Lab2: Methods for Generating Random Variables

Lecturer: Zhao Jianhua

Department of Statistics
Yunnan University of Finance and Economics

Task

The objective in this lab is to learn the methods for generating random variables, including the inverse transform method in both continuous and discrete cases, the acceptance-rejection method, transformation methods, sums and mixtures, multivariate distributions such as multivariate normal distribution, mixtures of multivariate normals, Wishart Distribution and uniform dist. on the d-Sphere.

1 Introduction

Generating Uniform random number

```
runif(n)  #generate a vector of size n in [0,1]
runif(n, a, b)  #generate n Uniform(a, b) numbers
matrix(runif(n * m), n, m)  #generate n by m matrix
```

The `sample` function can be used to sample from a finite population, with or without replace-

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

##  [1] 1 1 1 0 0 0 0 1 0 0

# choose some lottery numbers
sample(1:100, size = 6, replace = FALSE)

## [1] 38 79 98 44 26 85

# permuation of letters a-z
sample(letters)

##  [1] "e" "k" "i" "z" "p" "f" "l" "s" "d" "h" "m" "j" "x" "o" "w" "b" "r"
## [18] "u" "g" "v" "q" "t" "c" "y" "n" "a"

# sample from a multinomial dist.
x = sample(1:3, size = 100, replace = TRUE, prob = c(0.2, 0.3, 0.5))
table(x)

## x
##  1  2  3
## 15 30 55
```

1.1 Random Generators of Common Probability Distributions in R

The prob. mass function (pmf) or density (pdf), cumulative dist. function (cdf), quantile function, and random generator of many commonly used prob. dist. are available. For example:

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

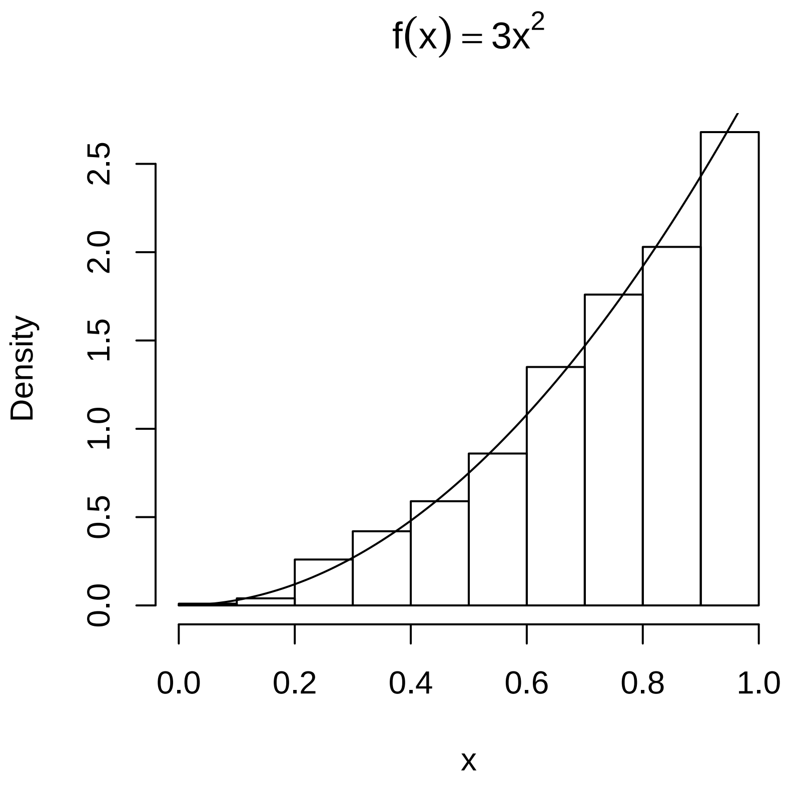
2 The Inverse Transform Method

2.1 Inverse Transform Method, Continuous Case

2.1.1 Example 3.2

Simulate a random sample from the dist. with density $f_X(x) = 3x^2, 0 < x < 1$. Here $F_X(x) = x^3$ for $0 < x < 1$, and $F_X^{-1}(u) = u^{1/3}$, then

```
n <- 1000
u <- runif(n)
x <- u^(1/3)
hist(x, prob = TRUE, main = expression(f(x) == 3 * x^2)) #density histogram of sample
y <- seq(0, 1, 0.01)
lines(y, 3 * y^2)  #density curve f(x)
```



Compare the timing of three different programing ways

```
# 1. using loop without preallocation of x
n = 1e+05
ptm = proc.time()  #start proc
x = 0
for (i in 1:n) {
    u = runif(1)
    x[i] = u^(1/3)
}
tmp = proc.time() - ptm  #end proc
T[1] = tmp[1]

# 2. using loop with preallocation of x
ptm = proc.time()  #start proc
x = rep(0, n)
for (i in 1:n) {
    u = runif(1)
    x[i] = u^(1/3)
}
tmp = proc.time() - ptm  #end proc
T[2] = tmp[1]

