Some Lecture Notes for STAT 812 Computational Statistics

Longhai Li
Department of Math and Statistics
University of Saskatchewan
106 Wiggins Road
Saskatoon, Saskatchewan, CANADA, S7N5E6

Email:longhai@math.usask.ca
Homepage: http://math.usask.ca/~longhai

Important notes**:**
These lecture notes are very rough without guarantee of completeness or accuracy**.** It serves as a reference for students taking stat**812** only**.**

Contents

Chapter 1

Introduction

Why do we need to use computer in statistics?

- Needed to calculate probability $P(X \in A)$

  e.g. $X \sim N(0,1)$, $P(X > 1) =?$, $X \sim$Binomial(n,p), $P(X > k) =?$

- Needed to evaluate statistical methods
  - Evaluating a point estimator $\hat{\theta}$ by computing its MSE $E((\hat{\theta}-\theta)^2)$, or more generally $E(L(\hat{\theta}, \theta))$. An estimator is better than the other if it has smaller MSE or average loss.
  - Evaluating a test by computing its power $P(T \in R_\alpha \mid H_1)$, where R_α is the rejection region for a given confidence level α. A test is superior then the other if it has larger power.
- Needed to carry out statistical methods
  - Calculating p-value: $P(T > t|H_0)$, where t is the observed value of a test statistics T.
  - Maximum likelihood estimation

    $$\hat{\theta}_{MLE} = \arg\max_{\theta} L(\theta) \tag{1.1}$$

    where $L(\theta) = \log(P(Data|\theta))$, called log likelihood function. In most pratical problems, we can not find $\hat{\theta}_{MLE}$ analytically. For example, we can not find analytical solutions for logistic regression.
  - Bayesian inference

    The procedure of Bayesian inference:

    First, we assign a prior distribution $P(\theta)$ to unknown parameter θ, which reflects our knowledge about θ before seeing data.

    Next, we collect data $Data$. The knowledge about θ is then updated:

    $$P(\theta|Data) = \frac{P(Data|\theta)P(\theta)}{P(Data)} \tag{1.2}$$

Finally, we make decision involving unknow quantity Y based on the conditional probability distribution $P(Y|Data)$, which is an integral:

$$P(Y|Data) = \int P(Y|\theta)P(\theta|Data)d\theta \tag{1.3}$$

The above integral is usually intractable analytically.

– Fitting linear models

The least square estimator is $\hat{\beta} = (X'X)^{-1}X'Y$. Calculating the inverse is very expensive and prone to numerical error. We need to avoid it.

Chapter 2

Introduction to R with examples

2.1 Invoking R

2.1.1 From unix terminal

```
R [options] [<infile] [>outfile]
```
or
```
R CMD BATCH infile
```

Explanations:

- Without specifying anything in `[ ]`, R will be opened from a terminal, waiting for inputing R commands. Output will be printed on screen. To exit from R, type `q()`, then you will be asked whether to save the image or not. If choose to save, the objects created will be saved in file ".RData", and all the commands you have input so far will be saved in file ".Rhistory". Note: these two files are hidden files.
- If "infile" is given, R will execute the R commands in "infile" and output is written to "outfile" or printed on screen if it is empty. You must specify in `[options]` whether to save objects: `--save,--no-save`.

  Demonstration:

  A file "demo-startup.R" is shown as follows:

  ```
  a <- matrix(1:8,2,4)

  a

  b <- matrix(1:8*0.1,2,4)

  b
  ```

```
a + b
```

Run: `R --no-save --quiet < demo-startup.R > demo-startup.Rout`

It generates a file named "demo-startup.Rout", shown as follows:

```
> a <- matrix(1:8,2,4)
>
> a
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
>
> b <- matrix(1:8*0.1,2,4)
>
> b
     [,1] [,2] [,3] [,4]
[1,]  0.1  0.3  0.5  0.7
[2,]  0.2  0.4  0.6  0.8
>
> a + b
     [,1] [,2] [,3] [,4]
[1,]  1.1  3.3  5.5  7.7
[2,]  2.2  4.4  6.6  8.8
>
```

- Alternatively, one can type the following commands in R console for running all commands in the file "demo-startup.R":
  `source("demo-startup.R",echo=TRUE).`
  The above content in the file "demo-startup.Rout" will be printed on screen.
- `R CMD BATCH file.R` is similar to
  `R --save < file.R > file.Rout`, but writting all messages printed on screen in "file.Rout", including the error messages.
- Run programs in the background using `nohup`, eg:
  `nohup R BATCH infile &`

2.1.2 From Windows

- Using Windows command lines: adding the directory containing R.exe to the environment variable `path`. You can run R in the same way as using Unix terminal. But `nohup` facility may not exist.
- Click R on the desktop to start up. You need to change the working directory for the first time. After you save image (file ".RData" will be created), for the second time, click the file ".RData" will open R from this directory.

2.2 Getting help

- From R console, type `?keyword`, or `help.search(keyword)`
- From internet: http://cran.r-project.org

2.3 Objects and operations

2.3.1 Numbers and vector

Below I use example to demonstrate how to use numeric vector.

```
> a <- 1
>
> a
[1] 1
>
> b <- 2:10
>
> b
[1]  2  3  4  5  6  7  8  9 10
>
> #concatenate two vectors
> c <- c(a,b)
>
> c
 [1]  1  2  3  4  5  6  7  8  9 10
>
> #vector arithmetics
> 1/c
 [1] 1.0000000 0.5000000 0.3333333 0.2500000 0.2000000 0.1666667 0.1428571
 [8] 0.1250000 0.1111111 0.1000000
>
> c^2
 [1]   1   4   9  16  25  36  49  64  81 100
>
> c^2 + 1
 [1]   2   5  10  17  26  37  50  65  82 101
>
> #apply a function to each element
> log(c)
 [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
 [8] 2.0794415 2.1972246 2.3025851
>
> sapply(c,log)
```

```
 [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
 [8] 2.0794415 2.1972246 2.3025851
>
>
> #operation on two vectors
> d <- (1:10)*10
>
> d
 [1]  10  20  30  40  50  60  70  80  90 100
>
> c + d
 [1]  11  22  33  44  55  66  77  88  99 110
>
> c * d
 [1]   10   40   90  160  250  360  490  640  810 1000
>
> d ^ c
 [1] 1.000000e+01 4.000000e+02 2.700000e+04 2.560000e+06 3.125000e+08
 [6] 4.665600e+10 8.235430e+12 1.677722e+15 3.874205e+17 1.000000e+20
>
> #more concrete example: computing variance of 'c'
> sum((c - mean(c))^2)/(length(c)-1)
[1] 9.166667
>
> #of course, there is build-in function for computing variance:
> var(c)
[1] 9.166667
>
> #subsetting vector
> c
 [1]  1  2  3  4  5  6  7  8  9 10
>
> c[2]
[1] 2
>
> c[c(2,3)]
[1] 2 3
>
> c[c(3,2)]
[1] 3 2
>
> c[c > 5]
[1]  6  7  8  9 10
>
> #let's see what is "c > 5"
> c > 5
```

```
 [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
>
> c[c > 5 & c < 10]
[1] 6 7 8 9
>
> c[as.logical((c > 8) + (c < 3))]
[1]  1  2  9 10
>
> log(c)
 [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
 [8] 2.0794415 2.1972246 2.3025851
>
> c[log(c) < 2]
[1] 1 2 3 4 5 6 7
>
> #modifying subset of vector
> c[log(c) < 2] <- 3
>
> c
 [1]  3  3  3  3  3  3  3  8  9 10
>
> #extending and cutting vector
> length(c) <- 20
>
> c
 [1]  3  3  3  3  3  3  3  8  9 10 NA NA NA NA NA NA NA NA NA NA
>
> c[25] <- 1
>
> c
 [1]  3  3  3  3  3  3  3  8  9 10 NA NA NA NA NA NA NA NA NA NA NA NA NA NA  1
>
> #getting back original vector
> length(c) <- 10
> c
 [1]  3  3  3  3  3  3  3  8  9 10
>
> #introduce a function ‘‘seq’’
> seq(0,10,by=1)
 [1]  0  1  2  3  4  5  6  7  8  9 10
>
> seq(0,10,length=20)
 [1]  0.0000000  0.5263158  1.0526316  1.5789474  2.1052632  2.6315789
 [7]  3.1578947  3.6842105  4.2105263  4.7368421  5.2631579  5.7894737
[13]  6.3157895  6.8421053  7.3684211  7.8947368  8.4210526  8.9473684
[19]  9.4736842 10.0000000
```

```
>
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
>
> #seq is more reliable than ":"
> n <- 0
>
> 1:n
[1] 1 0
>
> #seq(1,n,by=1)
> #Error in seq.default(1, n, by = 1) : wrong sign in 'by' argument
> #Execution halted
>
> #function ``rep''
> c<- 1:5
>
> c
[1] 1 2 3 4 5
>
> rep(c,5)
 [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
>
> rep(c,each=5)
 [1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3 4 4 4 4 4 5 5 5 5 5
>
```

