Solutions to Selected Exercises for Braun and Murdoch's A First Course in Statistical Programming with R, 2nd edition

Kristy Alexander, Yiwen Diao, Qiang Fu, and Yu Han
W. John Braun and Duncan J. Murdoch

August 9, 2016

Chapter 2

Introduction to the R Language

2.1 First steps

1.
```
170166719 %% 31079
```

```
## [1] 9194
```

3.
```
round(2000*(1.03^(1:30) - 1), 2)
```

```
##  [1]   60.00  121.80  185.45  251.02  318.55  388.10  459.75  533.54
##  [9]  609.55  687.83  768.47  851.52  937.07 1025.18 1115.93 1209.41
## [17] 1305.70 1404.87 1507.01 1612.22 1720.59 1832.21 1947.17 2065.59
## [25] 2187.56 2313.18 2442.58 2575.86 2713.13 2854.52
```

5.
```
r <- 7
area <- pi * r^2
area
```

```
## [1] 153.938
```

7.
```
48:(14*3)
```

```
## [1] 48 47 46 45 44 43 42
```

```
48:14*3
```

```
##  [1] 144 141 138 135 132 129 126 123 120 117 114 111 108 105 102  99  96
## [18]  93  90  87  84  81  78  75  72  69  66  63  60  57  54  51  48  45
## [35]  42
```

Yes, the parentheses are necessary, because the `:` has higher priority than the `*`, and in the second case the sequence is multiplied by 3.

2.3 Vectors in R

1. (a)
```
r <- 1.08
n <- c(10, 20, 30, 40)
sum1 <- c()
for(i in n){
  x <- 0:i
  sum1 <- c(sum1, sum(r^x))
}
sum1
```
```
## [1]  16.64549  50.42292 123.34587 280.78104
```

This gives the calculated sums for n = 10, 20, 30, 40.

```
sum2 <- (1 - r^(n + 1)) / (1 - r)
sum2
```
```
## [1]  16.64549  50.42292 123.34587 280.78104
```
```
sum2 - sum1
```
```
## [1] 0 0 0 0
```

The formula works.

3. (a)
```
n <- 100
sum(1:n)
```
```
## [1] 5050
```
```
(n * (n + 1)) / 2
```
```
## [1] 5050
```

(c)
```
n <- 400
sum(1:n)
```
```
## [1] 80200
```
```
(n * (n + 1)) / 2
```
```
## [1] 80200
```

7. (a)
```
N <- 500
sum(1/(1:N))
```
```
## [1] 6.792823
```
```
sum(1/(1:N)) - log(N) - 0.6
```
```
## [1] -0.02178467
```

(c)
```
N <- 2000
sum(1/(1:N))
```
```
## [1] 8.178368
```

```
sum(1/(1:N)) - log(N) - 0.6
```

```
## [1] -0.02253436
```

(e)
```
N <- 8000
sum(1/(1:N))
```

```
## [1] 9.564475
```

```
sum(1/(1:N)) - log(N) - 0.6
```

```
## [1] -0.02272184
```

8. (a)
```
rep( 0:4, rep(5,5))
```

```
##  [1] 0 0 0 0 0 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3 4 4 4 4 4
```

10.
```
x <- rep( c(rep(0, 3), rep(1, 4)), 5)
x
```

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

```
x <- as.factor(x)
levels(x) <- c("Male", "Female")
x
```

```
##  [1] Male   Male   Male   Female Female Female Female Male   Male   Male
## [11] Female Female Female Female Male   Male   Male   Female Female Female
## [21] Female Male   Male   Male   Female Female Female Female Male   Male
## [31] Male   Female Female Female Female
## Levels: Male Female
```

11.
```
more.colors <- c("red","yellow","blue","green","magenta","cyan")
more.colors[rep( seq(1,3), times = 4) + rep( seq(0,3), each=3)]
```

```
##  [1] "red"     "yellow"  "blue"    "yellow"  "blue"    "green"   "blue"
##  [8] "green"   "magenta" "green"   "magenta" "cyan"
```

2.4 Data storage in R

1. (a) To find binary expansions, use the method described in section 2.4.1.

```
x <- 6/7
2*x
```

```
## [1] 1.714286
```

```
x <- 2*x - 1
2*x
```

```
## [1] 1.428571
```

```
x <- 2*x - 1
2*x

## [1] 0.8571429

x <- 2*x
2*x

## [1] 1.714286
```

Thus the first 4 binary digits for 6/7 are 0.1101.

(b) Similarly, 1/7 in binary is 0.0010.

(c) Adding these gives 0.1111, which corresponds to

```
sum(2^(-(1:4)))

## [1] 0.9375
```

(d) The correct answer using exact arithmetic is 1; using only 4 binary digits gives us an error of

```
sum(2^(-(1:4))) - 1

## [1] -0.0625
```

3.
```
k <- 1:4
2^52 + k - 2^52

## [1] 1 2 3 4

2^53 + k - 2^53

## [1] 0 2 4 4

2^54 + k - 2^54

## [1] 0 0 4 4
```

We are right at the borderline of accuracy for double precision floating point. The first expression is done accurately, the other two suffer rounding error.

The result would be correct if done in a different order, e.g.

```
2^52  - 2^52 + k

## [1] 1 2 3 4

2^53  - 2^53 + k

## [1] 1 2 3 4

2^54  - 2^54 + k

## [1] 1 2 3 4
```

5. R guesses the century correctly.
6. R understands calendar dates.

2.6 Getting help

1. (a)
```
solar.radiation <- c(11.1,10.6,6.3,8.8,10.7,11.2,8.9,12.2)
```

   (b)
```
mean(solar.radiation)
## [1] 9.975
median(solar.radiation)
## [1] 10.65
range(solar.radiation)
## [1]  6.3 12.2
var(solar.radiation)
## [1] 3.525
```

   (c) i.
```
sr10 <- solar.radiation + 10
```

   ii.
```
mean(sr10)
## [1] 19.975
median(sr10)
## [1] 20.65
range(sr10)
## [1] 16.3 22.2
var(sr10)
## [1] 3.525
```

   iii. The mean, median and both ends of the range are increased by 10. The variance remains unchanged.

2.
```
n <- 1:15
sum(pmin(2^n, n^3))

## [1] 13396
```

2.7 Logical vectors and relational operators

1.
```
n <- 1:15
2^n > n^3

##  [1]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
## [12]  TRUE  TRUE  TRUE  TRUE
```

4. (a) `!(A & B) == (!A)|(!B)`

   The event that A and B are not both true is the same as the event that either A is not true or B is not true.

   The statement "There is no sun shower" is equivalent to the statement "Either the sky is not clear or it is not raining".

   `Truth Table to confirm that !(A & B) == (!A)|(!B)`

A	B	not (A and B)	(not A) or (not B)
True	True	False	False
True	False	True	True
False	True	True	True
False	False	True	True

   This implies that they are equivalent.

6. Use the `&&` operator:

```
testValue <- 7
(testValue > 0) && (sqrt(testValue) < 5)

## [1] TRUE

testValue <- -7
(testValue > 0) && (sqrt(testValue) < 5)

## [1] FALSE
```

9.
```
b * a

## [1] 13  0  0  2
```

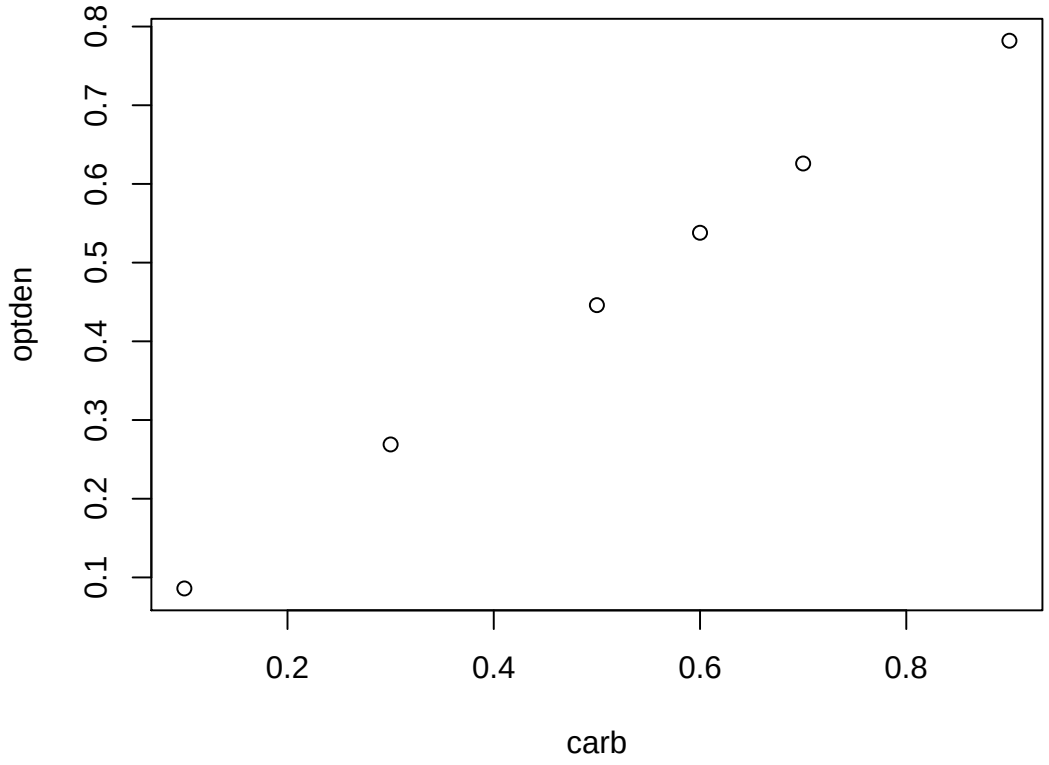
   This multiplies `b` with the (coerced) numeric values of `a`, element by element.

2.8 Data frames and lists

1. (a)
```
Formaldehyde[3,]

##   carb optden
## 3  0.5  0.446
```

   (b)
```
Formaldehyde[, "carb"]

## [1] 0.1 0.3 0.5 0.6 0.7 0.9
```

   (c)
```
plot(optden ~ carb, data = Formaldehyde)
```



It appears to be a linear relationship.

2.9 Data input and output

3.
```
numbers <- c(3,5,8,10,12)
dump( "numbers", file = "numbers.R")
rm(numbers)
ls()  # numbers should not appear in the resulting listing

##  [1] "a"               "A"               "ans"
##  [4] "area"            "b"               "B"
##  [7] "formula"         "FV"              "i"
## [10] "interest"        "intRate"         "k"
## [13] "more.colors"     "n"               "N"
## [16] "payment"         "principal"       "r"
## [19] "randomdata"      "solar.radiation" "sr10"
## [22] "srm2"            "sum1"            "sum2"
## [25] "sums"            "testValue"       "values"
## [28] "vp"              "vp1"             "x"

numbers

## Error in eval(expr, envir, enclos):  object 'numbers' not found
```

```
source("numbers.R")
ls()  # numbers should now appear in the resulting listing

##  [1] "a"               "A"               "ans"
##  [4] "area"            "b"               "B"
##  [7] "formula"         "FV"              "i"
## [10] "interest"        "intRate"         "k"
## [13] "more.colors"     "n"               "N"
## [16] "numbers"         "payment"         "principal"
## [19] "r"               "randomdata"      "solar.radiation"
## [22] "sr10"            "srm2"            "sum1"
## [25] "sum2"            "sums"            "testValue"
## [28] "values"          "vp"              "vp1"
## [31] "x"

numbers

## [1]  3  5  8 10 12
```

2.9.5 The `read.table` function

1. Creating `pretend.df`:

```
x <- c(61,175,111,124)
y <- c(13,21,24,23)
z <- c(4,18,14,18)
pretend.df <- cbind(x,y,z)
pretend.df <- data.frame(pretend.df)
pretend.df

##     x  y  z
## 1  61 13  4
## 2 175 21 18
## 3 111 24 14
## 4 124 23 18
```

Displaying the item:

```
pretend.df[ 1, 3]

## [1] 4
```

Chapter exercises

1.
```
11^2

## [1] 121

111^2
```

```
## [1] 12321

1111^2

## [1] 1234321

save <- options(digits = 18)
11111111^2

## [1] 123456787654321

111111111^2

## [1] 12345678987654320

options(save)
```

The final answer is wrong in the last digit due to rounding error.

3. (a)
```
chickwts300p <- subset(chickwts, weight > 300)
```

(b)
```
chickwtslinseed <- subset(chickwts, feed == "linseed")
```

(c)
```
mean(chickwtslinseed$weight)

## [1] 218.75
```

(d)
```
mean(subset(chickwts, feed != "linseed")$weight)

## [1] 269.9661
```

5.
```
rain.df <-read.table("http://www.statprogr.science/data/rnf6080.dat",
 header = FALSE, na.strings = "-999")
```

(a)
```
rain.df[ 2, 4]

## [1] 0
```

(b)
```
names(rain.df)

##  [1] "V1"  "V2"  "V3"  "V4"  "V5"  "V6"  "V7"  "V8"  "V9"  "V10" "V11"
## [12] "V12" "V13" "V14" "V15" "V16" "V17" "V18" "V19" "V20" "V21" "V22"
## [23] "V23" "V24" "V25" "V26" "V27"
```

(c)
```
rain.df[ 2, ]

##   V1 V2 V3 V4 V5 V6 V7 V8 V9 V10 V11 V12 V13 V14 V15 V16 V17 V18 V19 V20
## 2 60  4  2  0  0  0  0  0  0   0   0   0   0   0   0   0   0   0   0   0
##   V21 V22 V23 V24 V25 V26 V27
## 2   0   0   0   0   0   0   0
```

(e)
```
rain.df$daily <- rowSums(rain.df[ , 4:27], na.rm=TRUE)
```

7. (a)
```
dieRolls <- sample(1:6, 1000000, replace = TRUE)
```

(b)
```
dieRollsFactor <- factor(dieRolls)
levels(dieRollsFactor) <- c("One", "Two", "Three", "Four", "Five", "Six")
```

(c)
```
dieRollsChar <- as.character(dieRollsFactor)
```

(d)
```
table(dieRolls)
```

```
## dieRolls
##      1      2      3      4      5      6
## 166704 166063 167240 166433 167102 166458
```

```
table(dieRollsFactor)
```

```
## dieRollsFactor
##    One    Two  Three   Four   Five    Six
## 166704 166063 167240 166433 167102 166458
```

```
table(dieRollsChar)
```

```
## dieRollsChar
##   Five   Four    One    Six  Three    Two
## 167102 166433 166704 166458 167240 166063
```

(e)
```
system.time(table(dieRolls))
```

```
##    user  system elapsed
##    0.48    0.00    0.48
```

```
system.time(table(dieRollsFactor))
```

```
##    user  system elapsed
##    0.05    0.02    0.07
```

```
system.time(table(dieRollsChar))
```

```
##    user  system elapsed
##    0.06    0.01    0.08
```

9.
```
char <- c("2", "1", "0")
num <- 0:2
charnum <- data.frame(char, num)
as.numeric(char)
```

```
## [1] 2 1 0
```

```
as.numeric(charnum$char)
```

```
## [1] 2 1 0
```

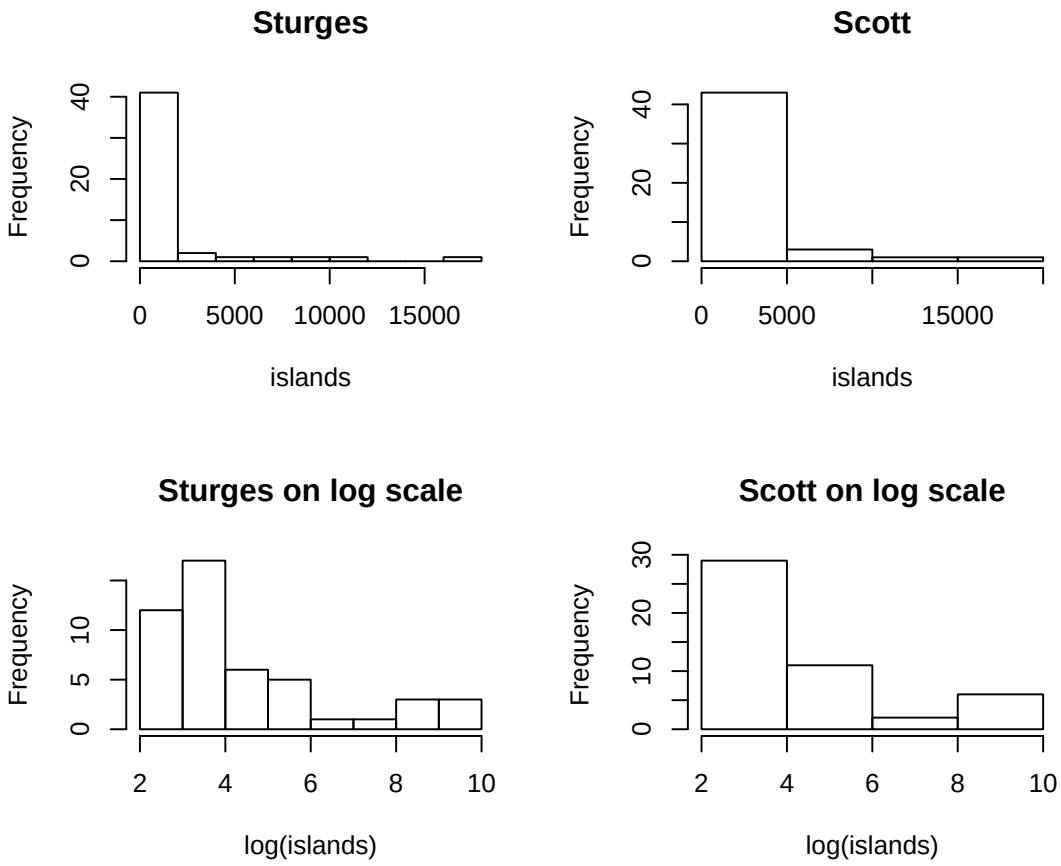
In the data frame, the `char` column is a factor, so we see the indices of the factor levels rather than conversions of the strings.