# 3. using vectorized programing (avoid using loop)
ptm = proc.time()  #start proc
u <- runif(n)
y <- u^(1/3)
tmp = proc.time() - ptm  #end proc
T[3] = tmp[1]
T

## [1] 40.12  1.29  0.06
```

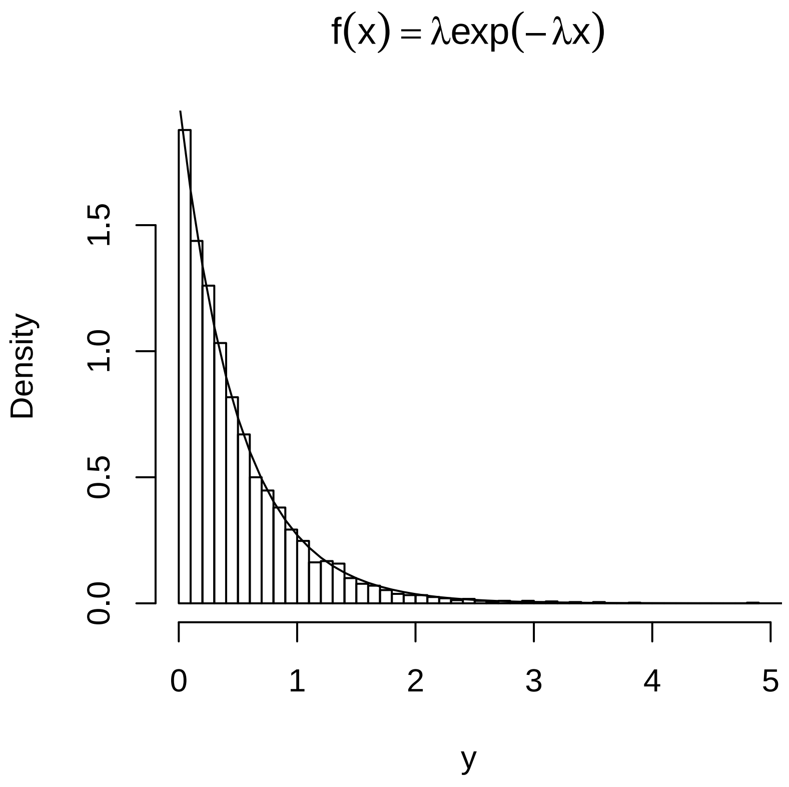
Using loop without preallocation of x took 40.12 and is the most inefficient. Vectorized programing took 0.06 only and is the most efficienttooks 40.12 only. In generaral, it is very important to avoid unnecessary loops and preallocate variables for efficient programming in R.

2.1.2 Example 3.3 (Exponential dist.)

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

```
n = 4000
lambda = 2
ptm = proc.time()  #start proc
# not using loop
u <- runif(n)
y <- -log(u)/lambda
proc.time() - ptm  #end the clock

##    user  system elapsed
##    0.01    0.00    0.02
```

```
hist(y, prob = TRUE, breaks = 50, main = expression(f(x) == lambda * exp(-lambda *
    x)))  #   density histogram of sample
y <- seq(0, 100, 0.1)
lines(y, lambda * exp(-lambda * y))  # density curve f(x)
```



Write your own function to generate Exponential r.v.

```
myexp = function(n, lambda) {
    u <- runif(n)
    x <- -log(u)/lambda
}
x = myexp(100, 3)
```

2.2 Inverse Transform Method, Discrete Case

2.2.1 Example 3.4 (Two point distribution)

Generate a random sample of Bernoulli($p = 0.4$) variates. 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 $F_X(u) = 0$ if $u \leq 0.6$,

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

mean(x)
```

```
## [1] 0.398

var(x)

## [1] 0.2398358
```

Sample mean and variance should be approximately $p = 0.398$ and $p(1-p) = 0.2398358$.

2.2.2 Example 3.5 (Geometric distribution)

Generate a random geometric sample with parameter $p = \frac{1}{4}$.
The pmf is $f(x) = pq^x$, $x = 0, 1, 2, \cdots$, where $q = 1 - p$. At the points of discontinuity $x = 0, 1, 2, \cdots$, cdf is $F(x) = 1 - q^{x+1}$. For each sample element, we generate a random uniform u and solve

$$1 - q^x < u \leq 1 - q^x,$$

which simplifies to $x < \log(1-u)/\log(q) \leq x + 1$. Thus $x + 1 = \lceil \log(1-u)/\log(q) \rceil$, where $\lceil t \rceil$ denotes ceiling function.