2.3.2 Missing values

```
> a <- 0/0
>
> a
[1] NaN
>
> is.nan(a)
[1] TRUE
>
> b <- log(0)
>
> b
[1] -Inf
>
> is.finite(b)
[1] FALSE
>
```

```
> c <- c(0:4,NA)
>
> c
[1]  0  1  2  3  4 NA
>
> is.na(c)
[1] FALSE FALSE FALSE FALSE FALSE  TRUE
>
```

2.3.3 Character vectors

Character strings are entered using either double (") or single (') quotes, but are printed using double quotes (or sometimes without quotes). They use C-style escape sequences, using \ as the escape character, so \\ is entered and printed as \\, and inside double quotes " is entered as \". Other useful escape sequences are \n, newline, \t, tab and \b, backspace.

```
> A <- c("a","b","c")
>
> A
[1] "a" "b" "c"
>
> paste("a","b",sep="")
[1] "ab"
>
> paste(A,c("d","e"))
[1] "a d" "b e" "c d"
>
> paste(A,10)
[1] "a 10" "b 10" "c 10"
>
> paste(A,10,sep="")
[1] "a10" "b10" "c10"
>
> paste(A,1:10,sep="")
 [1] "a1"  "b2"  "c3"  "a4"  "b5"  "c6"  "a7"  "b8"  "c9"  "a10"
>
```

2.3.4 Matrice

```
> A <- matrix(0,4,5)
>
> A
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    0    0    0    0
```

```
[2,]    0    0    0    0    0
[3,]    0    0    0    0    0
[4,]    0    0    0    0    0
>
> A <- matrix(1:20,4,5)
>
> A
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
>
>
> #subsectioning and modifying subsection
>
> A[c(1,4),c(2,3)]
     [,1] [,2]
[1,]    5    9
[2,]    8   12
>
> A[c(1,4),c(2,3)] <- 1
>
> A
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    1   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    1    1   16   20
>
> A[4,]
[1]  4  1  1 16 20
>
> A[4,,drop = FALSE]
     [,1] [,2] [,3] [,4] [,5]
[1,]    4    1    1   16   20
>
> #combining two matrices
>
> #create another matrix using another way
> A2 <- array(1:20,dim=c(4,5))
>
> A2
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
```

```
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
>
> cbind(A,A2)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    1    1   13   17    1    5    9   13    17
[2,]    2    6   10   14   18    2    6   10   14    18
[3,]    3    7   11   15   19    3    7   11   15    19
[4,]    4    1    1   16   20    4    8   12   16    20
>
> rbind(A,A2)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    1   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    1    1   16   20
[5,]    1    5    9   13   17
[6,]    2    6   10   14   18
[7,]    3    7   11   15   19
[8,]    4    8   12   16   20
>
> #operating matrice
>
> #transpose matrix
> t(A)
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    1    6    7    1
[3,]    1   10   11    1
[4,]   13   14   15   16
[5,]   17   18   19   20
>
> A
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    1   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    1    1   16   20
>
> A + 1
     [,1] [,2] [,3] [,4] [,5]
[1,]    2    2    2   14   18
[2,]    3    7   11   15   19
[3,]    4    8   12   16   20
[4,]    5    2    2   17   21
>
```

```
> x <- 1:4
>
> A*x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    1   13   17
[2,]    4   12   20   28   36
[3,]    9   21   33   45   57
[4,]   16    4    4   64   80
>
> #the logical here is coercing the matrix "A" into a vector by joining the column
> #and repeat the shorter vector,x, as many times as making it have the same
> #length as the vector coerced from "A"
>
> #see another example
>
> x <- 1:3
>
> A*x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3   13   34
[2,]    4   18   10   28   54
[3,]    9    7   22   45   19
[4,]    4    2    3   16   40
>
> A^2
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    1  169  289
[2,]    4   36  100  196  324
[3,]    9   49  121  225  361
[4,]   16    1    1  256  400
>
> A <- matrix(sample(1:20),4,5)
>
> A
     [,1] [,2] [,3] [,4] [,5]
[1,]    7    1    9    3   20
[2,]   11   10   12   15    4
[3,]    6   16    5    8   19
[4,]    2   17   14   13   18
>
> B <- matrix(sample(1:20),5,4)
>
> B
     [,1] [,2] [,3] [,4]
[1,]   11   13    3   16
[2,]    2    4    7   10
```

```
[3,]    9   12    1   14
[4,]    5   15   17    8
[5,]   19    6   20   18
>
> C <- A %*% B
>
> C
     [,1] [,2] [,3] [,4]
[1,]  555  368  488  632
[2,]  400  576  450  636
[3,]  544  436  651  732
[4,]  589  565  720  826
>
> solve(C)
              [,1]         [,2]        [,3]        [,4]
[1,]  0.0113721434 -0.011305781 -0.03593289  0.03184764
[2,]  0.0037070714 -0.006309519 -0.03288573  0.03116506
[3,] -0.0008241436 -0.012304505 -0.02598824  0.03313549
[4,] -0.0099265186  0.023103180  0.07077051 -0.07169985
>
> #solving linear equation
>
> x <- 1:4
>
> d <- C %*% x
>
> solve(C,d)
     [,1]
[1,]    1
[2,]    2
[3,]    3
[4,]    4
>
> #altenative way (but not recommended)
> solve(C) %*% d
     [,1]
[1,]    1
[2,]    2
[3,]    3
[4,]    4
>
> #SVD (C = UDV') and determinant
>
> svd.C <- svd(C)
>
> svd.C
```

```
$d
[1] 2332.552515  204.076932   98.799790    7.614026

$u
           [,1]        [,2]       [,3]       [,4]
[1,] -0.4430091  0.41432949  0.7886463  0.1005537
[2,] -0.4432302 -0.84065092  0.2206557 -0.2194631
[3,] -0.5143024  0.34835579 -0.3848769 -0.6826500
[4,] -0.5854766  0.01689212 -0.4256969  0.6897202

$v
           [,1]        [,2]        [,3]       [,4]
[1,] -0.4492025  0.45643331  0.66652400  0.3816170
[2,] -0.4172931 -0.83456029  0.09103668  0.3479769
[3,] -0.5024522  0.30793204 -0.73787822  0.3290219
[4,] -0.6096108 -0.01885767  0.05471573 -0.7905854

>
> #calculating determinant of C
>
> prod(svd.C$d)
[1] 358092916
>
>
```

2.3.5 List

An R list is an object consisting of an ordered collection of objects known as its components.