Chapter 3

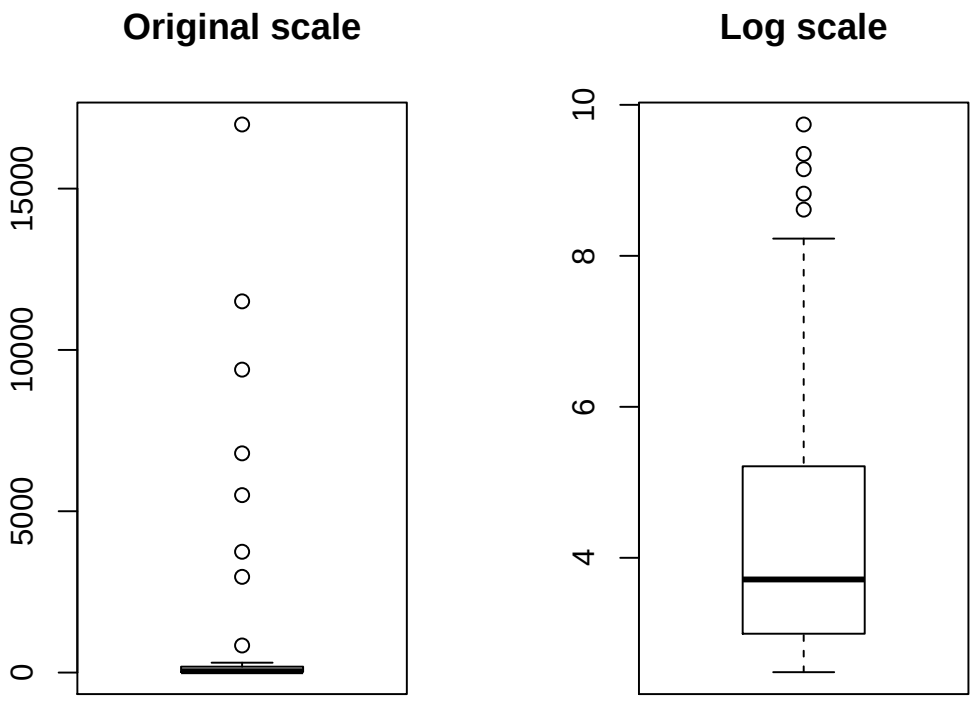
Programming statistical graphics

3.1 High level plots

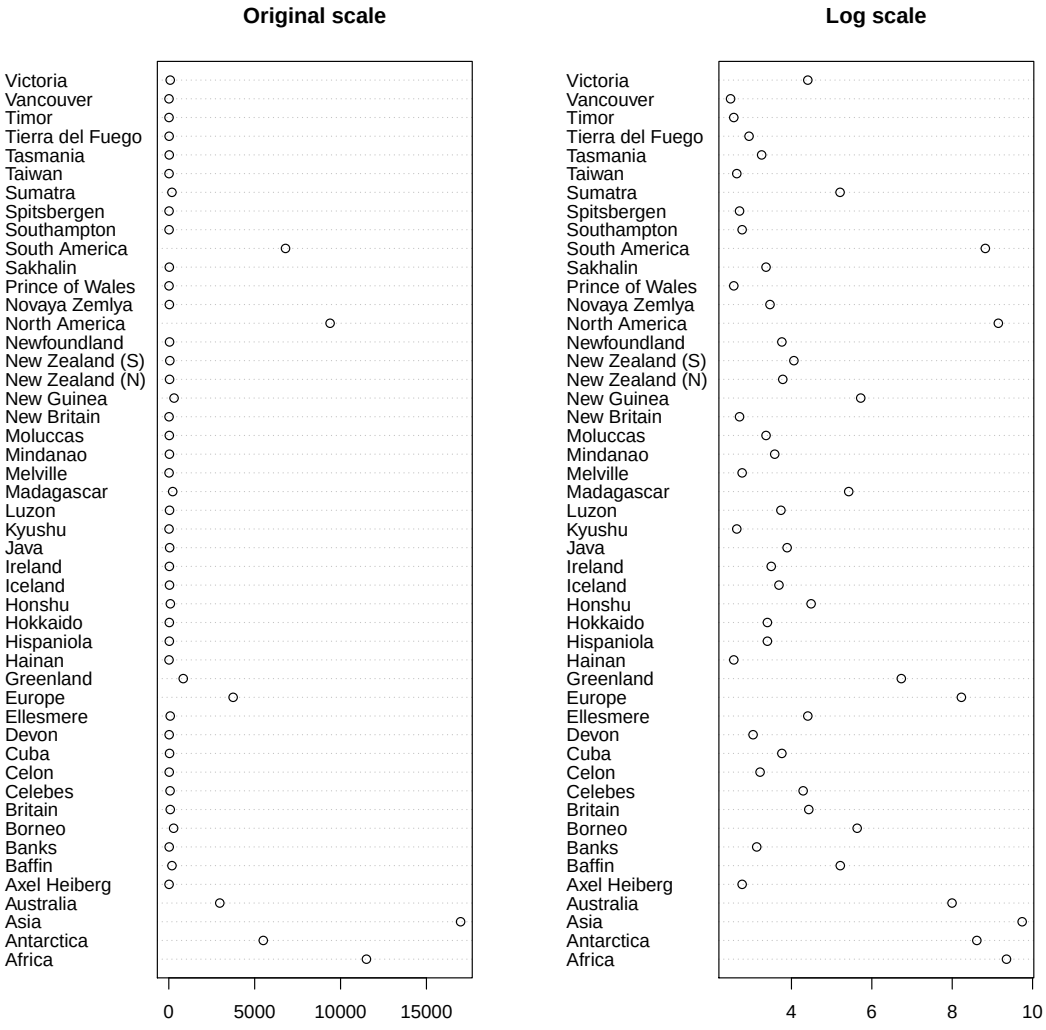
1. (a)
```
par(mfrow=c(2,2))
hist(islands, breaks = "Sturges", main = "Sturges")
hist(islands, breaks = "Scott", main = "Scott")
hist(log(islands), breaks = "Sturges", main = "Sturges on log scale")
hist(log(islands), breaks = "Scott", main = "Scott on log scale")
```



(c)
```
par(mfrow=c(1,2))
boxplot(islands, main = "Original scale")
boxplot(log(islands), main = "Log scale")
```



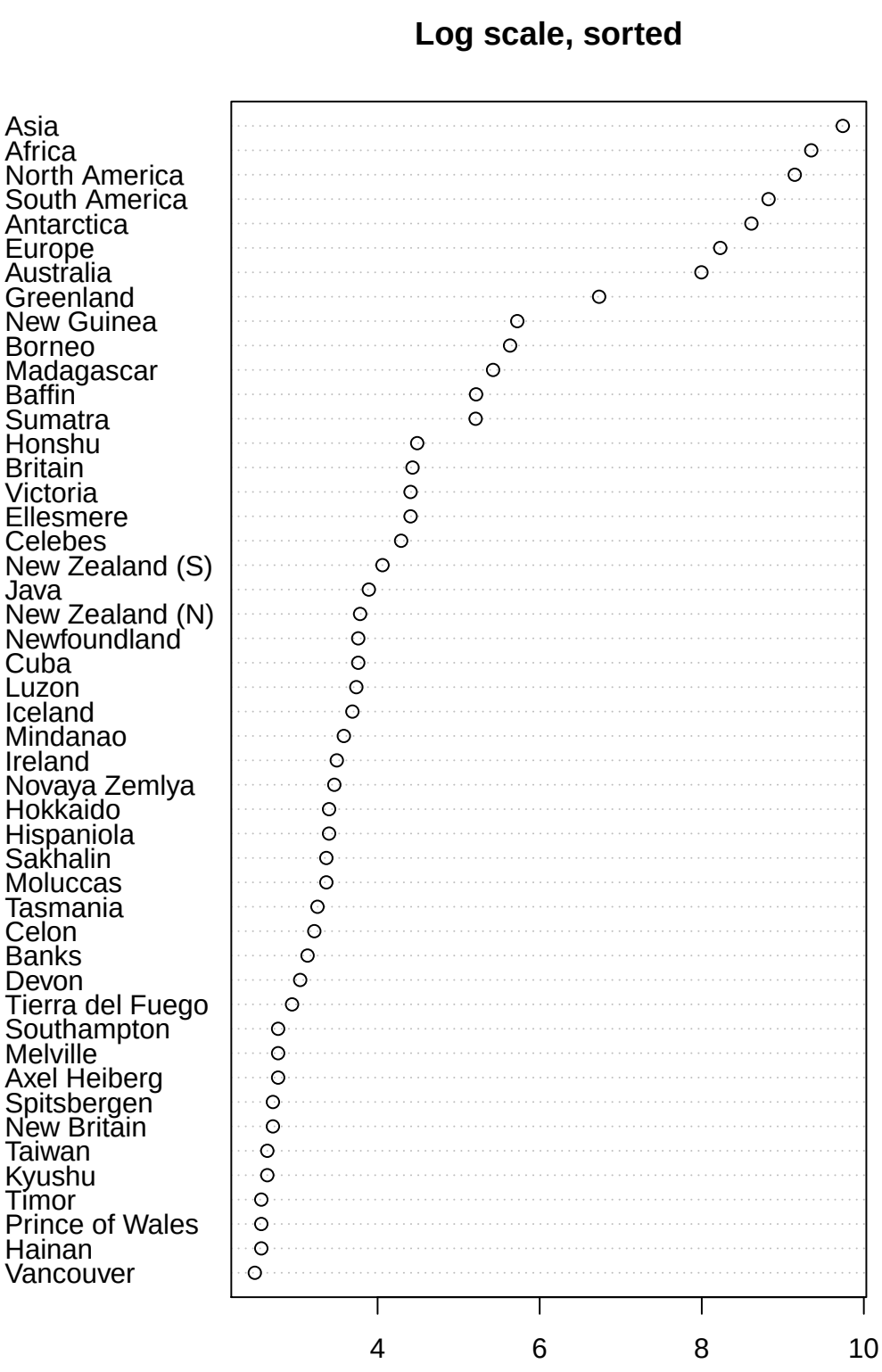
(d)
```
par(mfrow=c(1,2))
dotchart(islands,main="Original scale")
dotchart(log(islands),main="Log scale")
```



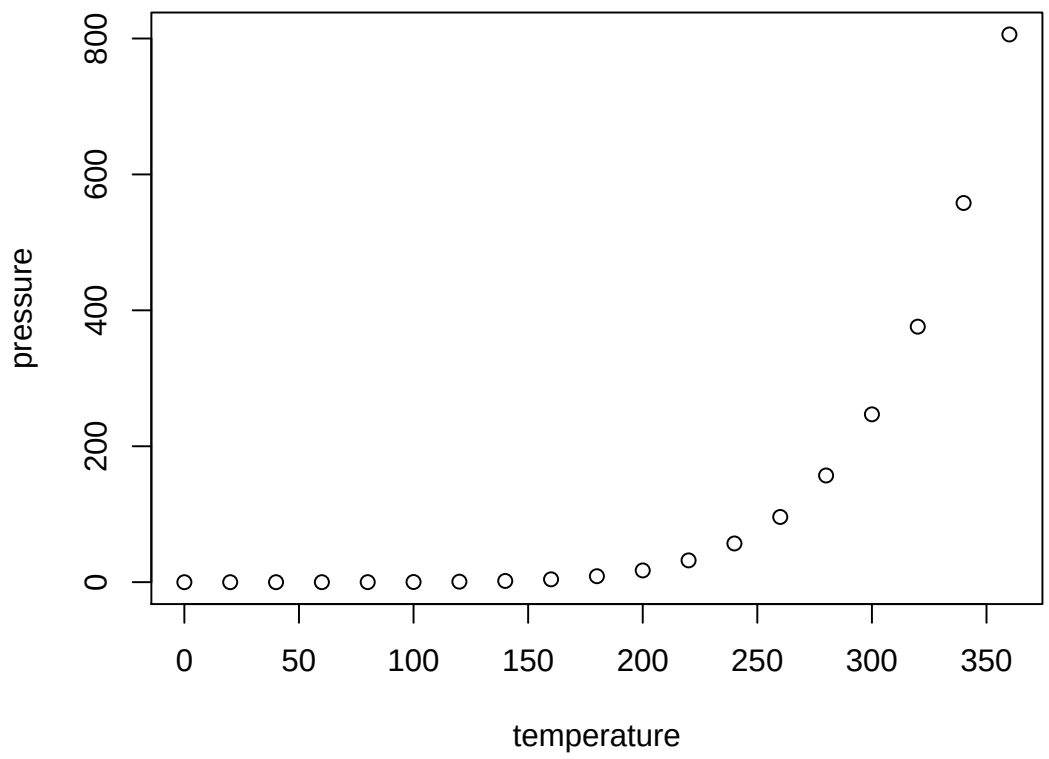
The log scale separates the islands better.

(e) This depends on the use; the ones we like best are the Sturges histogram on the log scale, and the dot chart on the log scale. The latter would be improved by sorting on size instead of alphabetically:

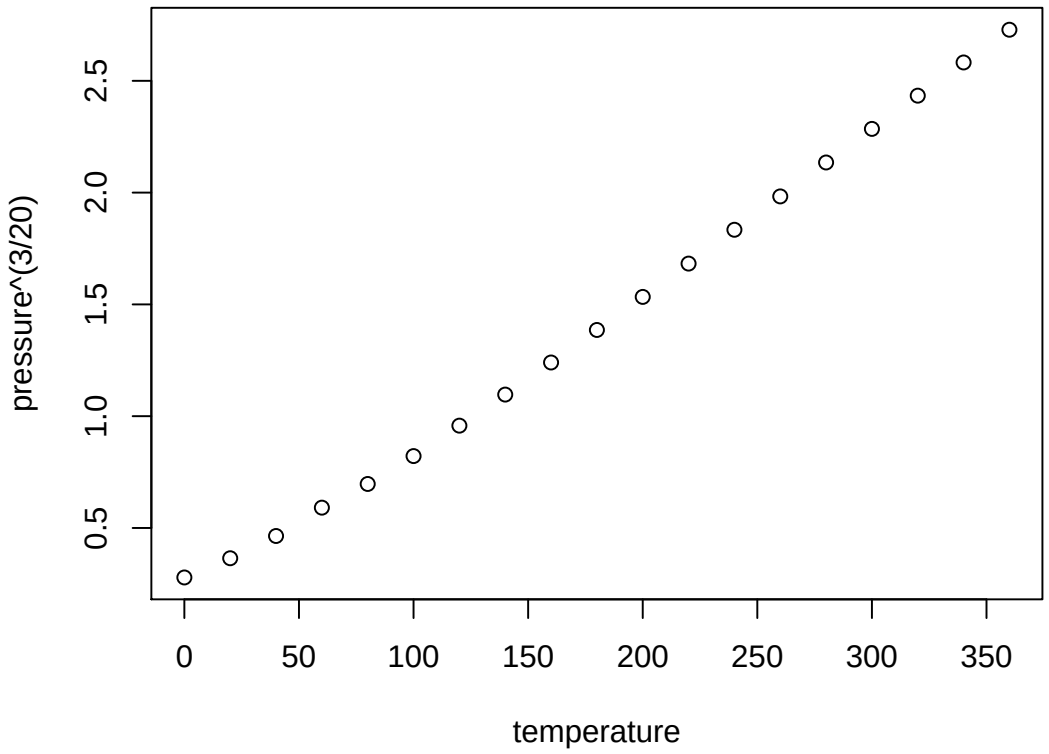
```
dotchart(sort(log(islands)), main = "Log scale, sorted")
```


Log scale, sorted
Asia
Africa
North America
South America
Antarctica
Europe
Australia
Greenland
New Guinea
Borneo
Madagascar
Baffin
Sumatra
Honshu
Britain
Victoria
Ellesmere
Celebes
New Zealand (S)
Java
New Zealand (N)
Newfoundland
Cuba
Luzon
Iceland
Mindanao
Ireland
Novaya Zemlya
Hokkaido
Hispaniola
Sakhalin
Moluccas
Tasmania
Celon
Banks
Devon
Tierra del Fuego
Southampton
Melville
Axel Heiberg
Spitsbergen
New Britain
Taiwan
Kyushu
Timor
Prince of Wales
Hainan
Vancouver
4
6
8
10