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

U and $1 - U$ have the same dist. and the prob. that $\log(1-u)/\log(1-p)$ equals an integer is zero, and thus the last step is

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

2.2.3 Example 3.6 (Logarithmic distribution)

Simulate a Logarithmic(θ) random sample

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

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

$$f(x+1) = \frac{\theta x}{x+1} f(x), x = 1, 2, \cdots \tag{3.2}$$

```
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
}
set.seed(10)
n <- 1e+05
theta <- 0.5
ptm = proc.time()
x <- rlogarithmic(n, theta)
proc.time() - ptm

##    user  system elapsed
##    0.91    0.02    0.92

# 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

round(rbind(table(x)/n, p, se), 3)

##          1     2     3     4     5     6     7     8 9  10  11  12  13
##      0.724 0.179 0.060 0.022 0.009 0.003 0.002 0.001 0   0   0   0   0
## p    0.721 0.180 0.060 0.023 0.009 0.004 0.002 0.001 0   0   0   0   0
## se   0.001 0.001 0.001 0.000 0.000 0.000 0.000 0.000 0   0   0   0   0
```

Example 3.6 (Logarithmic distribution, more efficient version by vectorization)

```
rlogarithmic <- function(n, theta) {
    u <- runif(n)
    # set the initial length of cdf vector
    N <- ceiling(-16/log10(theta))
    k <- 1:N
    a <- -1/log(1 - theta)
    fk <- a * theta^k/k
    Fk <- cumsum(fk)
    Mu = rep(1, N) %*% t(u)
    y = as.integer(colSums(Mu > Fk))
    I = which(y == N)
    n1 = length(I)
    if (n1 > 0) {
        for (i in 1:n1) {
            while (y[I[i]] == N) {
              # if x==N we need to extend the cdf very unlikely because N is large
                f <- a * theta^(N + 1)/N
                fk <- c(fk, f)
                Fk <- c(Fk, Fk[N] + fk[N + 1])
                N <- N + 1
                y[I[i]] <- as.integer(sum(u[I[i]] > Fk))
            }
        }
    }
    y + 1
}
set.seed(10)
n <- 1e+05
theta <- 0.5
ptm = proc.time()
x <- rlogarithmic(n, theta)
proc.time() - ptm

##    user  system elapsed
##    0.25    0.03    0.28

# 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

round(rbind(table(x)/n, p, se), 3)

##        1     2     3     4     5     6     7     8 9  10  11  12  13
##    0.724 0.179 0.060 0.022 0.009 0.003 0.002 0.001 0   0   0   0   0
## p  0.721 0.180 0.060 0.023 0.009 0.004 0.002 0.001 0   0   0   0   0
## se 0.001 0.001 0.001 0.000 0.000 0.000 0.000 0.000 0   0   0   0   0
```

3 The Acceptance-Rejection Method

The Beta(2,2) density is $f(x) = 6x(1-x), 0 < x < 1$. Let $g(x)$ be the $U(0,1)$ 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

$$f(x)/cg(x) = x(1-x) > u.$$

Averagely, cn = 6000 iterations will be required for sample size 1000.

Example 3.7 (Acceptance-rejection method)

```
ptm = proc.time()
```

```
n <- 1e+05
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
    }
}
proc.time() - ptm

##    user  system elapsed
##    8.99    0.07    9.08

# compare empirical and theoretical percentiles
p <- seq(0.1, 0.9, 0.1)
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
round(rbind(Qhat, Q, se), 3)

##        10%   20%   30%   40%   50%   60%   70%   80%   90%
## Qhat 0.196 0.288 0.364 0.435 0.502 0.567 0.636 0.712 0.804
## Q    0.196 0.287 0.363 0.433 0.500 0.567 0.637 0.713 0.804
## se   0.001 0.001 0.001 0.001 0.001 0.001 0.001 0.001 0.001
```

Example 3.7 (Acceptance-rejection method), more efficient version by vectorization

```
ptm = proc.time()
n = 1e+05
c = 6
cn = c * n
x = runif(cn)
u = runif(cn)
y = numeric(n)
ytmp = x[x * (1 - x) > u]
n1 = length(ytmp)
if (n1 >= n) y = ytmp[1:n] else {
    y[1:n1] = ytmp
    k <- n1 + 1  #counter for accepted
    j <- 0  #iterations
    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
        }
    }
}
proc.time() - ptm

##    user  system elapsed
##    0.19    0.02    0.20

# compare empirical and theoretical percentiles
p <- seq(0.1, 0.9, 0.1)
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
round(rbind(Qhat, Q, se), 3)

##        10%   20%   30%   40%   50%   60%   70%   80%   90%
## Qhat 0.195 0.287 0.363 0.432 0.501 0.569 0.638 0.713 0.804
## Q    0.196 0.287 0.363 0.433 0.500 0.567 0.637 0.713 0.804
## se   0.001 0.001 0.001 0.001 0.001 0.001 0.001 0.001 0.001
```

Sample percentiles (line 1) approximately match the Beta(2,2) percentiles computed by `qbeta` (line 2).