There is no particular need for the components to be of the same mode or type, and, for example, a list could consist of a numeric vector, a logical value, a matrix, a complex vector, a character array, a function, and so on.

```
> a <- 1:10
>
> b <- matrix(1:10,2,5)
>
> c <- c("name1","name2")
>
> alst <- list(a=a,b=b,c=c)
>
> alst
$a
 [1]  1  2  3  4  5  6  7  8  9 10
```

```
$b
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10

$c
[1] "name1" "name2"

>
> #refering to component of a list
>
> alst$a
 [1]  1  2  3  4  5  6  7  8  9 10
>
> alst[[2]]
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
>
> blst <- list(d=2:10*10)
>
> #concatenating list
> ablst <- c(alst,blst)
>
> ablst
$a
 [1]  1  2  3  4  5  6  7  8  9 10

$b
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10

$c
[1] "name1" "name2"

$d
[1]  20  30  40  50  60  70  80  90 100

>
```

A list is usually used to return the results of a large program, for example those of a linear regression fitting.

2.3.6 Data frames

A data frame may for many purposes be regarded as a matrix with columns possibly of differing modes and attributes. It may be displayed in matrix form, and its rows and columns extracted using matrix indexing conventions.

```
> name <- c("john","peter","jennifer")
>
> gender <- factor(c("m","m","f"))
>
> hw1 <- c(60,60,80)
>
> hw2 <- c(40,50,30)
>
> grades <- data.frame(name,gender,hw1,hw2)
>
>
> grades
      name gender hw1 hw2
1     john      m  60  40
2    peter      m  60  50
3 jennifer      f  80  30
>
> #subsectioning a data frame
>
> grades[1,2]
[1] m
Levels: f m
>
> grades[,"name"]
[1] john     peter    jennifer
Levels: jennifer john peter
>
> grades$name
[1] john     peter    jennifer
Levels: jennifer john peter
>
> grades[grades$gender=="m",]
   name gender hw1 hw2
1  john      m  60  40
2 peter      m  60  50
>
> grades[,"hw1"]
[1] 60 60 80
>
> #divide the subjects by "gender", and calculating means in each group
```

```
> tapply(grades[,"hw1"], grades[,"gender"],mean)
 f  m
80 60
>
>
```

2.3.7 Reading data from external files

File "grades" is shown as follows:

```
      name gender hw1 hw2
1     john      m  60  40
2    peter      m  60  50
3 jennifer      f  80  30
```

Creating a data frame `grades`:

```
> grades <- read.table("grades")
> grades
      name gender hw1 hw2
1     john      m  60  40
2    peter      m  60  50
3 jennifer      f  80  30
```

Other functions that read data from external files of different format: `read.csv,read.delim`. They are specific form of `read.table`. For more details, type `?read.table`.

2.4 Writing your own functions

In file "demo-fun.R", a function is defined:

```
#looking for the maximum value of a numeric vector x
find.max <- function(x)
{
   n <- length(x)

   x.m <- x[1]
   ix.m <- 1

   if(n > 1)
   {
      for( i in seq(2,n,by=1) )
      {
         if(x[i] > x.m)
         {
            x.m <- x[i]
            ix.m <- i
         }
      }
   }
   #return the maximum value and the index
   list(max=x.m,index.max=ix.m)
}
```

We now can use this function:

```
> #sourcing functions in file "demo-fun.R"
> source("demo-fun.R")
>
> x <- runif(12)
>
> x
 [1] 0.06779191 0.09266746 0.63309784 0.85986312 0.81900862 0.80315468
 [7] 0.71691262 0.35424083 0.59253821 0.23433190 0.60891787 0.35025762
>
> #calling "find.max"
> find.max(x)
$max
[1] 0.8598631

$index.max
[1] 4
```

```
>
```

2.5 Making graphics with R

2.5.1 Drawing plots on screen

Typing plotting commands, plots will be shown in a seperate window.

2.5.2 Making graphics in a file

In order to demonstrate how to use R to produce plots and save plots in a file, I use R to draw plots to illustrate the following two functions:

$$f_1(x) = \frac{a_0 + a_1\, x + a_2\, x^2 + \exp(x)}{x^2} \tag{2.1}$$

$$f_2(x) = \frac{a_0 + a_1\, (10 - x) + a_2\, (10 - x)^2 + \exp(10 - x)}{(10 - x)^2} \tag{2.2}$$

where $a_0 = 1, a_1 = 2$ and $a_2 = 3$.

The R commands are shown as follows:

```
demofun1 <- function(x)
{
   ( 1 + 2*x^2 + 3*x^3 + exp(x) ) / x^2
}

demofun2 <- function(x)
{
   ( 1 + 2*(10-x)^2 + 3*(10-x)^3 + exp(10-x) ) / (10-x)^2
}

#open a file to draw in it
postscript("fig-latexdemo.eps", paper="special",
          height=4.8, width=10, horizontal=FALSE)

#specify plotting parameters
par(mfrow=c(1,2), mar = c(4,4,3,1))

x <- seq(0,10,by=0.1)
```

```
#make "Plot 1"
plot(x, demofun1(x), type="p", pch = 1,
     ylab="y", main="Plot 1")

#add another line to "Plot 1"
points(x, demofun2(x), type="l",  lty = 1)

#make "plot 2"
plot(x, demofun1(x), type="b", pch = 3, lty=1 ,
     ylab="y", main="Plot 2")

#add another line to "Plot 2"
points(x, demofun2(x), type="b", pch = 4, lty = 2)

dev.off()
```

The file produced contains the resulting plots:

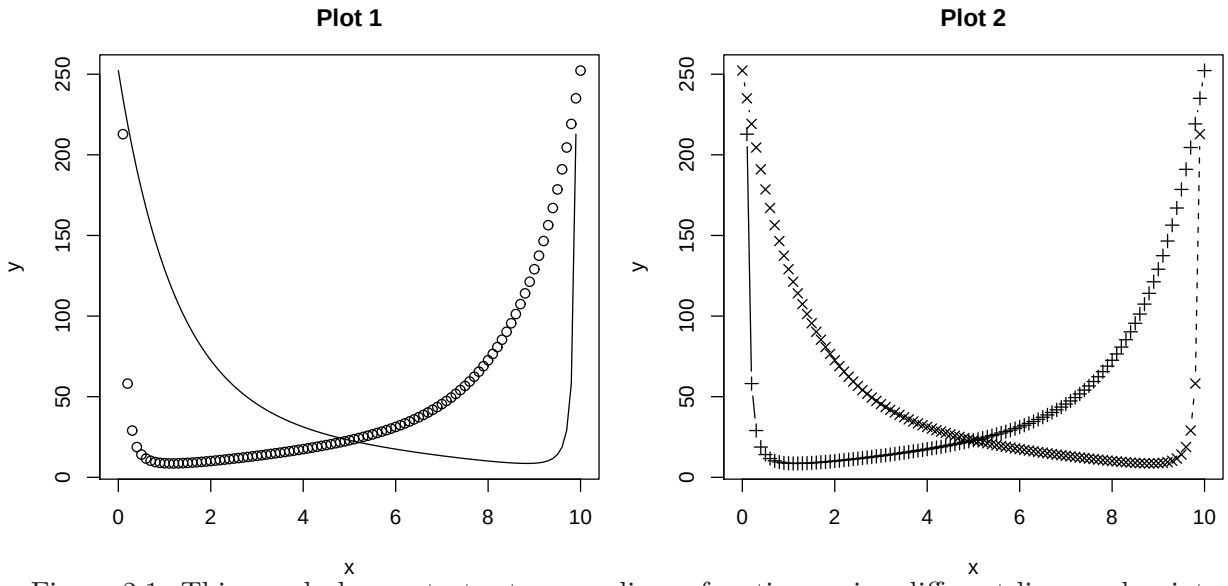


Figure 2.1: This graph demonstrates two non-linear functions using different lines and points