3. (a)
```
plot(pressure ~ temperature, data = pressure)
```

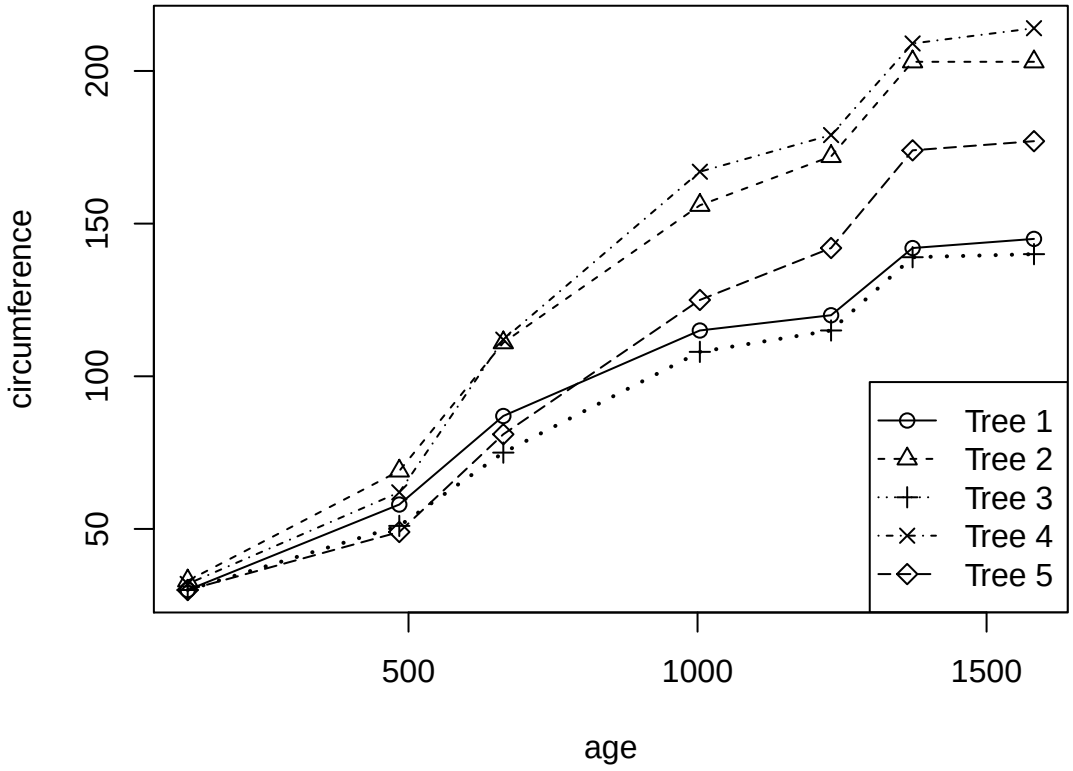


(c)
```
plot(pressure^(3/20) ~ temperature, data = pressure)
```

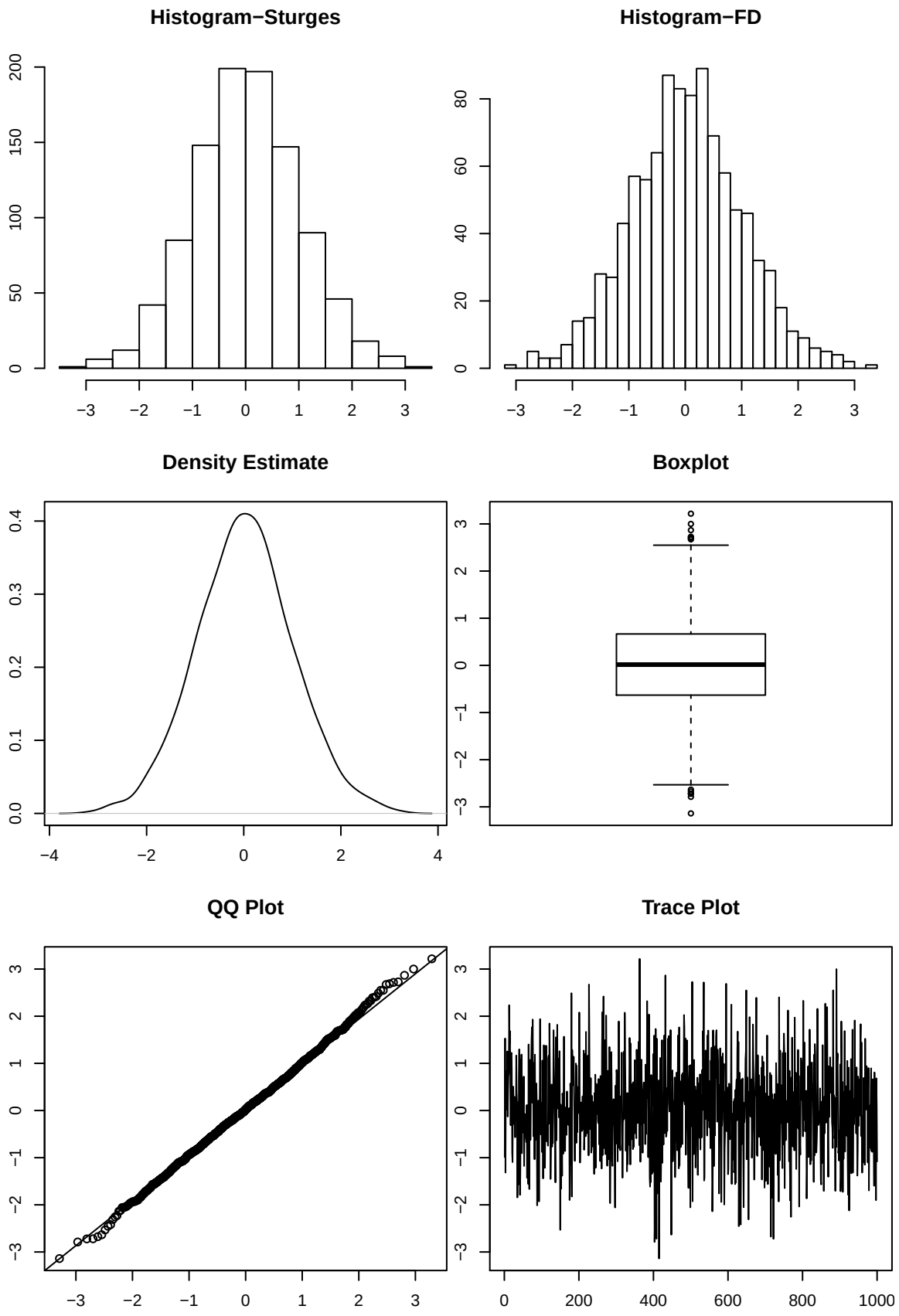


3.3 Low level graphics functions

1.
```
plot(circumference ~ age, pch = as.numeric(as.character(Tree)),
     data = Orange)
lines(circumference ~ age, data = Orange, subset = Tree == "1", lty = 1)
lines(circumference ~ age, data = Orange, subset = Tree == "2", lty = 2)
lines(circumference ~ age, data = Orange, subset = Tree == "3", lty = 3,
     lwd = 2)
lines(circumference ~ age, data = Orange, subset = Tree == "4", lty = 4)
lines(circumference ~ age, data = Orange, subset = Tree == "5", lty = 5)
legend("bottomright", legend = paste("Tree", 1:5), lty=1:5, pch = 1:5)
```

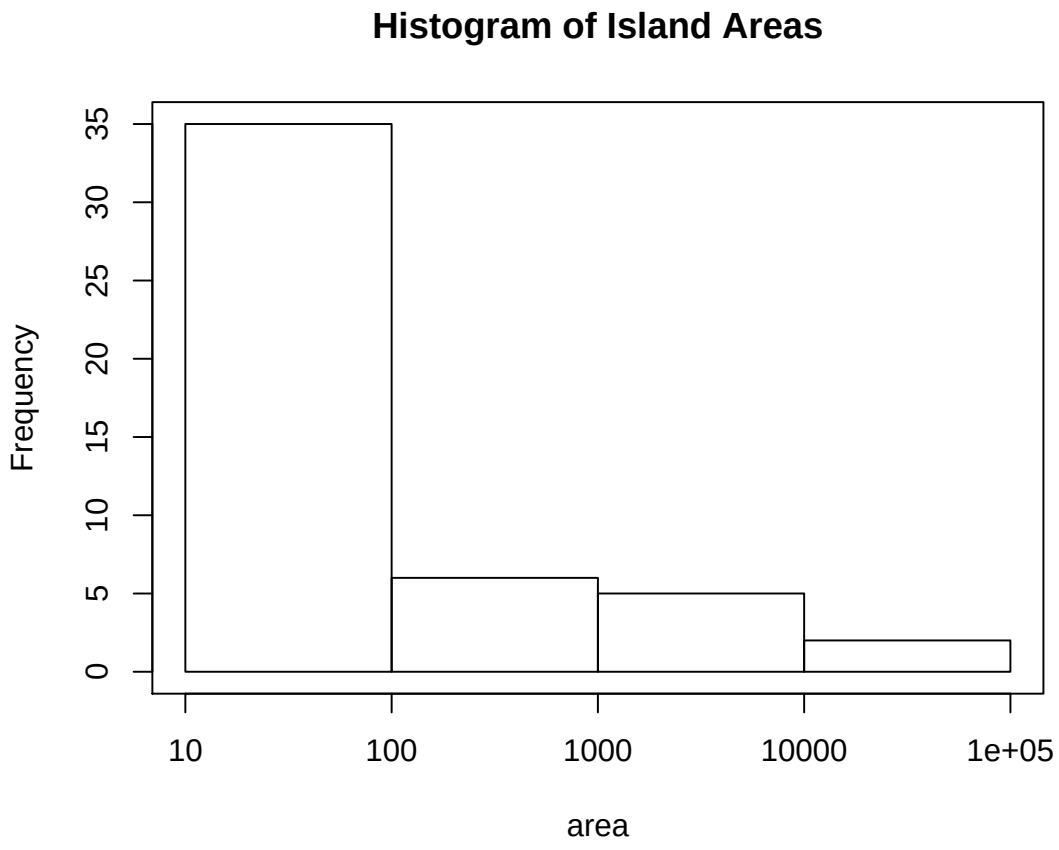


2.
```r
par(list = list(mfrow = c(3, 2), mar = c(2.1,2.1,4.1,0.1)))
Z <- rnorm(1000)
hist(Z, main = "Histogram-Sturges")
hist(Z, breaks = "Freedman-Diaconis", main = "Histogram-FD")
plot(density(Z), main = "Density Estimate")
boxplot(Z, main = "Boxplot")
qqnorm(Z, main = "QQ Plot"); qqline(Z)
ts.plot(Z, main = "Trace Plot")
```

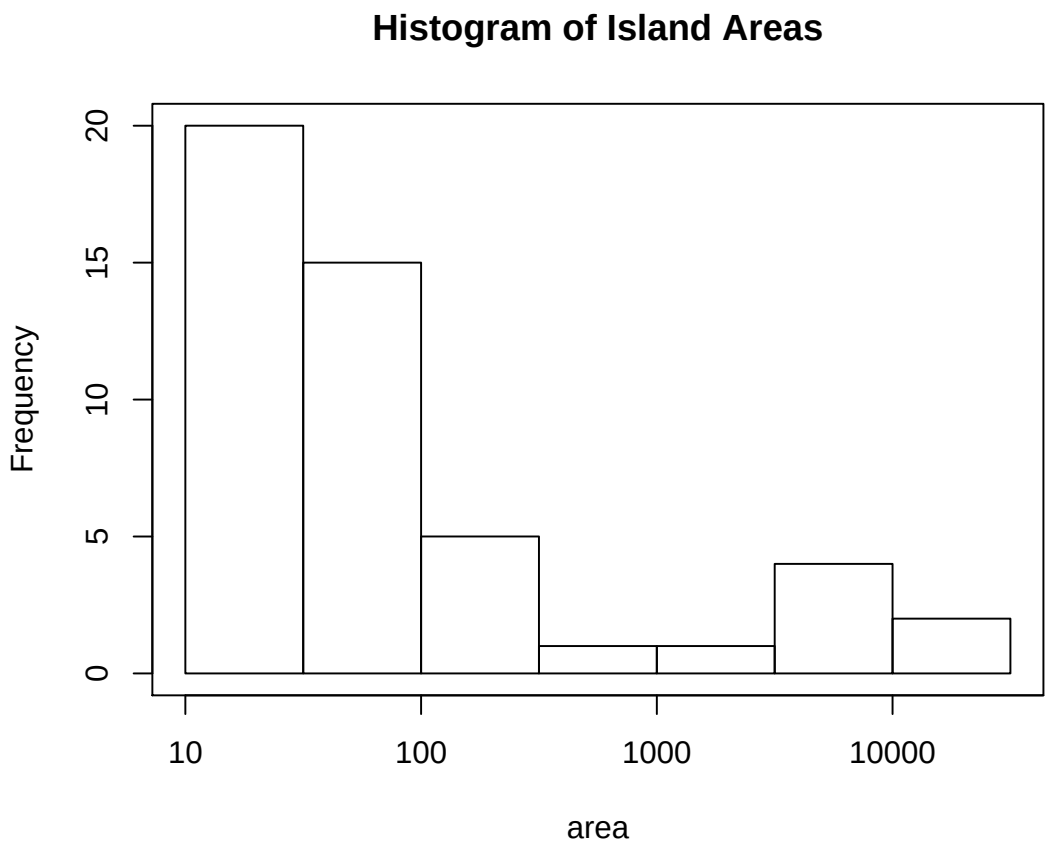

Histogram−Sturges
Histogram−FD
Density Estimate
Boxplot
QQ Plot
Trace Plot

Chapter exercises

1. (a)
```
hist(log(islands, 10), breaks = "Scott", axes = FALSE, xlab = "area",
                       main = "Histogram of Island Areas")
axis(1, at = 1:5, labels = 10^(1:5))
axis(2)
box()
```

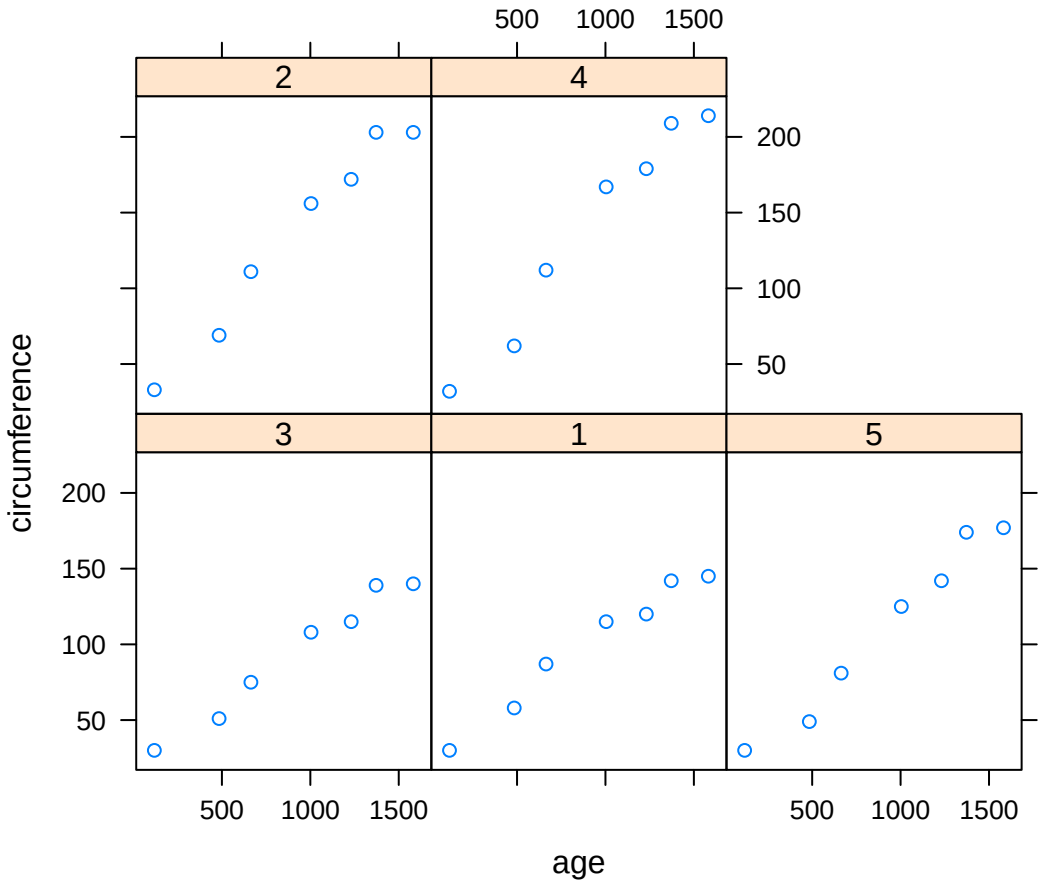


(c)
```
hist(log(islands, 10), breaks = "Sturges", axes = FALSE,
     xlab = "area", main = "Histogram of Island Areas")
axis(1, at=1:5, labels = 10^(1:5))
axis(2)
box()
```



The note about `round()` in the question is incorrect.