4 Transformation Methods

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

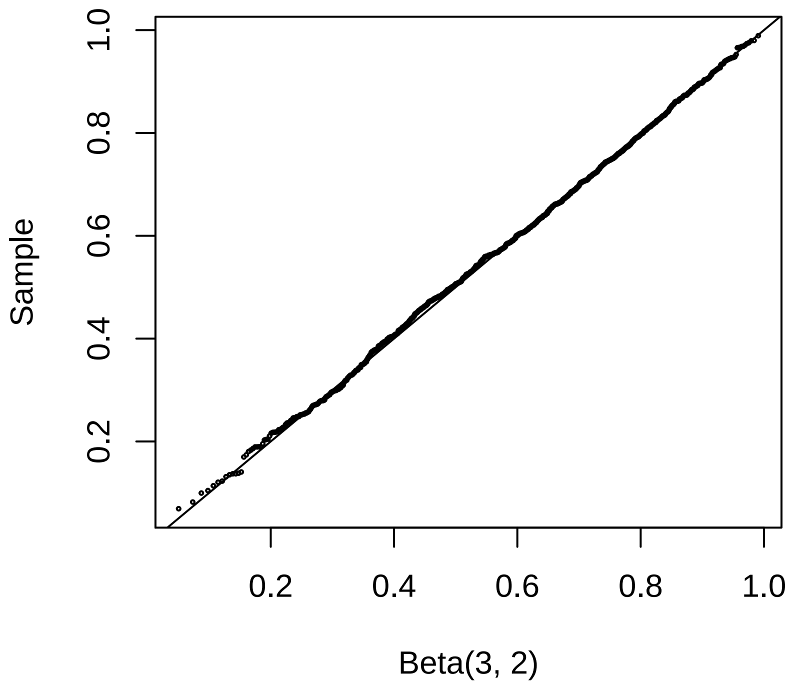
1. Generate u from Gamma$(a, 1)$.
2. Generate v from Gamma$(b, 1)$.
3. Deliver $x = u/(u + v)$.

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

4.1 Example 3.8 (Beta distribution)

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

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



The QQ plot of the ordered sample vs the Beta(3, 2) quantiles is very nearly linear.

4.2 Example 3.9 (Logarithmic dist., more efficient generator)

If U, V are independent $U(0,1)$ r.v., then $X = \lfloor 1 + \frac{\log(V)}{\log(1-(1-\theta)^U)} \rfloor$ has the Logarithmic(θ) dist.

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

```
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]  [,8]
## p.hat 0.716 0.171 0.066 0.032 0.007 0.004 0.002 0.002
## p     0.721 0.180 0.060 0.023 0.009 0.004 0.002 0.001
## se    0.014 0.012 0.008 0.005 0.003 0.002 0.001 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)
}
```

5 Sums and Mixtures

5.1 Example 3.10 (Chisquare)

Generate chisquare $\chi^2(\nu)$ random sample. If $Z_1, \cdots, Z_\nu$ are iid N(0,1) r.v., then $V = Z_1^2 + \cdots + Z_\nu^2$ has the $\chi^2(\nu)$ dist.

1. Fill an $n \times \nu$ matrix with ν r.v. from N(0,1).
2. Compute the row sums of the squared normals.
3. Deliver the vector of row sums.

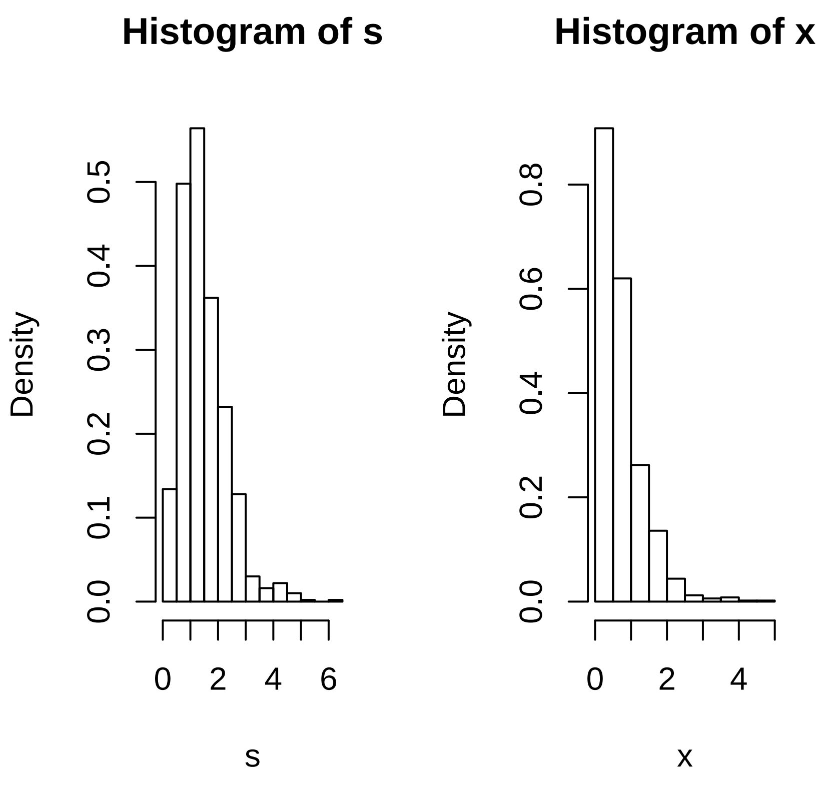
```
n <- 1000
nu <- 2
X <- matrix(rnorm(n * nu), n, nu)^2  #matrix of sq. normals
# sum the squared normals across each row: method 1
y <- rowSums(X)
# method 2
y <- apply(X, MARGIN = 1, FUN = sum)  #a vector length n
ym = mean(y)
y2m = mean(y^2)
rbind(ym, y2m)

##         [,1]
## ym  2.009791
## y2m 8.551426
```