2.6 Installing new packages and loading new packages

2.6.1 Installing a new package from source files

- download a source file, say `apacakge.tar.gz`.
- Run

  `R CMD INSTALL apackage.tar.gz -l /my/rlibrary`

  to compile the source file and install it to directory `/my/rlibrary`.

You must have GCC compiler installed, which is not available for windows system without doing much effort. The package needs not to be available from CRAN.

2.6.2 Installing a new package from CRAN

If the package is available from CRAN, you can install a new package without compilation. The precompiled package can be downloaded. To do this, type

`install.packages("apackage",lib="/my/rlibrary")`.

2.6.3 Loading a package

After installing new package `apackage`, type

`library("apackage",lib.loc="/my/library")`.

Then you can use the functionalities provided by this package.

2.7 Calling C functions (for unix system only)

R is slow. We better use C codes to do intensive computations. Pointers of R vectors (matrice will be coerced to vectors) are passed to C codes such that C program can use them. This is realized by function ".C".

Below is a simple demonstration.

File "sum.c" is shown as follows:

```
void newsum(int la[1], double a[], double s[1])
{
    int i;
    s[0] = 0;
    for(i=0;i<la[0];i++)
       s[0] += a[i];
}
```

Then compile the C program above:

`R CMD SHLIB sum.c`

.

Two files, `sum.o` and `sum.so`, will be produced.

Now we can load "sum.so" into R environment and call the C function with .C:

```
> dyn.load("sum.so")
>
> a <- c(1,2,3,4)
>
> .C("newsum",length(a),a,sum=0)
[[1]]
[1] 4

[[2]]
[1] 1 2 3 4

$sum
[1] 10

>
```

Typically, people write a "wrapper" function to ease calling C functions regularly in R, for example:

```
newsum <- function(a)
{
    .C("newsum",length(a),a,sum=0)$sum
}
```

2.8 Creating your own package and submitting to CRAN

It is for advanced users. However, it is not difficult. Following this procedure:

- Create a directory "apackage", which has 3 directories: R (containing all R sources), src (containing all C or Fortran files), man (documentation files for each R function), and a file "DESCRIPTION". There are required formats for writing documentation files and "DESCRIPTION".
- (suggested) Run `R CMD check apackage` to check the package and make suggested changes if some come up. A package accepted by CRAN must have no warnings and no errors in this step.
- Run `R CMD build apackage`

To submit to CRAN, just post it (with anonymous ftp) to ftp://cran.r-project.org/incoming/. Somebody will run your source file and if it passes `R CMD check` without warnings and errors, it will be posted online as CRAN contributed packages. Mac and Windows binaries will be created.

Chapter 3

Computer arithmetic

3.1 Integers in computer

Integers in a 32-bit computer is represented as vectors $(x_0, \ldots, x_{31})$, through this model (but it might not be used exactly by a particular computer):

$$u = \sum_{i=0}^{31} x_i \, 2^i - 2^{31} \tag{3.1}$$

In theory:

Largest integer : $(x_0, \ldots, x_{31}) = (1, \ldots, 1)$: $\sum_{i=0}^{31} 1 \times 2^i - 2^{31} = 2^{32} - 1 - 2^{31} = 2^{31} - 1$,

Smallest integer : $(x_0, \ldots, x_{31}) = (0, \ldots, 0)$: $\sum_{i=0}^{31} 0 \times 2^i - 2^{31} = -2^{31}$,

In R, due to implementation reason, the range of an integer u is:

$$-(2^{31} - 1) \leq u \leq 2^{31} - 1 \tag{3.2}$$

Or written as:

$$-2^{31} < u < 2^{31} \tag{3.3}$$

Let's test on R:

```
> options(digits=20)
>
> #seeing largest and smallest integers
```

```
>
> as.integer(2^31)
[1] NA
>
> as.integer(2^31 - 1)
[1] 2147483647
>
> as.integer(2^31 - 1) + as.integer(1)
[1] NA
>
> as.integer(-(2^31 - 1))
[1] -2147483647
>
> as.integer(- 2^31 )
[1] NA
>
> as.integer(- 2^31 -1 )
[1] NA
>
>
```

We see that $(2^{31}-1)+1 = \text{NA}$. R is smart in detecting this overflow. But not always for all programing language. In compiled language, like C,

$$(2^{31}-1)+1 = -2^{31}$$

Why?

$$\begin{array}{lll} & (\quad 1 \quad 1 \quad ..... \quad 1 \quad) & = 2^{31}-1 \\ +1 & & \\ \hline & (\quad 0 \quad 0 \quad ..... \quad 0 \quad 1 \quad) & \\ \text{round-off} & & \\ \hline & (\quad 0 \quad 0 \quad ..... \quad 0 \quad) & = 0-2^{31} \end{array}$$

3.2 Floating-point numbers

A real value is represented with a vector

$$(s, d_0, \ldots, d_{t-1}, e_0, \ldots, e_{k-1}),$$

by the following model:

$$f = (-1)^s \times \sum_{i=0}^{t-1} d_i \, 2^{-i} \times 2^{\sum_{i=0}^{k-1} e_i \, 2^i - 2^{11}} \tag{3.4}$$

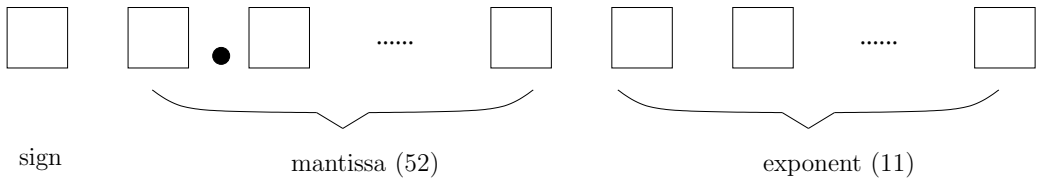
$$= (-1)^s \times (d_0.d_1 \cdots d_{t-1}) \times 2^{e_{k-1} \cdots e_0 - 2^{11}} \tag{3.5}$$

s is sign, $(d_0, \ldots, d_{t-1})$ is fraction, namely mantissa, $(e_0, \ldots, e_{k-1})$ is exponent in integer, t is the number of significant numbers in mantissa, called *precision*.

We can move the decimal point and add/minus exponent to represent the same number: eg.

$$0.011 \times 2^0 = 0.110 \times 2^{-1}$$

A standard double precision representation uses a sign bit, an 11 bit exponent, and 52 bits for the mantissa.