3.
```
library(lattice)
xyplot(circumference ~ age | Tree, data = Orange)
```


2
4
3
1
5
circumference
age
50
100
150
200
500
1000
1500

Chapter 4

Programming with R

4.1 Flow control

4.1.1 The `for()` loop

1. (a)
```
Fibonacci <- numeric(12)
Fibonacci[1] <- 2
Fibonacci[2] <- 2
for (i in 3:12) Fibonacci[i] <- Fibonacci[i-2] + Fibonacci[i-1]
Fibonacci
```
```
##  [1]   2   2   4   6  10  16  26  42  68 110 178 288
```

   (c)
```
Fibonacci <- numeric(12)
Fibonacci[1] <- Fibonacci[2] <- 1
for (i in 3:12) Fibonacci[i] <- Fibonacci[i-1] - Fibonacci[i-2]
Fibonacci
```
```
##  [1]  1  1  0 -1 -1  0  1  1  0 -1 -1  0
```

3. (a)
```
answer <- 0
for (j in 1:5) answer <- answer + j
answer
```
```
## [1] 15
```

   (b)
```
answer <- NULL
for (j in 1:5) answer <- answer + j
answer
```
```
## numeric(0)
```

   (c)
```
answer <- 0
for (j in 1:5) answer <- c(answer, j)
answer
```
```
## [1] 0 1 2 3 4 5
```

(d)
```
answer <- 1
for (j in 1:5) answer <- answer * j
answer
```

```
## [1] 120
```

(e)
```
answer <- 3
for (j in 1:15) answer <- c(answer, (7 * answer[j]) %% 31)
answer
```

```
##  [1]  3 21 23  6 11 15 12 22 30 24 13 29 17 26 27  3
```

5.
```
x <- 0.5
count <- 0
while(abs(x - cos(x)) > 0.01){
    x <- cos(x)
    count <- count + 1
}
count
```

```
## [1] 10
```

```
x <- 0.5
count <- 0
while(abs(x - cos(x)) > 0.001){
    x <- cos(x)
    count <- count + 1
}
count
```

```
## [1] 15
```

```
x <- 0.5
count <- 0
while(abs(x - cos(x)) > 0.0001){
    x <- cos(x)
    count <- count + 1
}
count
```

```
## [1] 21
```

```
x <- 0.7
count <- 0
while(abs(x - cos(x))>0.0001){
    x <- cos(x)
    count <- count + 1
}
count
```

```
## [1] 17
```

```
x <- 0.0
count <- 0
while(abs(x - cos(x)) > 0.0001){
    x <- cos(x)
    count <- count + 1
}
count

## [1] 23
```

4.1.2 The `if()` statement

1. Yes, the function will work properly when `n` is not an integer.

4. ```
   GIC <- function(P, n){
        if (n<=3) i <- 0.04 else i <- 0.05
        return(P*((1+i)^n -1))
    }
   ```

4.1.3 The `while()` loop

1. ```
   Fib1 <- 1
   Fib2 <- 1
   Fibonacci <- c(Fib1, Fib2)
   while (Fib1 < 300){
       Fibonacci <- c(Fibonacci, Fib2)
       Fib2 <- Fib1 + Fib2
       Fib1 <- max(Fibonacci)
   }
   print(Fibonacci)

   ##  [1]   1   1   1   2   3   5   8  13  21  34  55  89 144 233 377
   ```

3. ```
   Fibonacci <- c(1, 1, 1)
   while (max(Fibonacci) < 1000000){
        Fibonacci <- c(Fibonacci, Fibonacci[length(Fibonacci)]+
                       Fibonacci[length(Fibonacci)-1])
   }
   sum(Fibonacci < 1000000)

   ## [1] 31
   ```

4.1.4 Newton's method for root finding

1. ```
   x <- 0
   f <- x^7 + 10000*x^6 + 1.06 * x^5 + 10600*x^4 + 0.0605 * x^3 +
        605 * x^2 + 0.0005 *x + 5
   tolerance <- 0.000001
   ```

```
count <- 0
while (abs(f) > tolerance){
    f.prime <- 7*x^6 + 6*10000*x^5 + 5*1.06*x^4 + 4*10600*x^3 + 3*0.0605*x^2 +
               2*605*x + 0.0005
    x <- x - f/f.prime
    f <- x^7 + 10000*x^6 + 1.06*x^5 + 10600*x^4 + 0.0605*x^3 +
         605*x^2 + 0.0005*x + 5
    count <- count + 1
}
count

## [1] 1
```

3.
```
x <- -1.5
f <- cos(x) + exp(x)
tolerance <- 0.000001
while (abs(f) > tolerance){
    f.prime <- -sin(x) - 1/x
    x <- x - f/f.prime
    f <- cos(x) + exp(x)
}
x

## [1] -1.746139
```

5. The function has only one zero which is $x = 0.6$. For initial guess 0.5, 0.75 and 0.2, Newton's method gives a solution that approaches $x = 1$. For initial guess 1.25, the method does not converge.

7.
```
i <- 0.006
count <- 0
f <- (1- (1 + i)^(-20))/19 - i
tolerance <- 0.000001
while (abs(f) > tolerance){
    f.prime <- (20/19)*((1 + i)^(-21))-1
    i <- i - f/f.prime
    f <- (1 - (1 + i)^(-20))/19 - i
    count <- count+1
}
i

## [1] 0.004939979

count

## [1] 2
```

4.1.5 The `repeat` loop, and the `break` and `next` statements

1. (a)
```
x1 <- 0
x2 <- 2
repeat{
     f1 <- x1^3 + 2*x1^2 - 7
     f2 <- x2^3 + 2*x2^2 - 7
     x3 <- (x1 + x2)/2
     f3 <- x3^3 + 2*x3^2 - 7
     if(f3==0){
           print(x3)
           break
     } else {
           if (f3*f1 > 0) {
                x1 <- x3
           } else {
                x2 <- x3
           }
     }
     if(abs(x1 - x2) < 0.000001){
          print((x1 + x2)/2)
     break
     }
 }
```

```
## [1] 1.428817
```

4.2 Managing complexity through functions

4.2.1 What are functions?

1. Call `typeof` or `str` or just type in the function names on each:

```
typeof(var)
```

```
## [1] "closure"
```

```
str(cos)
```

```
## function (x)
```

```
median
```

```
## function (x, na.rm = FALSE)
## UseMethod("median")
## <bytecode: 0x03b92b40>
## <environment: namespace:stats>
```

```
typeof(read.table)
```

```
## [1] "closure"
```

```
str(dump)
```

```
## function (list, file = "dumpdata.R", append = FALSE, control = "all",
##     envir = parent.frame(), evaluate = TRUE)
```

Note that functions are usually listed as type “closure”.

3.
```
bisection <- function(f, x1, x2) {
    repeat{
        f1 <- f(x1)
        f2 <- f(x2)
        x3 <- (x1 + x2)/2
        f3 <- f(x3)
        if(f3 == 0) {
           return(x3)
        } else {
           if (f3*f1 > 0) {
              x1 <- x3
           } else {
              x2 <- x3
           }
        }
        if (abs(x1 - x2) < 0.000001) {
          return((x1 + x2)/2)
        }
    }
}

bisection(function(x) cos(x) - 0.5, 0, pi)

## [1] 1.047198
```

4.4 Miscellaneous Programming tips

1.
```
factorial(10)

## [1] 3628800

factorial(50)

## [1] 3.041409e+64

factorial(100)

## [1] 9.332622e+157

factorial(1000)

## Warning in factorial(1000):  value out of range in 'gammafn'

## [1] Inf
```

3.
```
compound.interest <- function(P, j, m, n){
    i.r <- j/m
    P*(1+i.r)^(n*m)
}
```

5. (a)
```
mortgage.payment <- function(P, i.r, n)
     P*i.r/(1 - (1 + i.r)^(-n))
```

4.5 Some general programming guidelines

4.5.1 Top-down design

1.
```
mergesort <- function (x, decreasing = FALSE) {
    len <- length(x)
    if (len < 2) result <- x
    else {
        y <- x[1:(len %/% 2)]
        z <- x[(len %/% 2 + 1):len]
        y <- mergesort(y, decreasing)
        z <- mergesort(z, decreasing)
        result <- c()
        while (min(length(y), length(z)) > 0) {
        if ((!decreasing && y[1] < z[1]) ||
            (decreasing && y[1] > z[1])) {
                result <- c(result, y[1])
                y <- y[-1]
            } else {
                result <- c(result, z[1])
                z <- z[-1]
            }
        }
        if (length(y) > 0)
            result <- c(result, y)
        else
            result <- c(result, z)
    }
    return(result)
}
x <- c(1:5, 10:6)
mergesort(x)

##  [1]  1  2  3  4  5  6  7  8  9 10

mergesort(x, decreasing = TRUE)

##  [1] 10  9  8  7  6  5  4  3  2  1
```

4.6 Debugging and maintenance

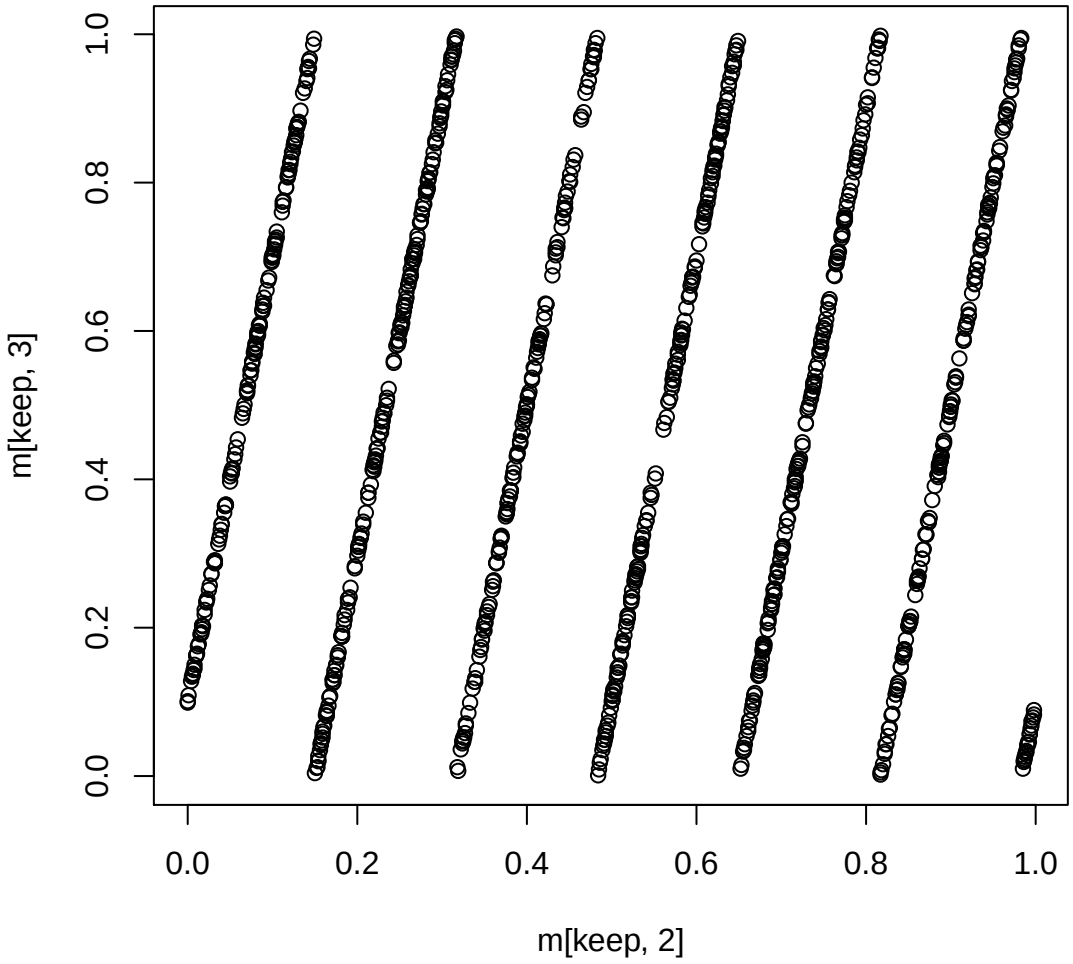
1. We'll modify the function to print values as it runs:

```
mergesort <- function (x) {
    cat("x = "); print(x)
    len <- length(x)
    if (len < 2) result <- x
    else {
        y <- x[1:(len %/% 2)]
        z <- x[(len %/% 2 + 1):len]
        cat("y = "); print(y)
        cat("z = "); print(z)
        y <- mergesort(y)
        z <- mergesort(z)
        result <- c()
        while (min(length(y), length(z)) > 0) {
            if (y[1] < z[1]) {
                result <- c(result, y[1])
                y <- y[-1]
            } else {
                result <- c(result, z[1])
                z <- z[-1]
            }
        }
        if (length(y) > 0)
            result <- c(result, y)
        else
            result <- c(result, z)
    }
    return(result)
}
mergesort(c(3, 5, 4, 2))

## x = [1] 3 5 4 2
## y = [1] 3 5
## z = [1] 4 2
## x = [1] 3 5
## y = [1] 3
## z = [1] 5
## x = [1] 3
## x = [1] 5
## x = [1] 4 2
## y = [1] 4
## z = [1] 2
## x = [1] 4
## x = [1] 2
## [1] 2 3 4 5
```

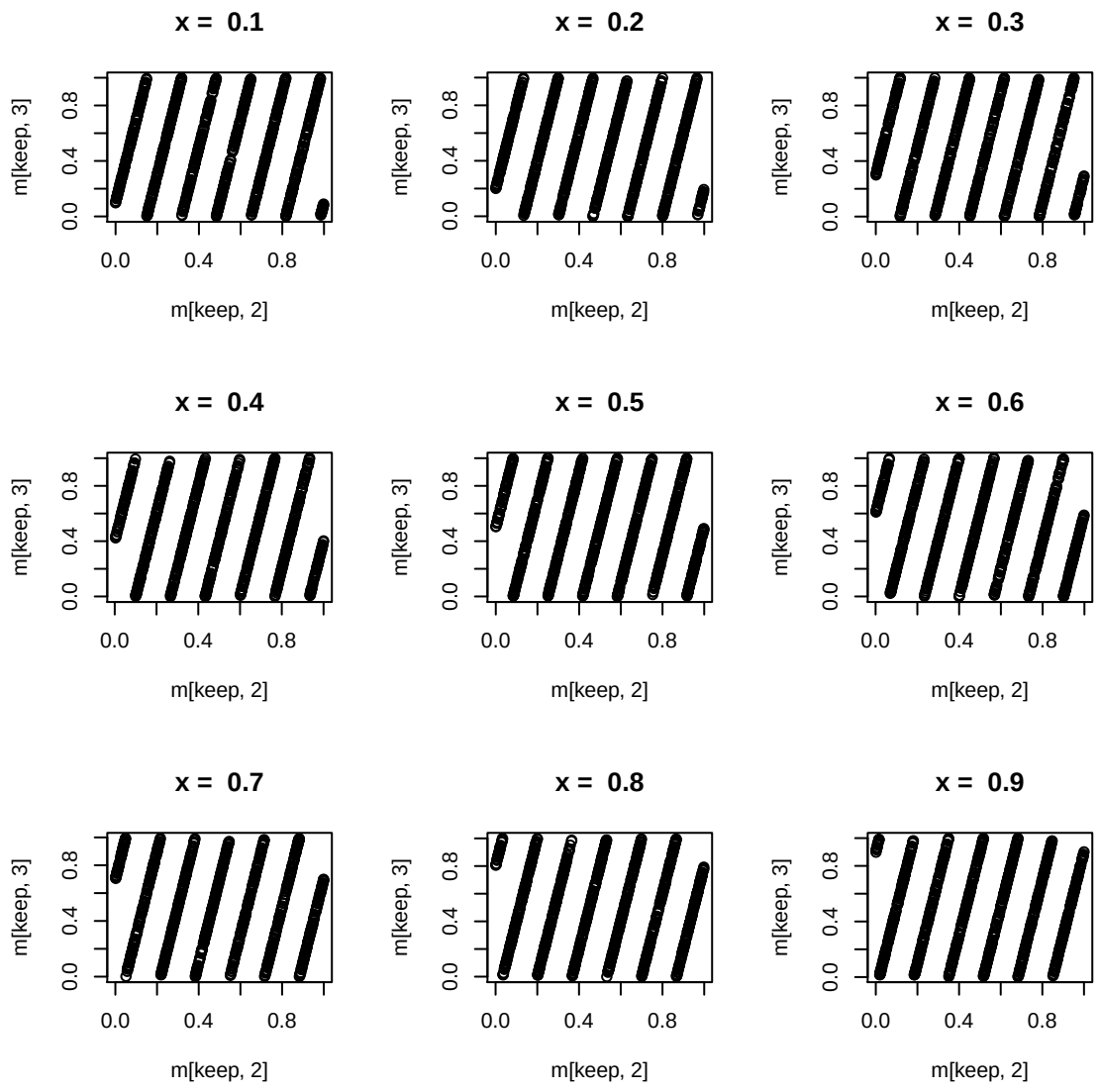
4.8 Chapter Exercises

1.
```
# RANDU exercise from programming.Rnw

n <- 3000000
results <- numeric(n)
```

```
x <- 123
for (i in 1:n) {
        x <- (65539*x) %% (2^31)
        results[i] <- x / (2^31)
}
m <- matrix(round(results, 3), ncol = 3, byrow = TRUE)
keep <- m[,1] == 0.1
plot(m[keep, 2], m[keep, 3])
```




```
par(mfrow = c(3, 3))
for (i in 1:9) {
  keep <- m[,1] == round(i/10, 3)
  plot(m[keep, 2], m[keep, 3], main=paste("x = ", i/10))
}
```



2.
```r
directpoly <- function(x, coef){
    n <- length(coef)
    result <- 0
    for(i in seq_len(n)){
        result <- result + coef[i]*x^(n-i)
    }
    result
}
```

3.
```r
hornerpoly <- function(x, coef){
    n <- length(coef)
    a <- rep(coef[1], length(x))
    for(i in rev(seq_len(n-1))) {
        a <- a*x + coef[n - i + 1]
    }
```

```
    a
}
```

6. (a)
```
microbenchmark({
x1 <- 2.1
x2 <- 3.1
repeat{
     f1 <- (x1-3)*exp(-x1)
     f2 <- (x2-3)*exp(-x2)
     x3 <- (x1+x2)/2
     f3 <- (x3-3)*exp(-x3)
     if(f3==0){
           break
     }else{
         if(f3*f1>0){
              x1 <- x3
         }else{
              x2 <- x3
         }
     }
     if(abs(x1-x2)<0.0000001){
          break
     }
 }
})
```

```
## Unit: microseconds
##
##  {     x1 <- 2.1     x2 <- 3.1     repeat {         f1 <- (x1 - 3) * exp(-x1)         f2 <-
##      min     lq     mean   median      uq    max neval
##  142.457 143.61 146.2549 144.5695 149.369 160.12   100
```

(b)
```
microbenchmark({
x1 <- 2.1
x2 <- 3.1
repeat{
    f1 <- (x1^2 -6*x1 +9)*exp(-x1)
    f2 <- (x2^2 -6*x2 +9)*exp(-x2)
    x3 <- (x1+x2)/2
    f3 <- (x3^2 -6*x3 +9)*exp(-x3)
    if(f3==0){
          break
    }else{
        if(f3*f1>0){
             x1 <- x3
        }else{
             x2 <- x3
        }
    }
    if(abs(x1-x2)<0.0000001){
         break
    }
```

```
}
})
```

```
## Unit: microseconds
##
##  {     x1 <- 2.1     x2 <- 3.1     repeat {         f1 <- (x1^2 - 6 * x1 + 9) * exp(-x1)
##      min      lq     mean  median      uq     max neval
##  188.151 195.063 196.8519 196.983 198.902 214.262   100
```

Chapter 5

Simulation

5.2 Generation of pseudorandom numbers

1.
```
x0 <- 17218
x <- numeric(20)
x[1] <- (172 * x0) %% 30307
for(i in 2:20){
 x[i] <- (x[i-1] * 172) %% 30307
 }
x
```

```
##  [1] 21717  7563 27942 17518 12703  2812 29059 27800 23401 24448 22690
## [12] 23384 21524  4674 15946 15082 18009  6234 11503  8561
```