Sample statistics agree with theoretical moments $E[Y] = \nu = 2$ and $E[Y^2] = 2\nu + \nu^2 = 8$. std. of sample moments are 2.0097906 and 8.5514256.

5.2 Example 3.11 (Convolutions and mixtures)

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

```r
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] 2.001382

mean(y^2)

## [1] 8.555974

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)
hist(x, prob = TRUE)
```



```r
par(mfcol = c(1, 1))  #restore display
```

Histograms of the convolution S and mixture X are different.

5.3 Example 3.12 (Mixture of several gamma distributions)

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 $X_j \sim$ Gamma$(r = 3, \lambda_j = 1/j)$ are independent and the mixing prob. are $\theta_j = j/15, j = 1, \cdots, 5$.

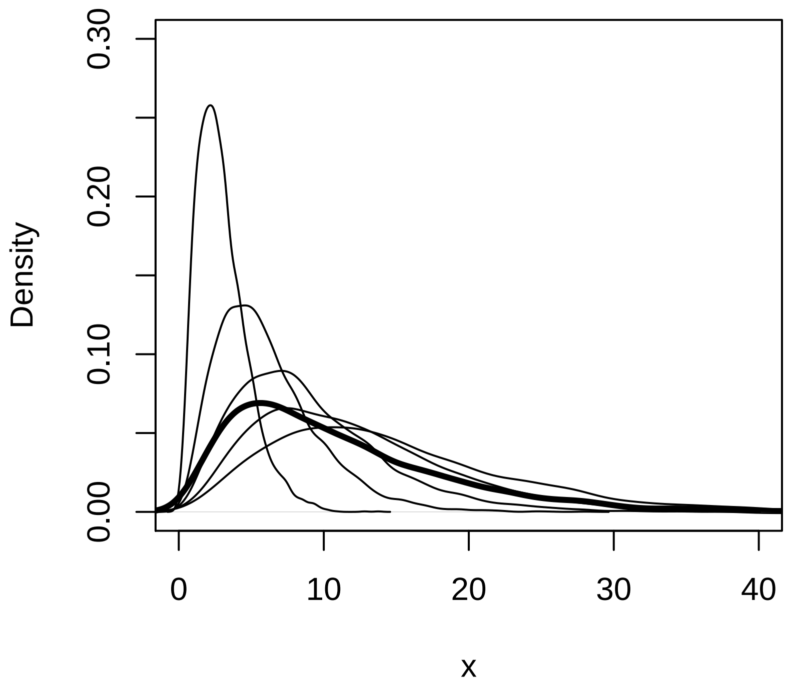
To simulate the mixture F_X:

1. Generate an integer $k \in \{1, 2, 3, 4, 5\}$, $P(k) = \theta_k, k = 1, \ldots, 5$.
2. Deliver a random Gamma(r, λ_k) variate.

which suggests using a `for` loop to generate a sample size n, but `for` loops are really inefficient in R.
Efficient vectorized algorithm:

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

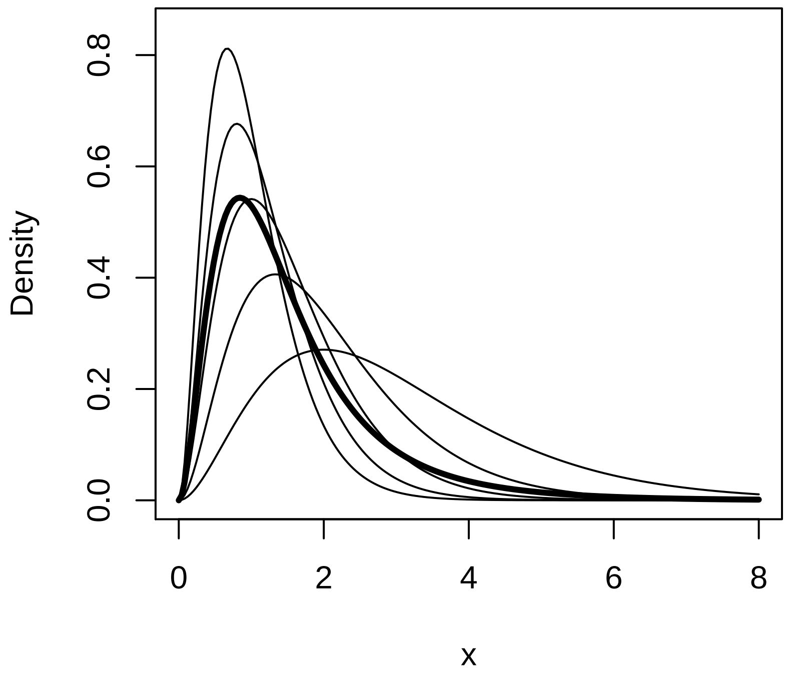
```r
# density estimates are plotted

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, 0.3), lwd = 3, xlab = "x", main = "")
for (i in 1:5) lines(density(rgamma(n, 3, 1/i)))
```