The largest double precision floating-point number:

$$1.11 \cdots 1 \times 2^{2^{10}-1} = \sum_{i=0}^{51} 2^{-i} \times 2^{2^{10}-1} \approx 2^{1024}$$

The smallest double precision floating-point number in magnitude greater than 0:

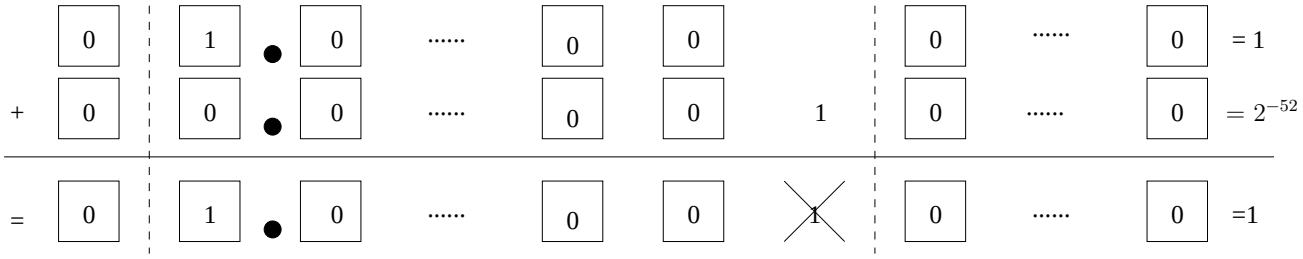
$$0.00 \cdots 01 \times 2^{-2^{10}+1} = 2^{-51} \times 2^{2^{10}-1} = 2^{-1074}$$

Let's check in R:

```
> options(digits=22)
>
> #largest double precision floating-point number:
> sum(2^(-(0:50)))*2^(2^10-1)
[1] 1.797693134862315e+308
>
> sum(2^(-(0:51)))*2^(2^10-1)
```

```
[1] 1.797693134862316e+308
>
> sum(2^(-(0:51)))*2^(2^10-1) + 1
[1] 1.797693134862316e+308
>
> sum(2^(-(0:51)))*2^(2^10-1) * 2
[1] Inf
>
>
> #smallest double precision floating-point number:
> 2^(-1073)
[1] 9.88131291682493e-324
>
> 2^(-1074)
[1] 4.940656458412465e-324
>
> 2^(-1075)
[1] 0
>
```

Computer will move floating-point number to nearest one if there are not sufficient digits to represent it, resulting so-called round-off error. The following graph shows how this occurs:



Therefore $1+2^{-52} = 1$. We can see that the number of t determines how precise a floating-point number can be.

Note: The representation for floating-point numbers described above is only one possibility, and may or may not be used on any particular computer. However, any finite binary representation of floating-point numbers will have restrictions on the accuracy of representations.

Let's check in R (Do not know why. the smallest number that can be added to 1 without error is 2^{-52}, not 2^{-51}.):

```
> options(digits=22)
>
> f1 <- 1
> i <- 1
> while(i <= 2^12){
+    f1 <- f1 + 2^(-52)
```

```
+    i <- i + 1
+ }
>
> f1
[1] 1.0000000000009095
>
> #If no round-off error, f should be equal to
>
> f2 <- 1 + 2^(-40)
>
> #If no round-off error, the following f should also be equal to 1 + 2^(-40)
>
> f3 <- 1
> i <- 1
> while(i <= 2^13){
+    f3 <- f3 + 2^(-53)
+    i <- i + 1
+ }
>
> f3
[1] 1
>
> f1==f2
[1] TRUE
>
> f2==f3
[1] FALSE
>
>
> #let's remember the number of significant number in different bases
>
> #base= exp(1) = 2.71
> log(2^(-52))
[1] -36.04365338911715
>
> #based 10
> log(2^(-52),base=10)
[1] -15.65355977452702
>
```

3.3 Examples of improving numerical accuracy

Avoiding computing $\frac{\texttt{Inf}}{\texttt{Inf}}$ and $\frac{0}{0}$.

Example: The following link between p and θ is used in logistic regression, where p is the probability of $y = 1$, and θ is linear function of predictor variables:

$$p = \frac{\exp(\theta)}{1 + \exp(\theta)} \tag{3.6}$$

There are two ways to compute p given θ:

$$\text{M1:} \quad p = \frac{\exp(\theta)}{1 + \exp(\theta)} \tag{3.7}$$

$$\text{M2:} \quad p = \frac{1}{1 + \exp(-\theta)} \tag{3.8}$$

M2 is safer than M1. Why?

- When θ is positively huge, M1 will result in Inf/Inf = NaN. In contrast, M2 will give $1/(1+0) = 1$.
- When θ is negatively huge, both methods give 0.

Let's check in R:

```
> p1 <- function(theta)
+ {
+    exp(theta)/(1+exp(theta))
+ }
>
> p2 <- function(theta)
+ {
+    1/(1+exp(-theta))
+ }
>
>
> theta <- 2000
>
> p1(theta)
[1] NaN
>
> p2(theta)
[1] 1
>
```

```
> theta <- -2000
>
> p1(theta)
[1] 0
>
> p2(theta)
[1] 0
>
```

Avoiding `large-large`

Example: Computing $\exp(x)$ using Taylor expansion:

$$e^x = \sum_{i=0}^{+\infty} \frac{x^i}{i!} \tag{3.9}$$

Below is a straightforward but unstable algorithm:

```
> fexp <- function(x)
+ {
+    i <- 0
+    expx <- 1
+    u <- 1
+    while(abs(u)>1e-8*abs(expx)) {
+      i <- i+1
+      u <- u*x/i
+      expx <- expx+u
+    }
+    expx
+ }
>
> c(exp(10),fexp(10))
[1] 22026.47 22026.47
>
> c(exp(20),fexp(20))
[1] 485165195 485165193
>
> c(exp(60),fexp(60))
[1] 1.142007e+26 1.142007e+26
>
> c(exp(-1),fexp(-1))
[1] 0.3678794 0.3678794
>
> c(exp(-10),fexp(-10))
```

```
[1] 4.539993e-05 4.539993e-05
>
> c(exp(-20),fexp(-20))
[1] 2.061154e-09 5.621884e-09
>
> c(exp(-30),fexp(-30))
[1]  9.357623e-14 -3.066812e-05
>
```

We see that `fexp` works well for positive large value but poorly for negative large value.