3. (a)
```
set.seed(32078)
runif(10, 0, 1)
```

```
##  [1] 0.2564626 0.4988177 0.5266549 0.6269816 0.8052754 0.1843452 0.5102327
##  [8] 0.3683905 0.1708176 0.7432888
```

(b)
```
set.seed(32078)
runif(10, 3, 7)
```

```
##  [1] 4.025850 4.995271 5.106620 5.507927 6.221102 3.737381 5.040931
##  [8] 4.473562 3.683270 5.973155
```

(c)
```
set.seed(32078)
runif(10, -2, 2)
```

```
##  [1] -0.974149697 -0.004729333  0.106619657  0.507926506  1.221101642
##  [6] -1.262619189  0.040930690 -0.526437979 -1.316729628  0.973155177
```

5. (a)
```
r <- runif(10000, 3.7, 5.8)
mean(r)
```

```
## [1] 4.747523
```

```
var(r)
```

```
## [1] 0.3623486
```

```
sd(r)
```

```
## [1] 0.601954
```

```
(3.7 + 5.8)/2
```

```
## [1] 4.75
```

```
(5.8 - 3.7)^2/12
```

```
## [1] 0.3675
```

```
sqrt((5.8 - 3.7)^2/12)
```

```
## [1] 0.6062178
```

(b)
```
mean(r > 4)
```

```
## [1] 0.86
```

```
(5.8 - 4)/(5.8 - 3.7)
```

```
## [1] 0.8571429
```

7.
```
U1 <- runif(10000)
U2 <- runif(10000)
U3 <- runif(10000)
U <- U1 + U2 + U3
```

(a)
```
mean(U)
```

```
## [1] 1.500841
```

(b)
```
var(U)
```

```
## [1] 0.2418042
```

```
var(U1) + var(U2) + var(U3)
```

```
## [1] 0.247073
```

(c)
```
mean(sqrt(U))
```

```
## [1] 1.206707
```

(d)
```
V <- sqrt(U1) + sqrt(U2) + sqrt(U3)
mean(V >= 0.8)
```

```
## [1] 0.9978
```

9. (a)
```
sample(1:100, size = 50, replace = FALSE)
```

```
##  [1]  53  14  46  23  61  75  49  83  44 100  73  40  26  65  79  82  97
## [18]   3  10  25  76  84   2  32  47  71   4  52  78   6  59  91   7  93
## [35]  58  31  38  42  21  72  99   8  98   1  36  66  22  74  30  24
```

(b)
```
sample(1:100, size = 50, replace = TRUE)
```

```
##  [1] 70 15  6 92 29 75 70 14 82 22 40 62 98 92 77 98 64 55 21 75 47  8 17
## [24] 76 93 52 81 52 75 76 93 19 31 98 63 52  8 54 71 24 28 84 13 48 51  3
## [47]  5  6 62 34
```

5.3 Simulation of other random variables

5.3.1 Bernoulli random variables

1. (a)
```
guess <- function(n) {
  rbinom(n, size = 1, prob = 0.5)
}
sum(guess(10))
```

```
## [1] 3
```

(b)
```
sum(guess(1000))
```

```
## [1] 486
```

3.
```
r <- rbinom(500, size = 1, prob = 0.99)
mean(r)
```

```
## [1] 0.992
```

```
var(r)
```

```
## [1] 0.007951904
```

```
0.99
```

```
## [1] 0.99
```

```
0.99*0.01
```

```
## [1] 0.0099
```

5.3.2 Binomial random variables

1.
```
x <- rbinom(24, prob = 0.15, size = 25); x
```

```
##  [1] 4 6 5 8 4 6 3 4 3 5 6 2 5 5 1 3 4 3 8 2 6 5 1 2
```

```
sum(x > 5)
```

```
## [1] 6

x <- rbinom(24, prob = 0.2, size = 25); x

##  [1] 3 2 5 7 4 7 3 4 7 2 5 6 6 3 6 4 1 5 7 1 5 2 5 1

sum(x > 5)

## [1] 7

x <- rbinom(24, prob = 0.25, size = 25); x

##  [1]  3  8  9  7  4  7  3 10  7  2  6  6  6  5  8  6  7  7  6  8 10  8  5
## [24]  9

sum(x > 5)

## [1] 18
```

3.
```
r <- rbinom(1000, size = 18, prob = 0.76)
mean(r)

## [1] 13.68

var(r)

## [1] 3.280881

18*0.76

## [1] 13.68

18*0.76*0.24

## [1] 3.2832
```

5. (a)
```
#Generate binomial pseudorandom variables by summing Bernoulli
ranbin2 <- function(n, size, prob){
    #'singlenumber' generates one binomial random variable.
    singlenumber <- function(){
        x <- runif(size)
        sum(x < prob)
    }
    replicate(n, singlenumber())
}
```

(b)
```
system.time(gcFirst = TRUE, ranbin2(n = 10000, size = 10, prob = 0.4))
```

```
##    user  system elapsed
##    0.05    0.00    0.05
```

```
system.time(gcFirst = TRUE, rbinom(n = 10000, size = 10, prob = 0.4))
```

```
##    user  system elapsed
##       0       0       0
```

```
system.time(gcFirst = TRUE, ranbin2(n = 10000, size = 100, prob = 0.4))
```

```
##    user  system elapsed
##     0.1     0.0     0.1
```

```
system.time(gcFirst = TRUE, rbinom(n = 10000, size = 100, prob = 0.4))
```

```
##    user  system elapsed
##       0       0       0
```

```
system.time(gcFirst = TRUE, ranbin2(n = 10000, size = 1000, prob = 0.4))
```

```
##    user  system elapsed
##    0.44    0.00    0.44
```

```
system.time(gcFirst = TRUE, rbinom(n = 10000, size = 1000, prob = 0.4))
```

```
##    user  system elapsed
##       0       0       0
```

5.3.3 Poisson random variables

1.
```
rpois(15, lambda = 2.8)
```

```
##  [1] 4 3 5 1 3 4 4 3 2 2 2 3 5 1 4
```

3.
```
x <- rpois(10000, lambda = 7.2)
mean(x)
```

```
## [1] 7.1707
```

```
var(x)
```

```
## [1] 6.990461
```

```
7.2
```

```
## [1] 7.2
```

```
7.2
```

```
## [1] 7.2
```

7.
```
N <- rpois(10000, lambda=2.5 * 2)
pts <- list()
for (i in 1:10000)
    pts[[i]] <- runif(N[i], 0, 2)
```

(a)
```
count1 <- numeric(10000)
count2 <- numeric(10000)

for(i in 1:10000){
    count1[i] <- sum(pts[[i]] < 1)
    count2[i] <- sum(pts[[i]] > 1)
}
```

(b)
```
table(count1)
```

```
## count1
##    0    1    2    3    4    5    6    7    8    9   10
##  880 2047 2506 2091 1375  694  264  100   33    5    5
```

```
table(count2)
```

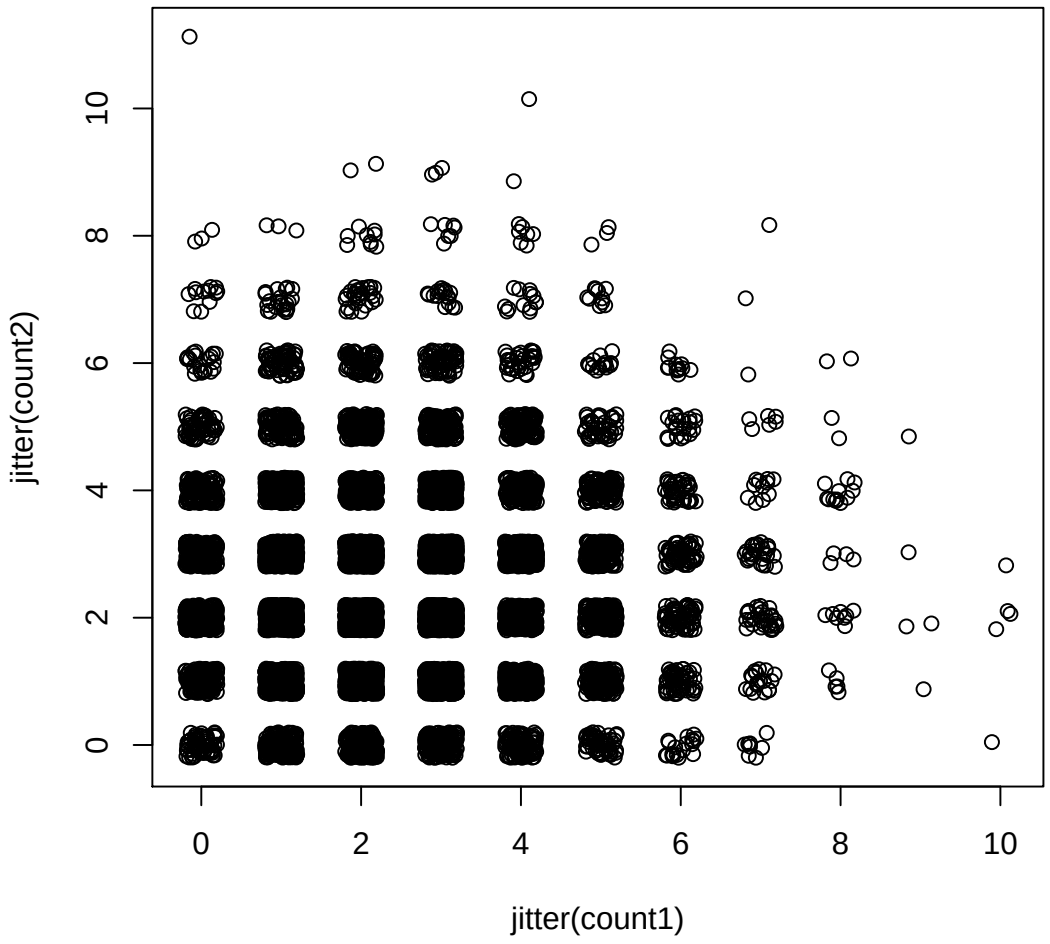
```
## count2
##    0    1    2    3    4    5    6    7    8    9   10   11
##  767 2057 2581 2131 1361  681  279  102   33    6    1    1
```

```
round(10000*dpois(0:12, 2.5))
```

```
##  [1]  821 2052 2565 2138 1336  668  278   99   31    9    2    0    0
```

The distributions look quite similar.

(c)
```
plot(jitter(count1), jitter(count2))
```

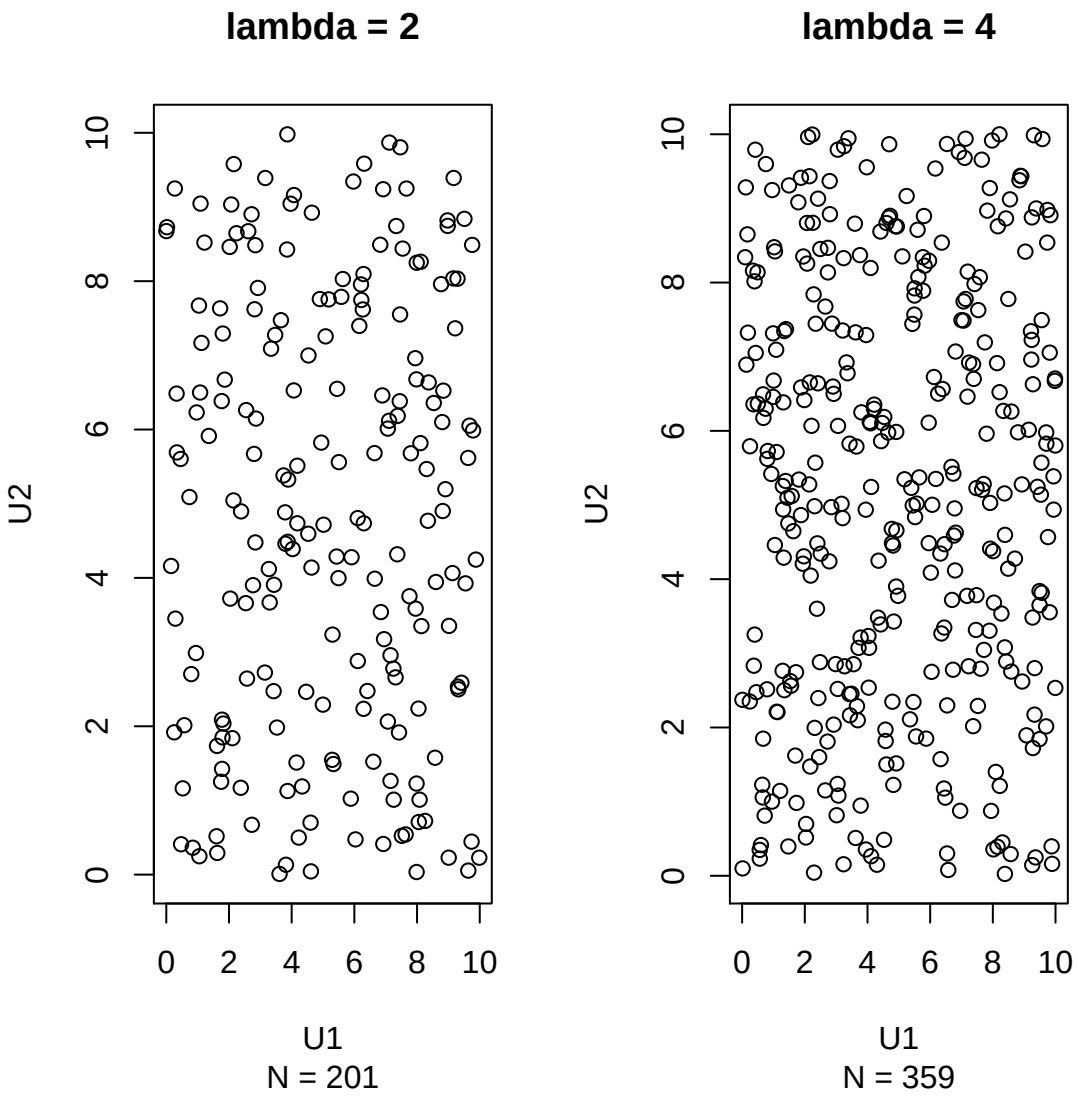


8.
```
N <- rpois(1, 0.023*24*62)
U <- sort(runif(N, 0, 62))
as.POSIXct("2016-07-01") + U*24*60*60
```

```
##  [1] "2016-07-01 01:57:11 EDT" "2016-07-01 05:19:35 EDT"
##  [3] "2016-07-01 05:24:23 EDT" "2016-07-02 18:34:38 EDT"
##  [5] "2016-07-05 10:23:56 EDT" "2016-07-05 20:05:36 EDT"
##  [7] "2016-07-06 18:44:26 EDT" "2016-07-08 19:58:28 EDT"
##  [9] "2016-07-10 07:38:41 EDT" "2016-07-13 15:28:27 EDT"
## [11] "2016-07-13 16:08:09 EDT" "2016-07-14 02:56:24 EDT"
## [13] "2016-07-17 04:01:04 EDT" "2016-07-17 14:50:43 EDT"
## [15] "2016-07-17 23:15:28 EDT" "2016-07-19 10:37:49 EDT"
## [17] "2016-07-22 05:11:20 EDT" "2016-07-22 23:08:15 EDT"
## [19] "2016-07-25 17:30:24 EDT" "2016-07-29 08:16:30 EDT"
## [21] "2016-07-31 02:35:10 EDT" "2016-07-31 11:18:08 EDT"
## [23] "2016-08-05 18:02:59 EDT" "2016-08-07 02:14:55 EDT"
## [25] "2016-08-09 18:08:02 EDT" "2016-08-12 09:55:13 EDT"
```

```
## [27] "2016-08-14 22:31:48 EDT" "2016-08-17 10:31:36 EDT"
## [29] "2016-08-18 07:17:18 EDT" "2016-08-20 21:01:45 EDT"
## [31] "2016-08-21 23:47:19 EDT" "2016-08-23 06:37:22 EDT"
## [33] "2016-08-23 21:08:27 EDT" "2016-08-23 21:24:02 EDT"
## [35] "2016-08-28 11:08:28 EDT" "2016-08-30 00:29:13 EDT"
## [37] "2016-08-30 17:10:35 EDT"
```

9.
```
par(mfrow = c(1, 2))
lambda <- 2
regionlength <- 10
regionwidth <- 10
N <- rpois(1, lambda*regionlength*regionwidth)
U1 <- runif(N, min = 0, max = regionwidth)
U2 <- runif(N, min = 0, max = regionlength)
plot(U1, U2, main = "lambda = 2", sub = paste("N =", N))

lambda <- 4
N <- rpois(1, lambda*regionlength*regionwidth)
U1 <- runif(N, min = 0, max = regionwidth)
U2 <- runif(N, min = 0, max = regionlength)
plot(U1, U2, main = "lambda = 4", sub = paste("N =", N))
```



5.3.4 Exponential random numbers

1. `r <- rexp(50000, rate = 3)`

   (a)
   ```
   mean(r < 1)
   ## [1] 0.94998
   pexp(1, rate = 3)
   ## [1] 0.9502129
   ```

   (b)
   ```
   mean(r)
   ## [1] 0.335176
   ```

```
1/3
```

```
## [1] 0.3333333
```

(c)
```
var(r)
```

```
## [1] 0.112321
```

```
1/3^2
```

```
## [1] 0.1111111
```

3.
```
r1 <- rexp(100000, rate=1/3)
r2 <- rexp(100000, rate=1/6)
r.min <- pmin(r1, r2)
mean(r.min)
```

```
## [1] 2.00572
```

```
var(r.min)
```

```
## [1] 4.020534
```