5.4 Example 3.14 (Plot density of mixture)

The density is $f(x) = \sum_{j=1}^{5} \theta_j f_j(x), x > 0$, where f_j is the Gamma(3,λ_j) density, with rates $\lambda = (1, 1.5, 2, 2.5, 3)$ and mixing prob. $\theta = (0.1, 0.2, 0.2, 0.3, 0.2)$.

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

p <- c(0.1, 0.2, 0.2, 0.3, 0.2)
lambda <- c(1, 1.5, 2, 2.5, 3)

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)

# plot the density of the mixture
plot(x, y, type = "l", ylim = c(0, 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)
}
```



5.5 Example 3.15 (Poisson-Gamma mixture)

If $(X|\Lambda = \lambda) \sim \text{Poisson}(\lambda)$ and $\Lambda \sim \text{Gamma}(r, \beta)$, then X has the negative binomial dist. with parameters r and $p = \beta/(1+\beta)$.

```
# 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)

round(rbind(mix, negbin, se), 3)

##         [,1]  [,2]  [,3]  [,4]  [,5]  [,6]  [,7]  [,8]  [,9] [,10] [,11]
## mix    0.311 0.299 0.212 0.095 0.046 0.022 0.007 0.003 0.001 0.002 0.001
## negbin 0.316 0.316 0.198 0.099 0.043 0.017 0.006 0.002 0.001 0.000 0.000
## se     0.015 0.015 0.013 0.009 0.006 0.004 0.002 0.001 0.001 0.000 0.000
##         [,12] [,13]
## mix         0 0.001
## negbin      0 0.000
## se          0 0.000
```

6 Multivariate Distributions

6.1 Multivariate Normal Distribution

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

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

Suppose that Σ can be factored as $\Sigma = CC^T$ for some matrix C. Then $CZ + \mu = N_d(\mu, \Sigma)$. Factorization of Σ can be obtained by the spectral decomposition method (eigenvector decomposition, `eigen`), Choleski factorization (`chol`), or singular value decomposition (`svd`).

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

1. Generate an n¡®d matrix Z containing nd random $N(0,1)$ variates
2. Compute a factorization $\Sigma = Q^TQ$.
3. Apply the transformation $X = ZQ + J_\mu{}^T$ and deliver X.

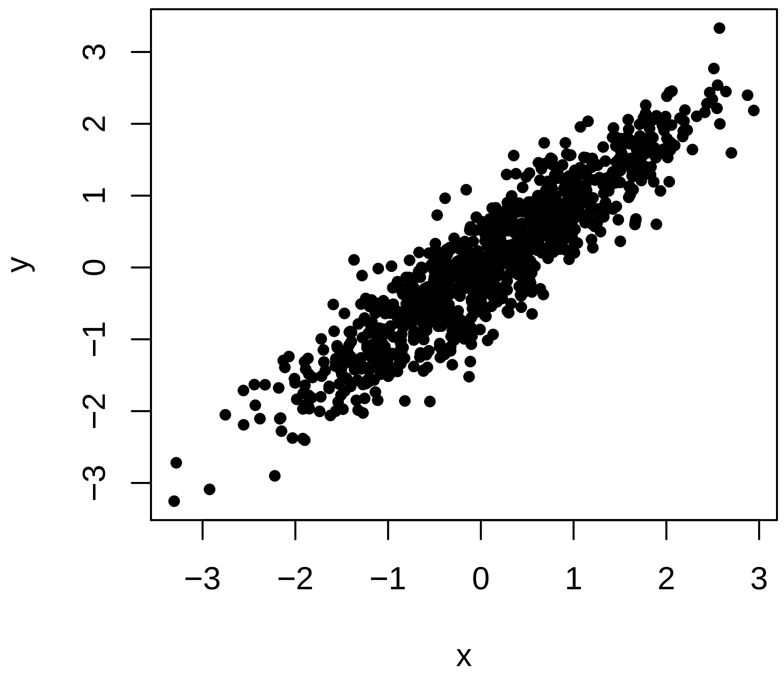
Generate a random sample from $N_d(\mu, \Sigma)$ with $\mu = (0,0)^T$ and

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

6.1.1 Example 3.16 (Spectral decomposition method)

Spectral decomposition: $\Sigma^{1/2} = P\Lambda^{1/2}P^{-1}$, $Q = \Sigma^{1/2}$, $Q^TQ = \Sigma$,
where Λ is the diagonal matrix with the eigenvalues of Σ along the diagonal and P is the matrix whose columns are the corresponding eigenvectors. This method also called eigen-decomposition, in which, $P^{-1} = P^T$ and $\Sigma^{1/2} = P\Lambda^{1/2}P^T$.