Why? If x_0 is positively large,

$$e^{-x_0} = \sum_{i=0}^{+\infty} (-1)^i \frac{x_0^i}{i!} \tag{3.10}$$

The sign alternates and $x_0^i/i!$ is large. A better way is to compute $\exp(-x_0) = 1/\exp(x_0)$:

```
> fexp <- function(x) {
+    xa <- abs(x)
+    i <- 0
+    expx <- 1
+    u <- 1
+    while(u>1.e-8*expx) {
+      i <- i+1
+      u <- u*xa/i
+      expx <- expx+u
+    }
+    if (x >= 0) expx else 1/expx
+ }
>
>
> c(exp(10),fexp(10))
[1] 22026.47 22026.47
>
> c(exp(20),fexp(20))
[1] 485165195 485165193
>
> c(exp(60),fexp(60))
[1] 1.142007e+26 1.142007e+26
>
> c(exp(-1),fexp(-1))
[1] 0.3678794 0.3678794
>
> c(exp(-10),fexp(-10))
```

```
[1] 4.539993e-05 4.539993e-05
>
> c(exp(-20),fexp(-20))
[1] 2.061154e-09 2.061154e-09
>
> c(exp(-30),fexp(-30))
[1] 9.357623e-14 9.357623e-14
>
```

Example: Computing variance. There are two ways. One is to use this formula:

$$Var_1(x) = (\sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n} x_i)^2/n)/(n-1) \tag{3.11}$$

The other is to use:

$$Var_2(x) = (\sum_{i=1}^{n} (x_i - \bar{x})^2)/(n-1) \tag{3.12}$$

```
> var1 <- function(x)
+ { n <- length(x)
+  ( sum(x^2) - sum(x)^2/n ) / (n-1)
+ }
>
>
> var2 <- function(x)
+ {
+   n <- length(x)
+
+   sum((x-mean(x))^2) / (n-1)
+ }
>
> x <- c(1,2,3)
>
> c(var1(x),var2(x),var(x))
[1] 1 1 1
>
>
> x <- c(1,2,3) + 1e10
>
> c(var1(x),var2(x),var(x))
[1] -32768      1      1
>
```

Example: Computing inner product

$$< x, y >= \sum_{i=1}^{n} x_i \, y_i.$$

The signs of $x_i \, y_i$ may be negative and $|x_i \, y_i|$ may be large. Inner product is used in performing matrix multiplication, therefore needed in many straightforward implementation of statistical methods, for example in the least square estimator: $(X'X)^{-1}X'Y$, and in computing density function of multivariate normal distribution: $x'\Sigma^{-1}x$. A better way is centralizing x and y first:

$$< x, y >= \sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y}) + \frac{\sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{n}.$$

Representing numbers in logarithm

In the intermediate steps of computing a quantity, we may express the intermediate numbers in logarithm to avoid overflow and underflow. For example, we know that $\frac{2^{2000}}{2^{2001}} = \frac{1}{2}$. However, both numbers under and over the lines are overflow. If we express them in logarithm, 2000 and 2001, they are both representable, and the final result can be calculated by $2^{2000-2001} = \frac{1}{2}$. It is similar when both number under and over lines are underflow.

It is trivial to compute the logarithm of multiplication of a sequence of numbers. The following is a way to compute the logarithm of a sum of a sequence numbers expressed in their logarithm. Suppose we want to compute $\sum_{i=1}^{n} x_i$. We store x_i in its logarithm: logx_i. Let $m_{logx} = \max(\text{logx}_1, \ldots, \text{logx}_n)$. The logarithm of the sum is computed as follows:

$$\log(\sum_{I=1}^{n} x_i) = m_{logx} + \log(\sum_{i=1}^{n} \exp(\text{logx}_i - m_{logx})) \tag{3.13}$$

Using (3.13), even when each x_i is underflow or overflow, as long as $\exp(\text{logx}_i - m_{logx})$ can be added to 1, we can still represent the logarithm of the sum correctly. When $\exp(\text{logx}_i - m_{logx})$ is very small, we only omit some very small terms, making little difference. By this way, we can compute the logarithm of the sum of numbers in a much wider range.

Similarly we can compute the logarithm of the difference of two numbers. Let $\log_x = \log(x)$, and $\log_y = \log(y)$, and assume $x > y$. We compute $\log(x - y)$ as follows:

$$\log(x - y) = \log_x + \log(1 - \exp(\log_y - \log_x)) \tag{3.14}$$

Let's demonstrate in R:

```
> options(digits=22)
>
```

```
> #ratio of two large numbers
>
> log_x <- 800
>
> x <- exp(log_x)
>
> x
[1] Inf
>
> log_y <- 805
>
> y <- exp(log_y)
>
> y
[1] Inf
>
> x/y
[1] NaN
>
> log_xovery <- log_x- log_y
>
> log_xovery
[1] -5
>
> exp(log_xovery)
[1] 0.006737946999085467
>
>
> #looking for log of sum of numbers in logorithm
>
> log_sum_exp <- function(log_x)
+ {
+    max_log_x <- max(log_x)
+
+    max_log_x + log( sum(exp(log_x - max_log_x) ))
+ }
>
> log_x <- c(2000,2010,2030)
>
> exp(log_x)
[1] Inf Inf Inf
>
> log_sum_exp(log_x)
[1] 2030.000000002061
>
> log_x <- - c(2000,2010,2030)
```

```
>
> exp(log_x)
[1] 0 0 0
>
> log_sum_exp(log_x)
[1] -1999.9999546011
>
> log_minus_exp <- function(log_x,log_y)
+
+ {  if(log_x < log_y)
+      stop("The first argument is bigger than the second")
+
+    log_x + log(1-exp(log_y-log_x))
+
+ }
>
> log_minus_exp(2020,2000)
[1] 2019.999999997939
>
>
> exp(2000) - exp(1999)
[1] NaN
>
```

Chapter 4

Generating Pseudo-random numbers

4.1 Generating unif(0,1) random numbers

The most commonly used unif(0,1) random number generator is multiplicative congruential generators, which generate a sequence of integers $\{a_i\}$ by calculating:

$$a_{i+1} = (A \times a_i) \bmod M \tag{4.1}$$

with carefully chosen positive integers A and M. The choice of A and M is crucial to the quality of pseudo-random numbers. Since it is not relevant to most applications in statistics, we do not want to go into many details. For your knowledge, the common choice of M is $2^{31} - 1$, the largest integer in 32-bit computer, and the common multiplier A is 7^5.

In R, a function is available for generating unif(0,1) random number: `runif`.

The following graph shows the histogram of 100000 random numbers generated by `runif`:

The above graph is produced by the following R commands:

```
n <- 100000

a <- runif(n)

postscript(
"test-runif.eps",paper="special",height=3.5,width=6.5,horizontal=FALSE)

par(mfrow=c(1,2),mar=c(4,4,3,1))

hist(a,,xlab="Random Numbers",main="")

acf(a,main="")
```

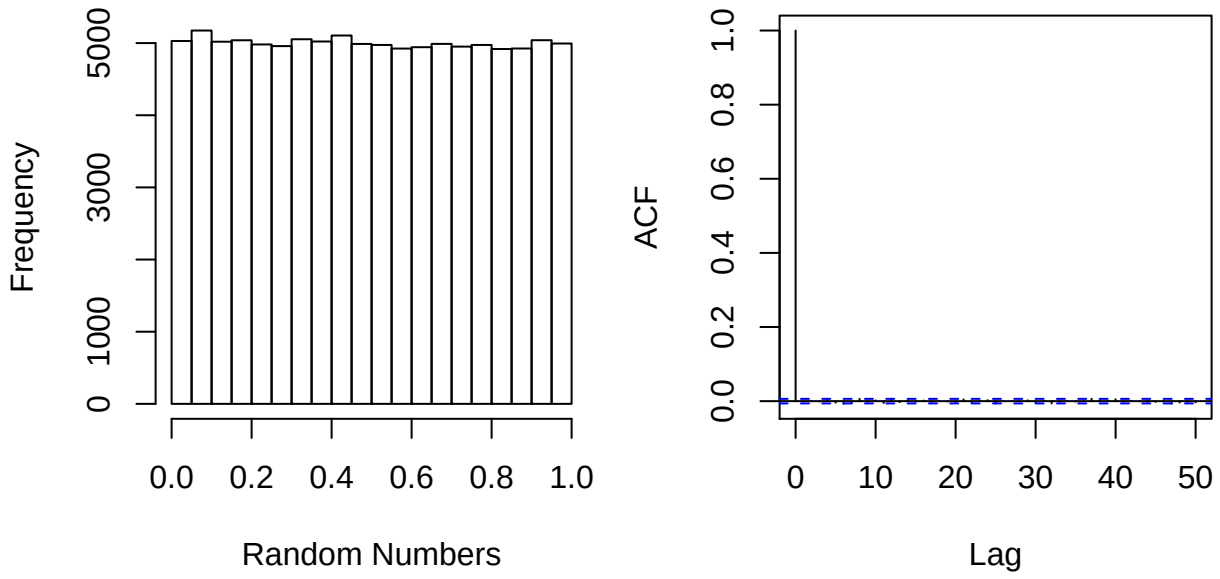


Figure 4.1: Histogram and ACF of 100000 random numbers

```
dev.off()
```

4.2 Transformation methods

Once we have unif(0,1) random numbers, we can generate random numbers for many distribution through transformation.