5.
```
count <- numeric(10000)
for(i in 1:10000){
    cum <- 0
    j <- 0
    while(cum <= 2){
        j <- j + 1
        r <- rexp(1, rate=2.5)
        cum <- cum + r
    }
    count[i] <- j - 1
}
table(count)
```

```
## count
##    0    1    2    3    4    5    6    7    8    9   10   11   12   13   14
##   69  327  809 1335 1784 1787 1430 1072  698  362  187   85   29   17    7
##   16   17
##    1    1
```

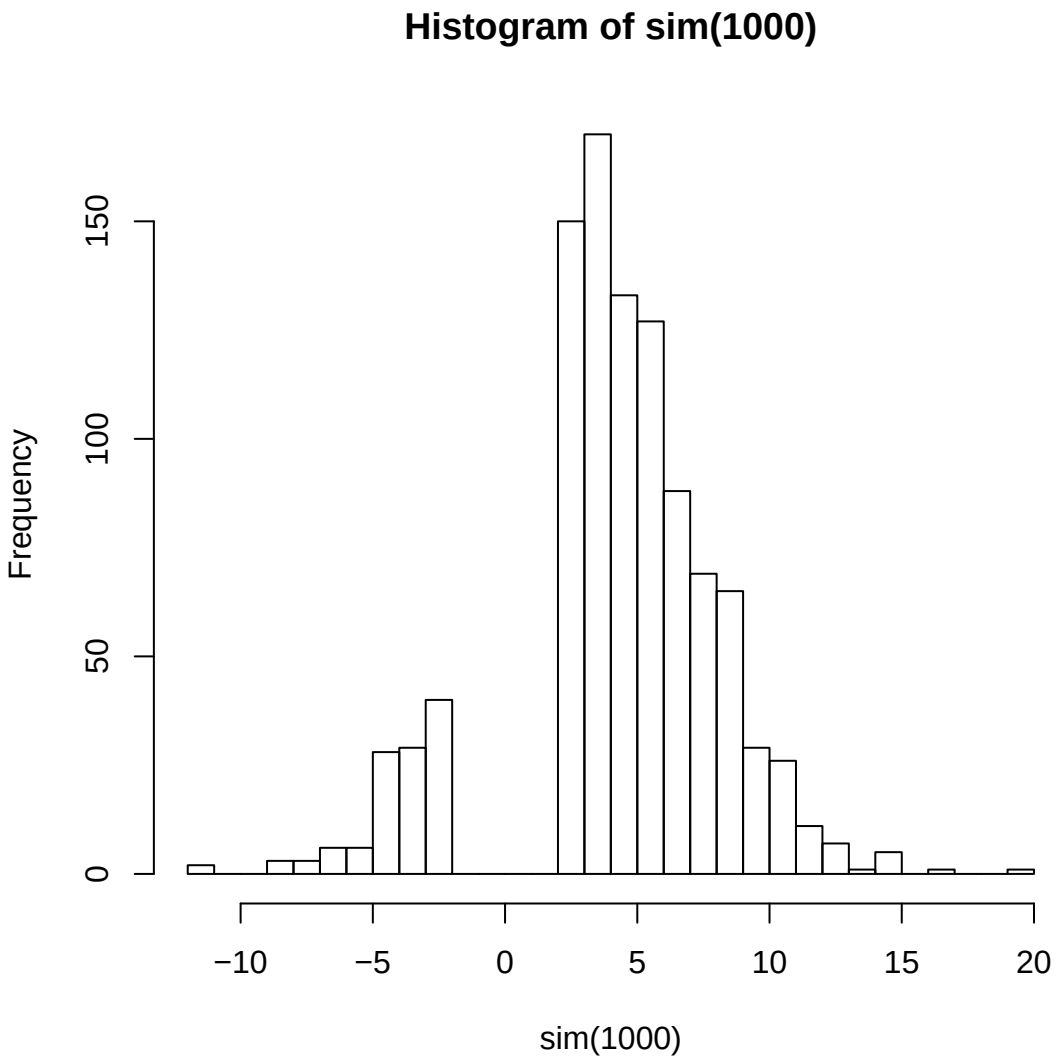
```
round(10000*dpois(0:20, lambda = 5))
```

```
##  [1]   67  337  842 1404 1755 1755 1462 1044  653  363  181   82   34   13
## [15]    5    2    0    0    0    0    0
```

5.3.6 All built-in distributions

1.
```r
r <- rnorm(100, mean = 51, sd = 5.2)
mean(r)

## [1] 51.10176

sd(r)

## [1] 5.116772
```

3.
```r
sim <- function(n){
    count <- 0
    ans <- numeric(n)
    while(count < n){
        X <- rnorm(n - count, mean = 3, sd = 4)
        X <- X[abs(X) > 2]
        ans[(count + 1):(count + length(X))] <- X
        count <- count + length(X)
    }
    ans
}
hist(sim(1000), breaks = "Scott")
```



5.
```r
r <- rchisq (10000, df = 8)
mean(r)

## [1] 8.018364

var(r)

## [1] 16.18888
```

5.6 Monte Carlo Integration

1.
```r
#Monte Carlo
u <- runif(1000)
mean(u)
```

```
## [1] 0.5068746

#Integrate function
f <- function(x){
    x
}
integrate(f, lower = 0, upper = 1)

## 0.5 with absolute error < 5.6e-15

#Monte Carlo
u <- runif(1000, min = 1, max = 3)
mean(u^2)*(3 - 1)

## [1] 8.674563

#Integrate function
f <- function(x){
    x^2
}
integrate(f, lower = 1, upper = 3)

## 8.666667 with absolute error < 9.6e-14

#Monte Carlo
u <- runif(1000, max = pi)
mean(sin(u))*(pi - 0)

## [1] 2.069972

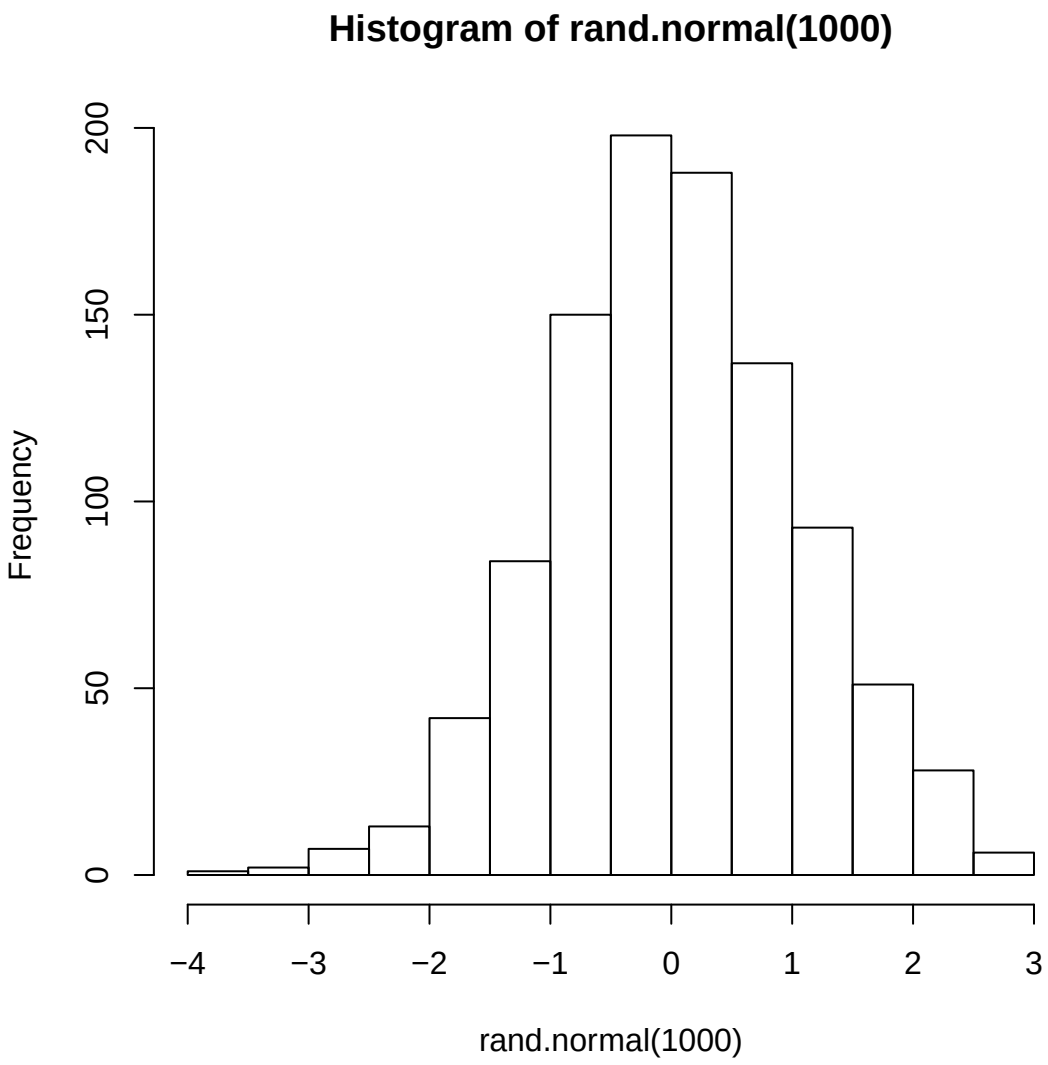
#Integrate function
f <- function(x){
    sin(x)
}
integrate(f, lower = 0, upper = pi)

## 2 with absolute error < 2.2e-14
```

5.7 Advanced simulation methods

1. ```
rand.normal <- function(n){
    r1 <- runif(n, min = -4, max = 4)
    r2 <- runif(n)
    ans <- r1[r2 < 2.5*dnorm(r1)]
    while(length(ans) < n){
        r1 <- runif(n, min = -4, max = 4)
        r2 <- runif(n)
        ans <- c(ans,r1[r2 < 2.5*dnorm(r1)])
    }
    ans[1:n]
```
```

```
}
hist(rand.normal(1000))
```



To handle the whole line, we'd need a proposal distribution with support on $(-\infty, \infty)$ instead of a uniform distribution.

4.
```
set.seed(91626)
probs <- runif(100) + 100
probs <- probs/sum(probs)
microbenchmark(randiscrete1(100, probs),
               randiscrete1(1000, probs),
               randiscrete1(10000, probs),
               randiscrete2(100, probs),
               randiscrete2(1000, probs),
               randiscrete2(10000, probs))

## Unit: microseconds
```

```
##                         expr       min         lq       mean    median
##    randiscrete1(100, probs)   578.658   608.9920   647.4518   621.088
##   randiscrete1(1000, probs)  5459.805  5678.2890  5962.3956  5766.220
##  randiscrete1(10000, probs) 57574.265 60486.7485 62494.7654 62484.210
##    randiscrete2(100, probs)   879.698   921.5515   998.2669   971.853
##   randiscrete2(1000, probs)  8509.370  8791.2105  9481.0834  9262.161
##  randiscrete2(10000, probs) 87624.497 92155.4540 96532.2479 96385.372
##          uq        max neval
##    630.8795   3405.899   100
##   5880.4545  10509.134   100
##  63433.5990  98544.488   100
##    998.1550   4788.992   100
##   9514.0520  14381.952   100
##  99778.9825 134648.541   100
```

For very long `probs` vectors, the rejection method is faster.

5.8 Chapter Exercises

1.
```
hitBall <- function(probSuccess) {
    rbinom(1, 1, probSuccess)
}

# hitBall generates a 1 (success) or a 0 (failure)

rally <- function(probs, hitter = 1) {
    serve <- 1
    while (serve == 1) {
        serve <- hitBall(probs[hitter])
        hitter <- 3 - hitter
    }
    hitter
}

# rally generates the outcome of a combination of serve and returns until first failed hit
# at which point the opponent wins the rally; probs is a 2-vector of success probabilities for
# the 2 players

game <- function(successProbs = c(.5, .5), winningScore = 11, initialServer = 1) {
    score <- c(0, 0)
    server <- initialServer
    while (max(score) < max(11, min(score) + 2)) {
        winner <- rally(successProbs, server)
        score[winner] <- score[winner] + 1
        if (sum(score) %% 2 == 0) server <- 3 - server
    }
    score
}

# the probability of a successful serve is usually higher than any subsequent return; the
# following code adds this feature, through the 2-vector serveProbs which specifies
# the probability of success on the serve for each player
```

```
rally <- function(probs, hitter = 1, serveProbs) {
  serve <- 1
  serve <- hitBall(serveProbs[hitter])
  hitter <- 3 - hitter
  while (serve == 1) {
    serve <- hitBall(probs[hitter])
    hitter <- 3 - hitter
  }
  hitter
}


game <- function(successProbs = c(0.5, 0.5), winningScore = 11, initialServer = 1, serveProbs = c(0
  score <- c(0, 0)
  server <- initialServer
  while (max(score) < max(11, min(score) + 2)) {
    winner <- rally(successProbs, server, serveProbs)
    score[winner] <- score[winner] + 1
    if (sum(score) %% 2 == 0) server <- 3 - server
  }
  score
}
```

2. The question asks for a population of 10000, but that takes too long, so we'll use $N = 100$.

   (a)
```
simulate <- function(N, p, init = 1, stop = N){
    time.line <- 0
    time.current <- 0
    n.sick <- init

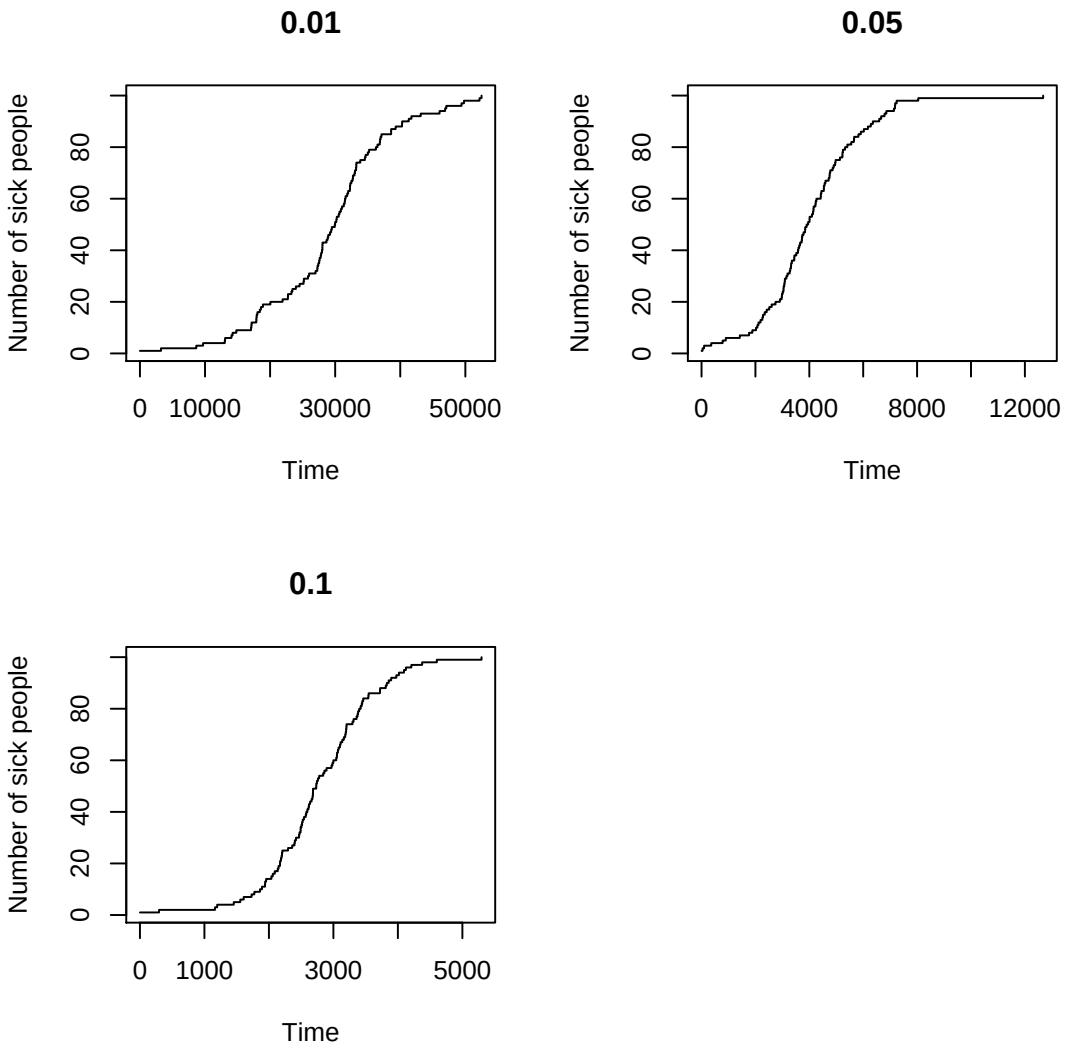
    while (n.sick < stop) {
        time.current <- time.current + 1

        #if encounters==0, then none of the two persons is sick
        #if encounters==1, then one of the two persons is sick
        #if encounters==2, then both of the two persons are sick
        encounters <- rhyper(nn = 1, m = n.sick, n = N - n.sick, k = 2)

        if (encounters == 1){
            contage <- rbinom(n = 1, size = 1, p)
            if (contage == 1){
                #record current time
                time.line <- c(time.line, time.current)
                n.sick <- n.sick + 1
            }
        }
    }
    time.line
}

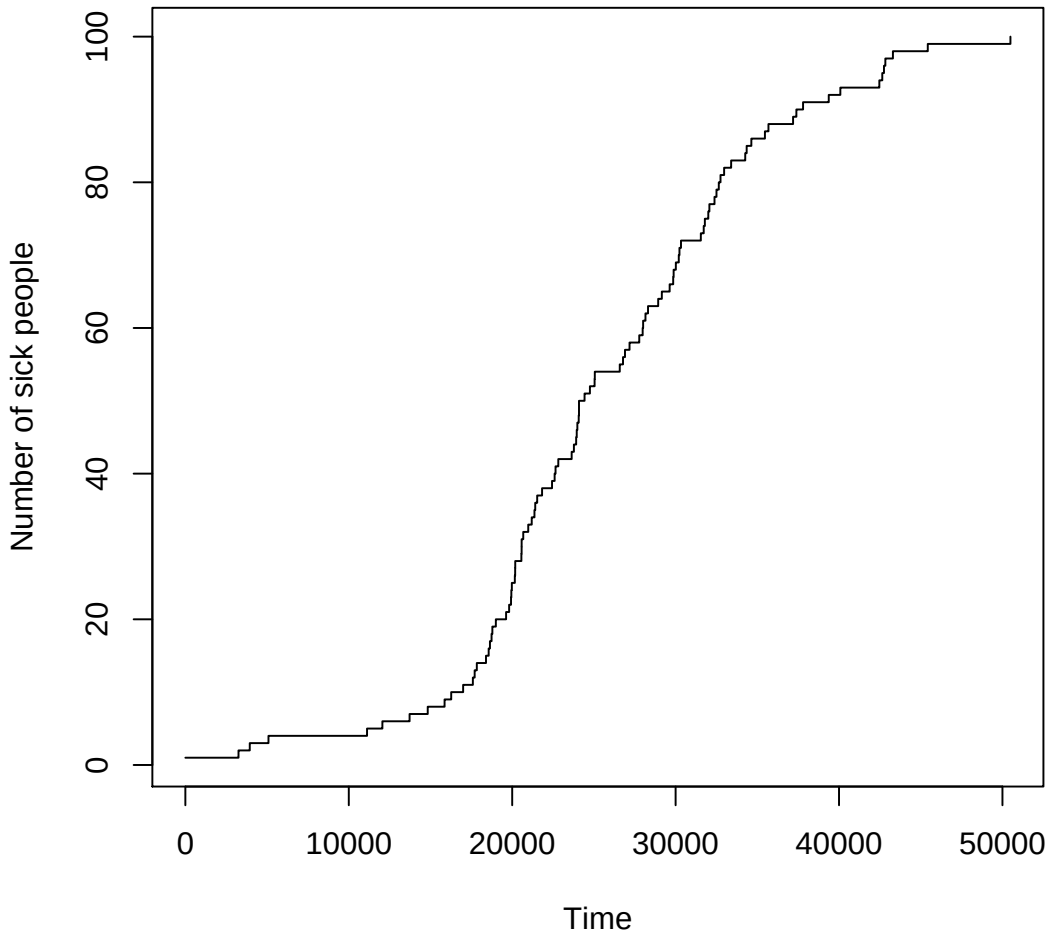
N <- 100
```

```r
par(mfrow = c(2,2))
for(p in c(0.01, 0.05, 0.1)) {
  ans <- simulate(N, p)
  plot(ans, 1:N, xlab='Time', ylab='Number of sick people',
       type = 's', main = p)
}
```