```r
# mean and covariance parameters
mu <- c(0, 0)
Sigma <- matrix(c(1, 0.9, 0.9, 1), nrow = 2, ncol = 2)

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
}

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

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



```
print(colMeans(X))

## [1] 0.09136876 0.06501544

print(cor(X))

##           [,1]      [,2]
## [1,] 1.0000000 0.9134794
## [2,] 0.9134794 1.0000000
```

The scatter plot exhibits the elliptical symmetry of multivariate normal dist.

6.1.2 Example 3.17 (SVD method)

SVD generalizes the idea of eigenvectors to rectangular matrices.

- svd: $X = UDV^T$, 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.

- Since $\Sigma \succ 0$ (positive definite), $UV^T = I$, thus $\Sigma^{1/2} = U\Lambda^{1/2}U^T$ and svd is equivalent to spectral decomposition.
- svd is less efficient because it does not take advantage of the fact that the matrix Σ is square symmetric.

```
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
}
```

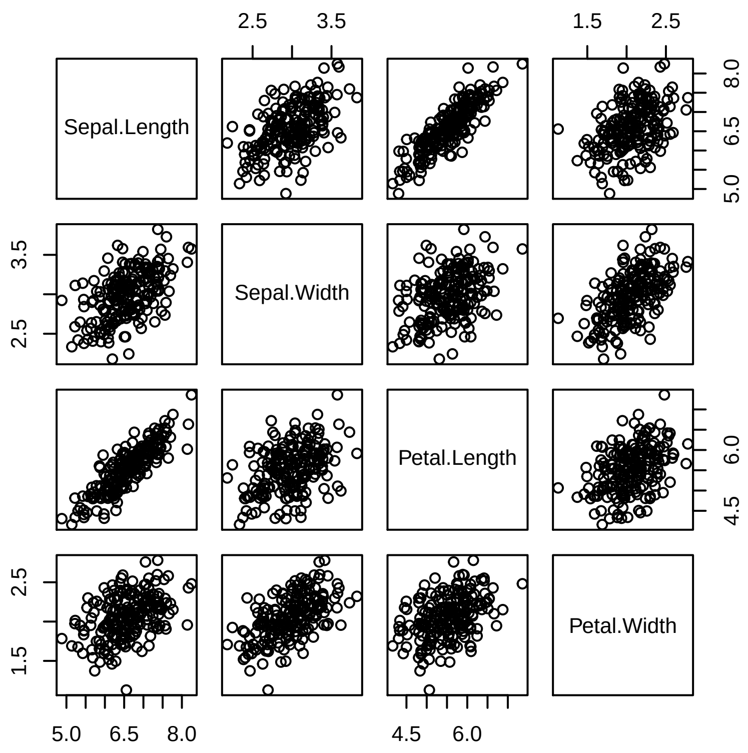
The scatter plot exhibits the elliptical symmetry of multivariate normal dist.

6.1.3 Example 3.18 (Choleski factorization method)

Choleski factorization: $X = Q^TQ(X \succ 0)$, where Q is an upper triangular matrix. The R syntax: `Q=chol(X)`.

```
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
}

# generating the samples according to the 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 joint distribution of each pair of marginal distributions is theoretically bivariate normal. The plot can be compared with Figure 4.1, which displays the iris virginica data.

6.1.4 Example 3.19 (Comparing performance of MVN generators)

```
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))))

##    user  system elapsed
##    4.77    0.00    4.78

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

##    user  system elapsed
##    5.44    0.03    5.46
```

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

##    user  system elapsed
##    3.47    0.00    3.49

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

##    user  system elapsed
##    4.60    0.00    4.61

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

##    user  system elapsed
##    5.69    0.00    6.20

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

##    user  system elapsed
##    1.81    0.00    1.81

detach(package:MASS)
detach(package:mvtnorm)
```

