lambda) \sim f_Y(y|\lambda) = \lambda e^{-\lambda y}$. Generate 1000 random observations from this mixture with $r = 4$ and $\beta = 2$.

3.13 It can be shown that the mixture in Exercise 3.12 has a Pareto distribution with cdf

$$F(y) = 1 - \left(\frac{\beta}{\beta + y}\right)^r, \quad y \ge 0.$$

(This is an alternative parameterization of the Pareto cdf given in Exercise 3.3.) Generate 1000 random observations from the mixture with $r = 4$ and $\beta = 2$. Compare the empirical and theoretical (Pareto) distributions by graphing the density histogram of the sample and superimposing the Pareto density curve.

3.14 Generate 200 random observations from the 3-dimensional multivariate normal distribution having mean vector $\mu = (0, 1, 2)$ and covariance matrix

$$\Sigma = \begin{bmatrix} 1.0 & -0.5 & 0.5 \\ -0.5 & 1.0 & -0.5 \\ 0.5 & -0.5 & 1.0 \end{bmatrix}$$

using the Choleski factorization method. Use the R `pairs` plot to graph an array of scatter plots for each pair of variables. For each pair of variables, (visually) check that the location and correlation approximately agree with the theoretical parameters of the corresponding bivariate normal distribution.

3.15 Write a function that will standardize a multivariate normal sample for arbitrary n and d. That is, transform the sample so that the sample mean vector is zero and sample covariance is the identity matrix. To check your results, generate multivariate normal samples and print the sample mean vector and covariance matrix before and after standardization.

3.16 Efron and Tibshirani discuss the `scor (bootstrap)` test score data on 88 students who took examinations in five subjects [91, Table 7.1], [194, Table 1.2.1]. Each row of the data frame is a set of scores $(x_{i1}, \dots, x_{i5})$ for the i^{th} student. Standardize the scores by type of exam. That is, standardize the bivariate samples (X_1, X_2) (closed book) and the trivariate samples (X_3, X_4, X_5) (open book). Compute the covariance matrix of the transformed sample of test scores.

3.17 Compare the performance of the Beta generator of Exercise 3.7, Example 3.8 and the R generator `rbeta`. Fix the parameters $a = 2, b = 2$ and time each generator on 1000 iterations with sample size 5000. (See Example 3.19.) Are the results different for different choices of a and b?

3.18 Write a function to generate a random sample from a $W_d(\Sigma, n)$ (Wishart) distribution for $n > d + 1 \geq 1$, based on Bartlett's decomposition.