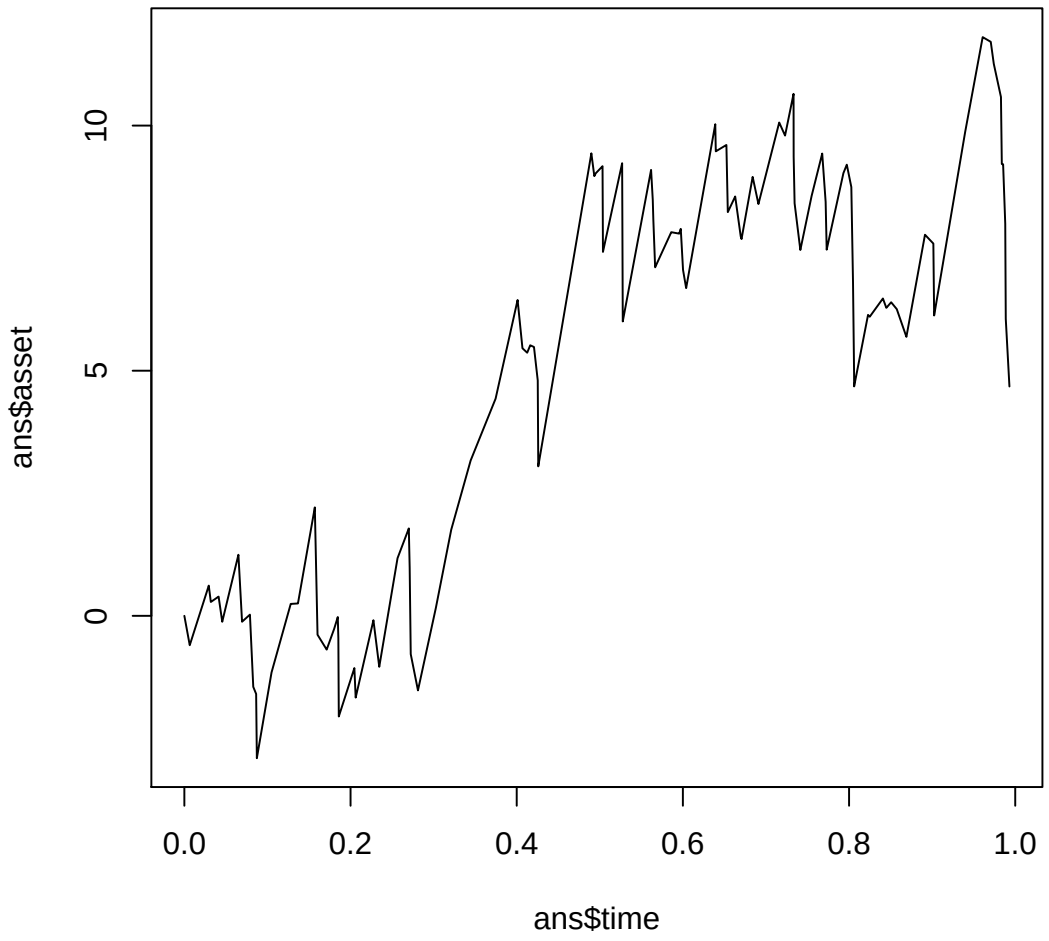
(d)
```r
simulate <- function(N, p, precover, init = 1, stop = N){
    time.line <- 0
    nsick.line <- init
    time.current <- 0
    n.sick <- init

    while (n.sick > 0 & n.sick < stop) {
        time.current <- time.current + rexp(1, rate = 1/5)
        #if encounters==0, then none of the two persons is sick
        #if encounters==1, then one of the two persons is sick
```

```
        #if encounters==2, then both of the two persons are sick
        encounters <- rhyper(nn = 1, m = n.sick, n = N - n.sick, k = 2)

        if (encounters == 1){
            contage <- rbinom(n = 1, size = 1, p)
            if (contage == 1){
                #record current time
                n.sick <- n.sick + 1
            }
        }

        n.sick <- rbinom(1, size = n.sick, prob = 1 - precover)
        if (n.sick != nsick.line[length(nsick.line)]) {
                time.line <- c(time.line, time.current)
            nsick.line <- c(nsick.line, n.sick)
        }
    }
    cbind(time.line, nsick.line)
}


N <- 100
p <- .05
ans <- simulate(N, p, 0)
plot(ans, xlab='Time', ylab='Number of sick people', type = 's')
```



4. (a)
```r
simulate <- function(){
    N <- rpois(1, lambda = 100)
    claimSize <- rgamma(N, shape = 2, rate = 2)
    claimTime <- sort(runif(N))

    asset <- 0
    asset.1 <- 105 * claimTime[1] - claimSize[1]
    asset <- c(asset, asset.1)

    for(i in 2:N){
        len <- length(asset)
        asset.i <- asset[len] + (claimTime[i] - claimTime[i-1])*105 -
                   claimSize[i]
        asset <- c(asset, asset.i)
    }
    claimTime <- c(0, claimTime)
    list(time=claimTime, asset=asset)
```

```
}
ans <- simulate()
plot(ans$time, ans$asset, type='l')
```



(b)
```
minimum <- c()
final <- c()
for(i in 1:1000){
    ans <- simulate()
    minimum <- c(minimum, min(ans$asset))
    final <- c(final, rev(ans$asset)[1])
}

mean(minimum)

## [1] -7.374455

mean(final)

## [1] 3.829063
```

Chapter 6

Computational Linear Algebra

6.1 Vectors and Matrices in R

6.1.1 Constructing matrix objects

1.
```
A <- matrix(rep(seq(1, 5), 5), nrow = 5) +
     matrix(rep(seq(0, 4), 5), byrow = TRUE, nrow = 5)
A
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    2    3    4    5    6
## [3,]    3    4    5    6    7
## [4,]    4    5    6    7    8
## [5,]    5    6    7    8    9
```

```
Hankel <- function(x){
 A <- matrix(rep(seq(1, x), x), nrow = x) +
      matrix(rep(seq(0, x - 1), x), byrow = TRUE, nrow = x)
 return(A)
 }
Hankel(10)
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    1    2    3    4    5    6    7    8    9    10
##  [2,]    2    3    4    5    6    7    8    9   10    11
##  [3,]    3    4    5    6    7    8    9   10   11    12
##  [4,]    4    5    6    7    8    9   10   11   12    13
##  [5,]    5    6    7    8    9   10   11   12   13    14
##  [6,]    6    7    8    9   10   11   12   13   14    15
##  [7,]    7    8    9   10   11   12   13   14   15    16
##  [8,]    8    9   10   11   12   13   14   15   16    17
##  [9,]    9   10   11   12   13   14   15   16   17    18
## [10,]   10   11   12   13   14   15   16   17   18    19
```

```
Hankel(12)
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
##  [1,]    1    2    3    4    5    6    7    8    9    10    11    12
```

```
##  [2,]    2    3    4    5    6    7    8    9   10   11   12   13
##  [3,]    3    4    5    6    7    8    9   10   11   12   13   14
##  [4,]    4    5    6    7    8    9   10   11   12   13   14   15
##  [5,]    5    6    7    8    9   10   11   12   13   14   15   16
##  [6,]    6    7    8    9   10   11   12   13   14   15   16   17
##  [7,]    7    8    9   10   11   12   13   14   15   16   17   18
##  [8,]    8    9   10   11   12   13   14   15   16   17   18   19
##  [9,]    9   10   11   12   13   14   15   16   17   18   19   20
## [10,]   10   11   12   13   14   15   16   17   18   19   20   21
## [11,]   11   12   13   14   15   16   17   18   19   20   21   22
## [12,]   12   13   14   15   16   17   18   19   20   21   22   23
```

3.
```
W <- cbind(c(1, 1, 1, 1, 1, 1, 1),
           c(2, 3, 4, 5, 6, 7, 8),
           c(4, 7, 5, 6, 7, 5, 3))
W
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    4
## [2,]    1    3    7
## [3,]    1    4    5
## [4,]    1    5    6
## [5,]    1    6    7
## [6,]    1    7    5
## [7,]    1    8    3
```

6.1.2 Accessing matrix elements; row and column names

1.
```
P <- matrix(c(0.2, 0.3, 0.8, 0.7), ncol = 2)
rownames(P) <- c("sunny", "rainy")
colnames(P) <- c("sunny", "rainy")
P
```

```
##       sunny rainy
## sunny   0.2   0.8
## rainy   0.3   0.7
```

```
w <- character(30)
w[1] <- "sunny"
for (i in 2:30)
  w[i] <- sample(colnames(P), 1,
                 prob = P[w[i-1], ])
w
```

```
##  [1] "sunny" "rainy" "rainy" "rainy" "rainy" "rainy" "rainy" "rainy"
##  [9] "rainy" "sunny" "rainy" "rainy" "sunny" "sunny" "rainy" "sunny"
## [17] "rainy" "rainy" "rainy" "rainy" "rainy" "rainy" "sunny" "rainy"
## [25] "rainy" "rainy" "rainy" "rainy" "rainy" "rainy"
```

3.
```
matrix["Pardeep", "height"] <- 162
matrix["Hao", "height"] <- 181
matrix["Hao", "weight"] <- 68
matrix
```

```
##         height weight
## Neil       172     62
## Cindy      168     64
## Pardeep    162     51
## Deepak     175     71
## Hao        181     68
```

6.1.3 Matrix Properties

1.
```
A <- matrix(c(3, 5, 4, 8), ncol = 2)
det(A)
```

```
## [1] 4
```

```
det(t(A))
```

```
## [1] 4
```

```
A <- matrix(c(3, 20, 15, 7), ncol = 2)
det(A)
```

```
## [1] -279
```

```
det(t(A))
```

```
## [1] -279
```

```
A <- matrix(c(4, 2, 25, 23), ncol = 2)
det(A)
```

```
## [1] 42
```

```
det(t(A))
```

```
## [1] 42
```

6.1.4 Triangular matrices

1.
```
H3 <- matrix(c(1, 1/2, 1/3, 1/2, 1/3, 1/4, 1/3, 1/4, 1/5), nrow=3)
Hnew <- H3
Hnew[lower.tri(H3, diag = FALSE)] <- 0
Hnew
```

```
##      [,1]      [,2]      [,3]
```

```
## [1,]    1 0.5000000 0.3333333
## [2,]    0 0.3333333 0.2500000
## [3,]    0 0.0000000 0.2000000
```

3.
```
X <- matrix(c(1, 2, 3, 1, 4, 9), ncol = 2)
X[3, 2]

## [1] 9

X[3, 2, drop = FALSE]

##      [,1]
## [1,]    9

dim(X[3, 2])

## NULL

dim(X[3, 2, drop = FALSE])

## [1] 1 1
```

6.2 Matrix multiplication and inversion

1.
```
X <- matrix(c(1, 2, 3, 1, 4, 9), ncol = 2)
1.5*X

##      [,1] [,2]
## [1,]  1.5  1.5
## [2,]  3.0  6.0
## [3,]  4.5 13.5
```

3. (a) No.
   (b) Only vector multiplications.
   (c)
```
A <- matrix(rep(1, 1000000), nrow = 1000) # a matrix of 1s
B <- diag(1000) # 1000 x 1000 identity matrix
v <- rep(1, 1000) # a vector of 1s
system.time(A %*% B %*% v)

##    user  system elapsed
##    1.13    0.00    1.13

system.time((A %*% B) %*% v)

##    user  system elapsed
##    1.12    0.00    1.12

system.time(A %*% (B %*% v))

##    user  system elapsed
##       0       0       0
```

This varies from computer to computer. Some linear algebra libraries recognize the identity matrix and essentially eliminate B.

6.2.3 Matrix inversion in R

1.
```
XX <- t(X) %*% X
XXinv <- solve(XX)
XXinv

##             [,1]       [,2]
## [1,]  1.2894737 -0.4736842
## [2,] -0.4736842  0.1842105

# Verification:
crossprod(XX, XXinv)

##               [,1]          [,2]
## [1,]  1.000000e+00 -1.332268e-15
## [2,] -1.887379e-15  1.000000e+00
```

This is numerically close to the identity matrix.

3. (a)
```
Hilbert <- function(n) {
    A <- matrix(rep(NA, n*n), nrow=n)
    for(i in 1:n){
        for(j in 1:n){
            A[i, j] <- 1/(i + j - 1)
        }
    }
    A
}

# or

Hilbert <- function(n) {
    outer(1:n, 1:n, function(x, y) 1/(x + y - 1))
}
```

(b) Yes. (Showing that all Hilbert matrices are invertible is not obvious, but see `https://en.wikipedia.org/wiki/Hilbert_matrix` for the formula.)