6.2 Mixtures of Multivariate Normals

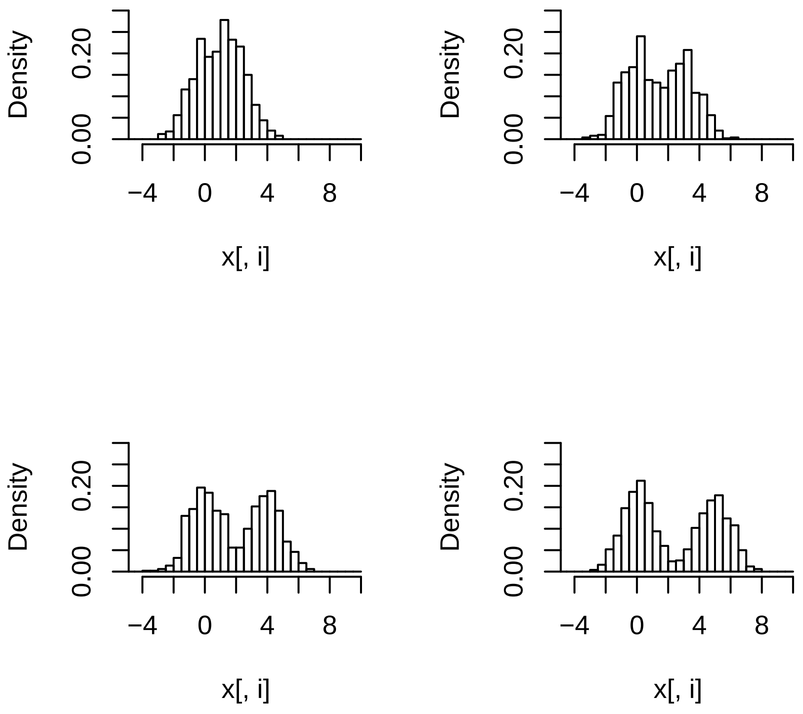
A multivariate normal mixture is denoted

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

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

6.2.1 Example 3.20 (Multivariate normal mixture)

```r
library(MASS)  #for mvrnorm
# ineffecient 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)
}

# 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
}

x <- loc.mix(1000, 0.5, 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, 0.3), freq = FALSE, main = "",
    breaks = seq(-5, 10, 0.5))
```



```
detach(package:MASS)
par(mfrow = c(1, 1))
```

It is difficult to visualize data in $\mathbb{R}^4$, so display only the histograms of the marginal distributions. All of the 1-D marginal distributions are univariate normal location mixtures. Methods for visualization of multivariate data are covered in Chapter 4.

6.3 Uniform Dist. on the d-Sphere

The d-sphere is the set of all points $x \in R^d$ such that $\| x \| = (x^T x)^{1/2} = 1$. If $X_1, ..., X_d$ are iid $N(0,1)$, then $U = (U_1, ..., U_d)$ is uniformly distributed on the unit sphere in $\mathbb{R}^d$, where

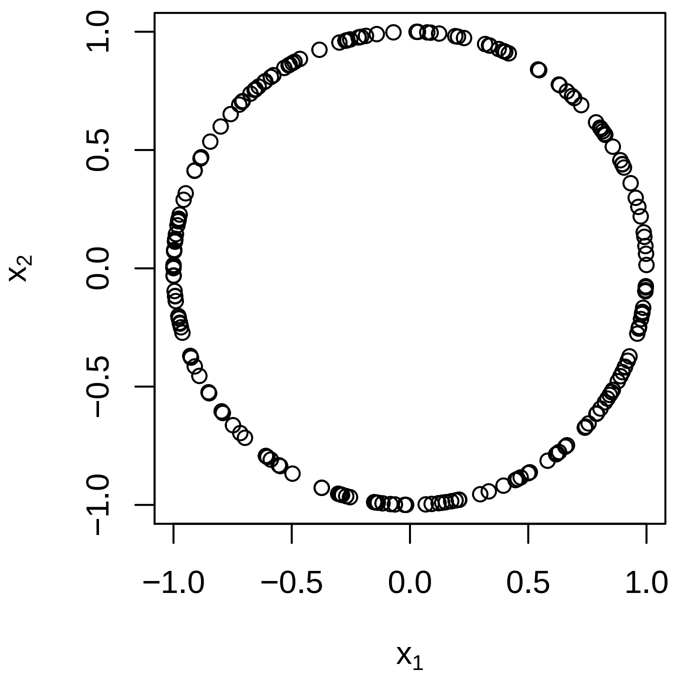
$$U_j = \frac{X_j}{(X_1^2 + \cdots + X_d^2)^{1/2}}, j = 1, \cdots, d. \tag{3.8}$$

To generate uniform r.v. on the d-Sphere, for each variate $u_i, i = 1, \cdots, n$, repeat

1. Generate a random sample $x_{i1}, \ldots, x_{id}$ from $N(0,1)$
2. Compute the Euclidean norm $\| x \| = (x_{i1}^2 + ¡¤¡¤¡¤ + x_{id}^2)^{1/2}$
3. Set $u_{ij} = x_{ij} / \| x \|, j = 1, \cdots, d.$
4. Deliver $u_i = (u_{i1}, \ldots, u_{id})$.

6.3.1 Example 3.21 (Generating variates on a sphere)

```r
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
}

# 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")
```