(c)
```
solve(Hilbert(1))

##      [,1]
## [1,]    1

solve(Hilbert(2))

##      [,1] [,2]
## [1,]    4   -6
## [2,]   -6   12
```

```
qr.solve(Hilbert(1))
```

```
##      [,1]
## [1,]    1
```

```
qr.solve(Hilbert(2))
```

```
##      [,1] [,2]
## [1,]    4   -6
## [2,]   -6   12
```

6.2.4 Solving linear systems

1.
```
X1 <- c(10, 11, 12, 13, 14, 15)
X2 <- X1^2
X3 <- X1^3
X4 <- X1^4
X5 <- X1^5
X0 <- c(1, 1, 1, 1, 1, 1)
A <- cbind(X0, X1, X2, X3, X4, X5)
f <- matrix(c(25, 16, 26, 19, 21, 20), nrow = 6)
a <- solve(A, f)
a
```

```
##             [,1]
## X0  2.536100e+05
## X1 -1.025510e+05
## X2  1.650092e+04
## X3 -1.320667e+03
## X4  5.258333e+01
## X5 -8.333333e-01
```

```
A %*% a
```

```
##      [,1]
## [1,]   25
## [2,]   16
## [3,]   26
## [4,]   19
## [5,]   21
## [6,]   20
```

```
# or

x <- c(10, 11, 12, 13, 14, 15)
y <- c(25, 16, 26, 19, 21, 20)
A <- outer(x, 0:5, function(x, y) x^y)
A
```

```
##      [,1] [,2] [,3] [,4]  [,5]   [,6]
## [1,]    1   10  100 1000 10000 100000
## [2,]    1   11  121 1331 14641 161051
## [3,]    1   12  144 1728 20736 248832
```

```
## [4,]    1   13  169 2197 28561 371293
## [5,]    1   14  196 2744 38416 537824
## [6,]    1   15  225 3375 50625 759375

A <- outer(x, 0:5, function(x, y) x^y)
solve(A, y)   # Solve Aa = y  for a

## [1]  2.536100e+05 -1.025510e+05  1.650092e+04 -1.320667e+03  5.258333e+01
## [6] -8.333333e-01
```

$f(x) = 253610 - 102551x + 16500.92x^2 - 1320.667x^3 + 52.58333x^4 - .8333x^5$

6.3 Eigenvalues and eigenvectors

1.
```
X <- matrix(c(1, 2, 3, 1, 4, 9), ncol=2)
H <- X %*% solve(t(X) %*% X) %*% t(X)
H

##            [,1]      [,2]       [,3]
## [1,]  0.5263158 0.4736842 -0.1578947
## [2,]  0.4736842 0.5263158  0.1578947
## [3,] -0.1578947 0.1578947  0.9473684
```

5. Hint: Note that $HX = X$ and $H(I - X) = 0$.

6.4 Other matrix decompositions

1.
```
SVD <- svd(Hilbert(4))
SVD

## $d
## [1] 1.5002142801 0.1691412202 0.0067382736 0.0000967023
##
## $u
##            [,1]       [,2]       [,3]        [,4]
## [1,] -0.7926083  0.5820757 -0.1791863 -0.02919332
## [2,] -0.4519231 -0.3705022  0.7419178  0.32871206
## [3,] -0.3224164 -0.5095786 -0.1002281 -0.79141115
## [4,] -0.2521612 -0.5140483 -0.6382825  0.51455275
##
## $v
##            [,1]       [,2]       [,3]        [,4]
## [1,] -0.7926083  0.5820757 -0.1791863 -0.02919332
## [2,] -0.4519231 -0.3705022  0.7419178  0.32871206
## [3,] -0.3224164 -0.5095786 -0.1002281 -0.79141115
## [4,] -0.2521612 -0.5140483 -0.6382825  0.51455275

inv <- SVD$v %*% diag(1/SVD$d) %*% t(SVD$u)
inv
```

```
##      [,1]  [,2]  [,3]  [,4]
## [1,]   16  -120   240  -140
## [2,] -120  1200 -2700  1680
## [3,]  240 -2700  6480 -4200
## [4,] -140  1680 -4200  2800

inv %*% Hilbert(4)

##               [,1]          [,2]          [,3]          [,4]
## [1,]  1.000000e+00  4.839878e-15  1.295838e-15 -1.427677e-15
## [2,]  1.136868e-13  1.000000e+00  4.130030e-14  5.570544e-14
## [3,] -2.273737e-13  2.159384e-14  1.000000e+00 -8.038015e-14
## [4,]  0.000000e+00 -1.280642e-13 -2.589595e-14  1.000000e+00
```

2. This is kind of a square root:

```
SQRT <- chol(Hilbert(4))
SQRT

##      [,1]      [,2]      [,3]       [,4]
## [1,]    1 0.5000000 0.3333333 0.25000000
## [2,]    0 0.2886751 0.2886751 0.25980762
## [3,]    0 0.0000000 0.0745356 0.11180340
## [4,]    0 0.0000000 0.0000000 0.01889822

t(SQRT) %*% SQRT

##           [,1]      [,2]      [,3]      [,4]
## [1,] 1.0000000 0.5000000 0.3333333 0.2500000
## [2,] 0.5000000 0.3333333 0.2500000 0.2000000
## [3,] 0.3333333 0.2500000 0.2000000 0.1666667
## [4,] 0.2500000 0.2000000 0.1666667 0.1428571
```

Since the matrix is symmetric, here's a true square root:

```
SVD <- svd(Hilbert(4))
SQRT <- SVD$u %*% diag(sqrt(SVD$d)) %*% t(SVD$v)
SQRT

##           [,1]      [,2]      [,3]      [,4]
## [1,] 0.9114604 0.3390312 0.1927197 0.1309843
## [2,] 0.3390312 0.3528552 0.2474522 0.1806979
## [3,] 0.1927197 0.2474522 0.2411021 0.2085577
## [4,] 0.1309843 0.1806979 0.2085577 0.2226032

SQRT %*% SQRT

##           [,1]      [,2]      [,3]      [,4]
## [1,] 1.0000000 0.5000000 0.3333333 0.2500000
## [2,] 0.5000000 0.3333333 0.2500000 0.2000000
## [3,] 0.3333333 0.2500000 0.2000000 0.1666667
## [4,] 0.2500000 0.2000000 0.1666667 0.1428571
```

6.6 Chapter Exercises

1. (a)
```
P <- matrix(c(0.1, 0.4, 0.3, 0.2,
              0.2, 0.1, 0.4, 0.3,
              0.3, 0.2, 0.1, 0.4,
              0.4, 0.3, 0.2, 0.1), nrow=4)
apply(P, 1, sum)
```
```
## [1] 1 1 1 1
```

   (b)
```
P2 <- P %*% P
P2
```
```
##      [,1] [,2] [,3] [,4]
## [1,] 0.26 0.28 0.26 0.20
## [2,] 0.20 0.26 0.28 0.26
## [3,] 0.26 0.20 0.26 0.28
## [4,] 0.28 0.26 0.20 0.26
```
```
P3 <- P %*% P2
P3
```
```
##       [,1]  [,2]  [,3]  [,4]
## [1,] 0.256 0.244 0.240 0.260
## [2,] 0.260 0.256 0.244 0.240
## [3,] 0.240 0.260 0.256 0.244
## [4,] 0.244 0.240 0.260 0.256
```
```
P5 <- P2 %*% P3
P5
```
```
##         [,1]    [,2]    [,3]    [,4]
## [1,] 0.25056 0.25072 0.24928 0.24944
## [2,] 0.24944 0.25056 0.25072 0.24928
## [3,] 0.24928 0.24944 0.25056 0.25072
## [4,] 0.25072 0.24928 0.24944 0.25056
```
```
P10 <- P5 %*% P5
P10
```
```
##           [,1]      [,2]      [,3]      [,4]
## [1,] 0.2500000 0.2500016 0.2500000 0.2499983
## [2,] 0.2499983 0.2500000 0.2500016 0.2500000
## [3,] 0.2500000 0.2499983 0.2500000 0.2500016
## [4,] 0.2500016 0.2500000 0.2499983 0.2500000
```

   (c)
```
solve(rbind(rep(1, 4), (diag(rep(1, 4)) - t(P))[-4, ]), c(1, 0, 0, 0))
```
```
## [1] 0.25 0.25 0.25 0.25
```

   The values in P^n and the values in x all appear to be converging to 0.25.

   (d)
```
set.seed(361)
pseudo <- function(n) {
    Y <- numeric(n)
    Y[1] <- 1
```

```
    for(j in 2:n) {
        Y[j] <- sample(1:4, 1,
                       prob=P[Y[j-1], ],
                       replace=T)
    }
    return(Y)
}

yresult <- pseudo(10000)
```

(e)
```
table(yresult)
```

```
## yresult
##    1    2    3    4
## 2502 2508 2523 2467
```

The relative frequency distribution is near x.

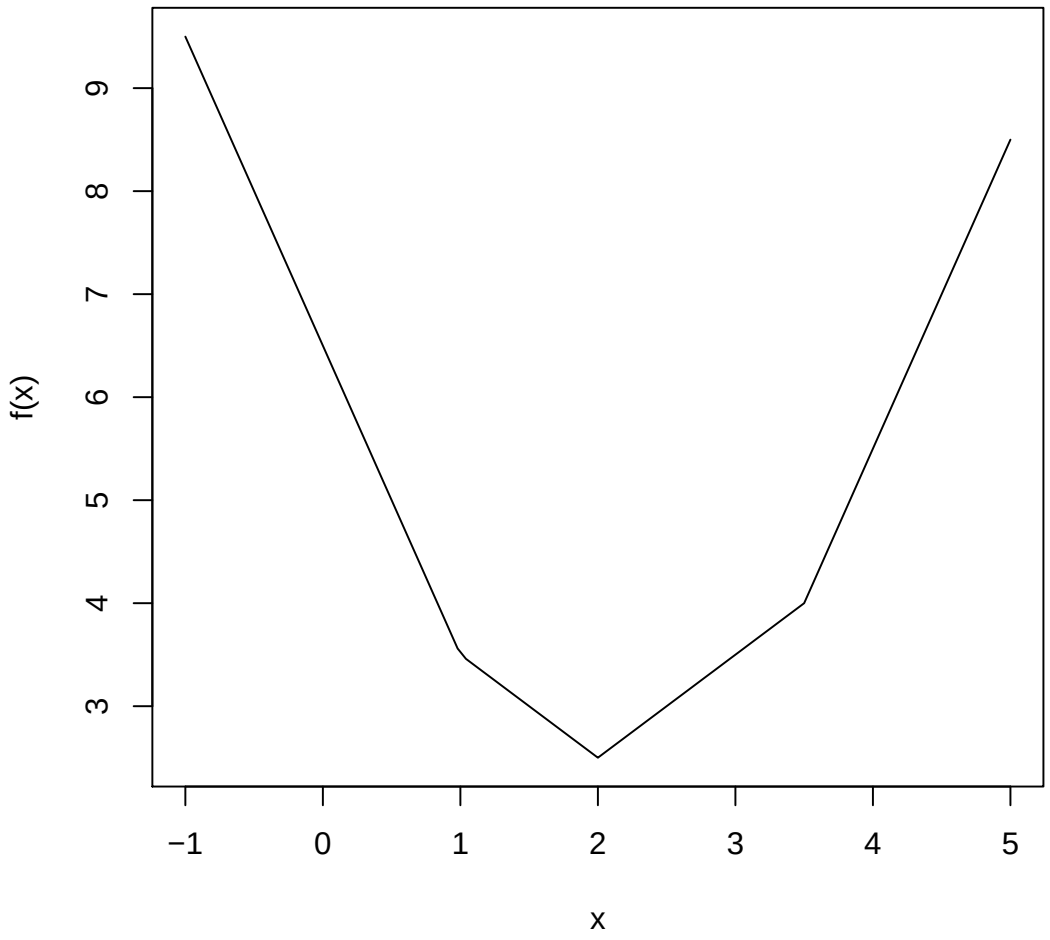
Chapter 7

Numerical optimization

7.1 The golden section search method

```
golden <- function(f, a, b, tol = 0.0000001){
   ratio <- 2/(sqrt(5) + 1)
   x1 <- b - ratio*(b - a)
   x2 <- a + ratio*(b - a)

   f1 <- f(x1)
   f2 <- f(x2)

   while( abs(b -a) > tol){
           if(f2 > f1){
               b <- x2
               x2 <- x1
               f2 <- f1
               x1 <- b - ratio*(b - a)
               f1 <- f(x1)
           }else{
               a <- x1
               x1 <- x2
               f1 <- f2
               x2 <- a + ratio*(b - a)
               f2 <- f(x2)
           }
   }
   return((a + b) / 2)
}
```

1. (a)
```
f <- function(x) {
    abs(x - 3.5) + abs(x - 2) + abs(x - 1)
}
curve ( f, from = -1, to = 5)
```

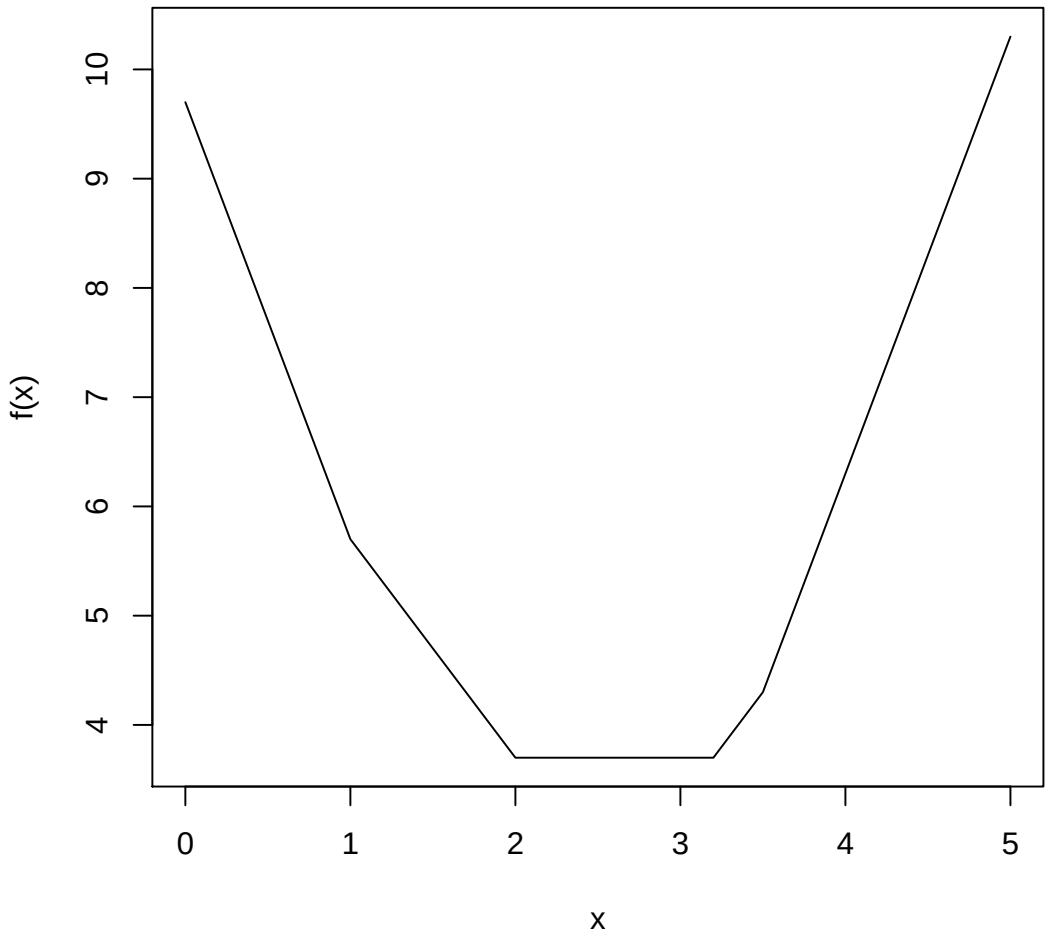


This indicates that there is only one minimum close to zero and it is located to the right of zero.

```
golden( f, 1, 5)
```

```
## [1] 2
```

(b)
```
f <- function(x) {
      abs(x - 3.2) + abs(x - 3.5) + abs(x - 2) + abs(x - 1)
}
curve(f, from = 0, to = 5)
```



This shows that between 1 and 4 there is more than one minimum point.

```
golden( f, 1, 2.15)
## [1] 2.15
golden( f, 1, 5)
## [1] 3.2
golden( f, 1, 3)
## [1] 3
golden( f, 2.5, 3.5)
## [1] 3.2
```

3.
```
goldenmax <- function(f, a, b, tol = 0.0000001)
        golden(function(x) -f(x), a, b, tol = 0.0000001)
```

```r
f <- function(x) -(x-2)^2
goldenmax(f, 1, 10)

## [1] 2
```

7.4 Built-in functions

1. (a)
```r
f <- function(x) {
     abs(x - 3.5) + abs(x - 2) + abs(x - 1)
}
optimize( f, c(1, 3))

## $minimum
## [1] 2
##
## $objective
## [1] 2.5
```

   (b)
```r
f <- function(x) {
    abs(x - 3.2) + abs(x - 3.5) + abs(x - 2) + abs(x - 1)
}
optimize( f, c(1, 4))

## $minimum
## [1] 2.145956
##
## $objective
## [1] 3.7
```

7.5 Linear programming

1. (a)
```r
library(lpSolve)
a.lp <- lp(objective.in = c(1, 3, 4, 1),
           const.mat = matrix(c(1, -2, 0, 0,
                                0, 3, 1, 0,
                                0, 1, 0, 1), nrow = 3, byrow = TRUE),
           const.rhs = c(9, 9, 10),
           const.dir = c(">=", ">=", ">="))
a.lp

## Success: the objective function is 31

a.lp$solution

## [1] 15  3  0  7
```

   (b) The solution will not change since the optimal solution requires that all decision variables are integers.

(c)
```
c.lp <- lp(objective.in = c(1, -3, 4, 1),
           const.mat = matrix(c(1, -2, 0, 0,
                                0, 3, 1, 0,
                                0, 1, 0, 1), nrow = 3, byrow = TRUE),
           const.rhs = c(9, 9, 10),
           const.dir = c(">=", ">=", ">="))
c.lp

## Error: status 3
```

This says that the solution is unbounded.

7.6 Chapter exercises

1.
```
x <- c( 0.45, 0.08, -1.08, 0.92, 1.65, 0.53, 0.52, -2.15, -2.2, -0.32, -1.87, -0.16,
 -0.19, -0.98, -0.2, 0.67, 0.08, 0.38, 0.76, -0.78)
y <- c( 1.26, 0.58, -1, 1.07, 1.28, -0.33, 0.68, -2.22, -1.82, -1.17, -1.54, 0.35,
 -0.23, -1.53, 0.16, 0.91, 0.22, 0.44, 0.98, -0.98)
library(quadprog)
X <- cbind( rep(1, 20), x)
XX <- t(X) %*% X
Xy <- t(X) %*% y
A <- matrix( c(1, -1), ncol = 1)
b <- 0
ans <- solve.QP( Dmat = XX, dvec = Xy, Amat = A, bvec = b)
ans$solution

## [1] 0.5314036 0.5314036
```

This implies that the intercept = slope = 0.5314036

3.
```
A <- matrix( c( 1, 1, -1, 0, 0, 0, 0, 1, 0, 0, 1, -1, 0, 0, 1, 0, 0, 0, 0, 1, -1),
 nrow=3, byrow=TRUE)
D <- matrix( c( 0.01, 0.002, 0.002,
                0.002, 0.01, 0.002,
                0.002, 0.002, 0.01), nrow = 3)
x <- c( 0.002, 0.005, 0.01)
b <- c( 1, 0, -0.5, 0, -0.5, 0, -0.5)
ans <- solve.QP(2*D, x, A, b, meq = 1)
ans$solution

## [1] 0.15625 0.34375 0.50000
```

5.
```
D <- matrix( c( 0.01, 0,
                0, 0.04), nrow = 2, byrow = TRUE)
A <- matrix( c( 1, 1, 1,
                0, 0, 1), nrow = 2, byrow = TRUE)
x <- c( 0.005, 0.01)
b <- c( 1, 0, 0)
```

(a)
```
ans <- solve.QP(D, x, A, b, meq = 1)
ans$solution
```

```
## [1] 1.00 0.25
```

(b)
```
ans <- solve.QP(2*D, x, A, b, meq = 1)
ans$solution
```

```
## [1] 1.000 0.125
```