Inverting CDF

This method is based on this fact: If F is the CDF of a distribution, and U has a distribution unif(0,1), then

$$F^{-1}(U) \sim F \tag{4.2}$$

where $F^{-1}(u) = \inf\{x : F(x) \geq u\}$.

This method is illustrated by the following graph:

Example: Generate random numbers from exp(1).

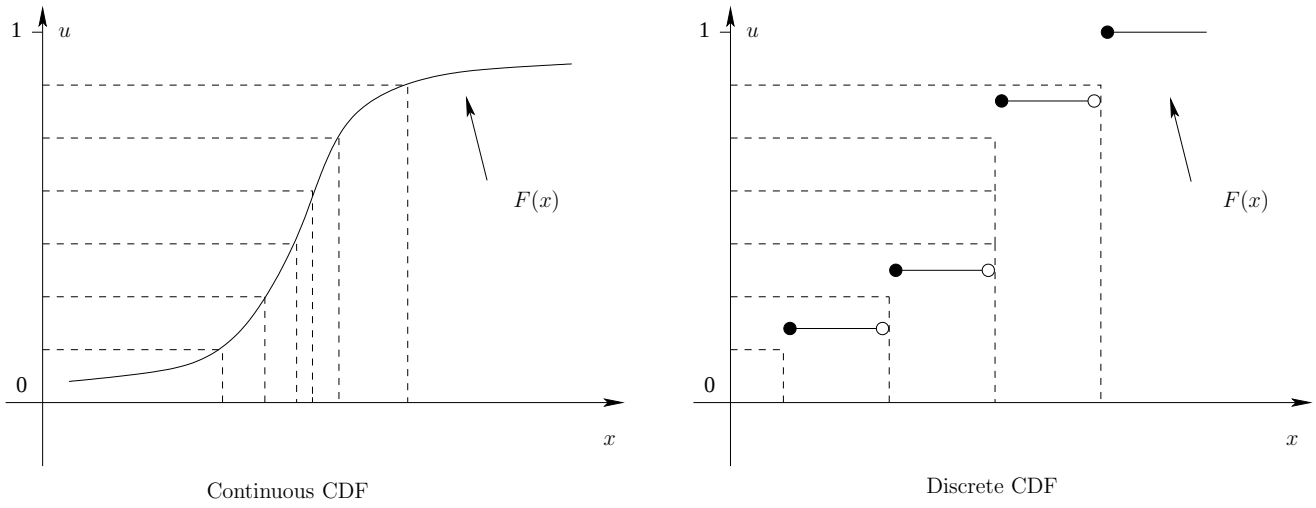


Figure 4.2: Illustration o the Distribution of $F^{-1}(U)$

The CDF of exp(1) is:

$$F(x) = 1 - \exp(-x), \quad \text{for } x > 0. \tag{4.3}$$

The inverse of $F(x)$ above is:

$$F^{-1}(u) = -\log(1-u) \tag{4.4}$$

Below is an R function for generating exp(1):

```
#use method of inverse cdf to generate iid sample from exp(1)
gen_exp <- function(n)
{
   #generate unif(0,) random numbers
   u <- runif(n)
   #transform the random numbers
   -log(1-u)
}
```

Question: how to generate exp(λ) random numbers?

We need to compute CDF and inverse CDF, which is not necessarily doable in an exact way. For example, the CDF and inverse CDF of normal distributions do not have close form. We often need some approximate methods. For example, if we can not compute the inverse CDF directly, we need to use numerical methods to find the root of equation $F(x) = u$. Such methods will also be introduced in this course.

Generating normal samples

Normal samples are intensively used in statistics. In this section, we introduce a special transformation for generating normal samples. We need to only consider generating $N(0,1)$. Consider the following polar transformation:

$$X = R\cos(\theta), \quad Y = R\sin(\theta), \quad \text{for } \theta \in (0, 2\pi) \tag{4.5}$$

If $X,\ Y$ are independent and both have distribution $N(0,1)$, then

$$R^2 = X^2 + Y^2 \sim \chi^2(2) = \exp(1) \times 2 \tag{4.6}$$

$$\theta \sim \text{unif}(0, 2\pi) \tag{4.7}$$

We can generate R and θ from distributions $\sqrt{\exp(1) \times 2}$ and unif(0,1) respectively, then use transformation (4.5) to obtain $N(0,1)$ samples.

The following is an R function which uses the above method:

```
#use method of inverse cdf to generate "n" iid samples from exp(1)
gen_exp <- function(n)
{
   #generate unif(0,1) random numbers
   u <- runif(n)
   #transform the random numbers
   -log(1-u)
}

#use polar transformation to generate "n" normal samples
gen_normal <- function(n)
{
  #calculates size of random samples, which is greater than half of n
  size_sample <- ceiling(n/2)

  R <- sqrt(2*gen_exp(size_sample))
  theta <- runif(size_sample,0,2*pi)

  X <- R*cos(theta)
  Y <- R*sin(theta)

  c(X,Y)[1:n]
}
```

We generated 10000 samples in this way. The histogram and qq-plot are shown as follows:

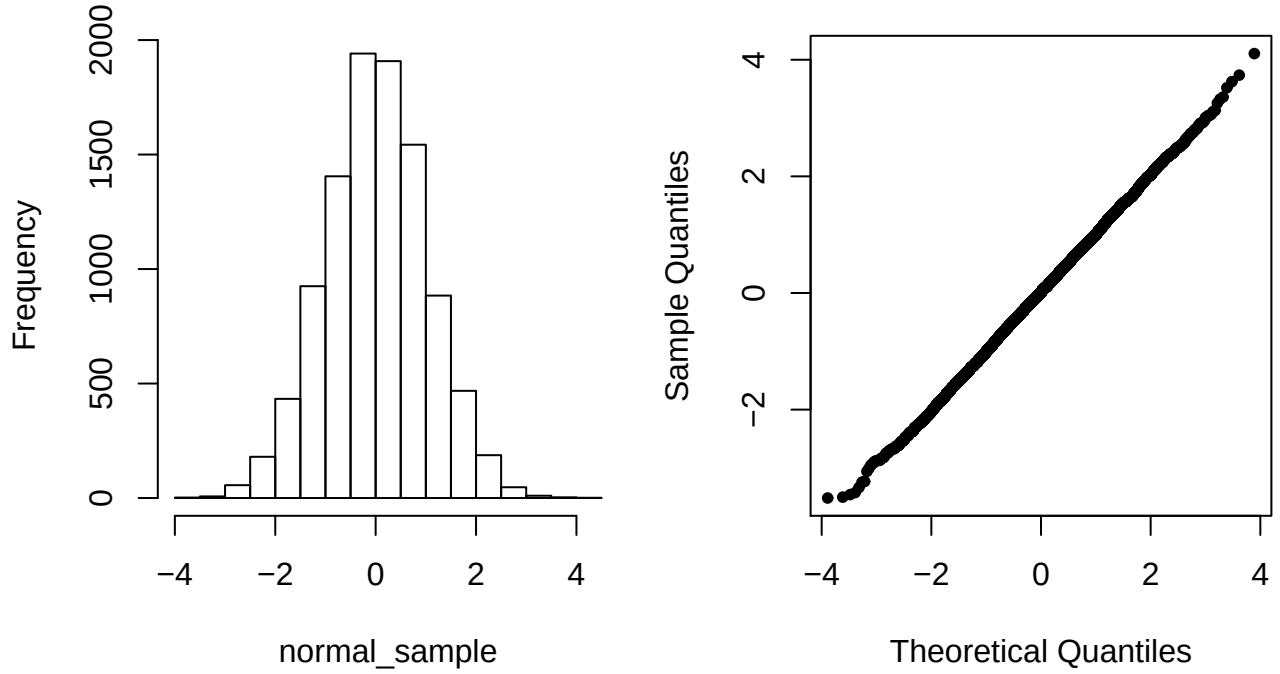


Figure 4.3: Histogram and QQ plot of 10000 Normal random numbers

4.3 Rejection sampling

Rejection sampling is a widely applicable method for generating random numbers from complicated 1-dimensional distributions.

Suppose the distribution we want to sample from has a density function $f(x)$, and we can find an envelop function $g(x)$ satisfying

$$g(x) \geq f(x), \qquad \text{for } x \in R.$$

The following procedure due to John von Neumann can generate n samples from $f(x)$:

1. Draw $X \sim \frac{g(x)}{\int_R g(x)dx}$, and $U \sim$ unif(0,1),
2. If $U \leq \frac{f(X)}{g(X)}$, keep X, otherwise discard X and go back to step 1,
3. Repeat step 1 and 2 until we obtain n samples.

The rejection sampling above can be rewritten as follows, which is ready to validate the algorithm:

1. Draw $X \sim \frac{g(x)}{\int_R g(x)dx}$, and $U \sim$ unif(0,1), let $V = U \times g(X)$,
2. If $V \leq f(X)$, keep (X, V), otherwise discard it and go back to step 1,

3. Repeat step 1 and 2 until we obtain n samples.

Before we validate the above procedure, we state a theorem:

Theorem 4.3.1 *Let $g(x)$ be an arbitrary non-negative function, i.e. $g(x) \geq 0$ for each x. If random variables (X, V) are distributed uniformly over the region between the curve $g(x)$ and the x-axis (referring to the blue region in Figure 4.4), then the marginal density function of X is proportional to $g(x)$:*

$$f_X(x) = \frac{g(x)}{\int_R g(x)} \tag{4.8}$$

Proof: The joint density function of (X, V) is:

$$\begin{aligned} f_{X,V}(x,v) &= \frac{1}{\text{area}(\{(x,v)|0 \leq v \leq g(x)\})} I(0 \leq v \leq g(x)) && (4.9)\\ &= \frac{1}{\int_R g(x) d\, x} I(0 \leq v \leq g(x)) && (4.10) \end{aligned}$$

The marginal density function of X is found by integrating away v:

$$\begin{aligned} f_X(x) &= \int_0^{g(x)} f(x,v) d\, v && (4.11)\\ &= \frac{g(x)}{\int_R g(x) d\, x} && (4.12) \end{aligned}$$

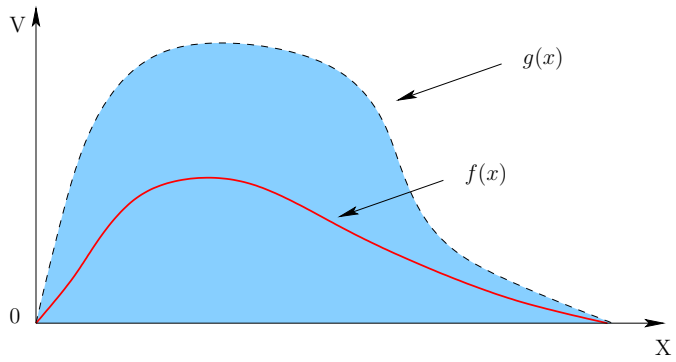
In other words, the joint density function $f_{X,V}(x, v)$ can be written as

$$\begin{aligned} f_{X,V}(x,v) &= f_X(x)\, f_{V|X}(v|x) && (4.13)\\ &= \frac{g(x)}{\int_R g(x) d\, x} \frac{I(0 \leq v \leq g(x))}{g(x)} && (4.14) \end{aligned}$$

From equation (4.14), if we draw a sample X from the $\frac{g(x)}{\int_R g(x) d\, x}$, and then draw a sample from uniform distribution over $(0, g(x))$, we obtain a sample (X, V) from the uniform distribution over the region between the curve $g(x)$ and the x-axis.

Now, we can easily verify the rejection sampling procedure. In step 1, we actually draw a sample (X, V) uniformly from the region under the curve $g(x)$ and above the x-axis. In step 2 we keep (X, V) only under the curve $f(x)$. The retained (X, V) is distributed uniformly over the region under the curve $f(x)$ and above the x-axis. The marginal density function of X is therefore the curve $f(x)$.

From Figure 4.4, we can see that the overall efficiency is determined by the ratio of the area between the curve $f(x)$ and the x-axis to the area between the curve $g(x)$ and the x-axis. When $f(x)$ and $g(x)$



Rejection Sampling:

1. Draw (X, V) uniformly from the blue region
2. Keep the points under the curve $f(x)$

Figure 4.4: Graph demonstrating rejection sampling

are close, this ratio is high and the efficiency of rejection sampling is accordingly high.

Example: We first use a toy example to demonstrate the rejection sampling. Suppose we want to sample from unif$(0, 1/2)$, and we do not know use transformation methods. The density function we want to sample from is

$$f(x) = 2\, I(0 \le x \le 1/2). \tag{4.15}$$

We know how to sample from the other distribution unif$(0, 1)$, whose density function is proportional to the following function:

$$g(x) = 2\, I(0 \le x \le 1). \tag{4.16}$$

It is easy to see that $g(x) \ge f(x)$ for each $x \in R$. Applying the above rejection sampling to this choice of $g(x)$, we first draw sample (X, V) uniformly from the rectangle $(0, 1) \times (0, 2)$, and only keep the samples in the rectangle $(0, 1/2) \times (0, 1)$. Obviously, the marginal distribution of X of these retained samples is uniformly distributed over $(0, 1/2)$.

Example: Let's look at a more concrete example. we use it to sample from another class of intensively used distributions — Gamma distributions. A Gamma distribution with shape parameter α and rate parameter 1 is denoted as Gamma$(\alpha, 1)$, whose density function is:

$$f(x) = \frac{1}{\Gamma(\alpha)} x^{\alpha-1} \exp(-x), \quad \text{for } x > 0. \tag{4.17}$$

It can be shown that when $\alpha > 1$, the above $f(x)$ is dominated by the following function:

$$g(x) = \frac{(\alpha-1)^{\alpha-1} \exp(-(\alpha-1))}{1 + (x - (\alpha-1))^2/(2\alpha-1)} \tag{4.18}$$

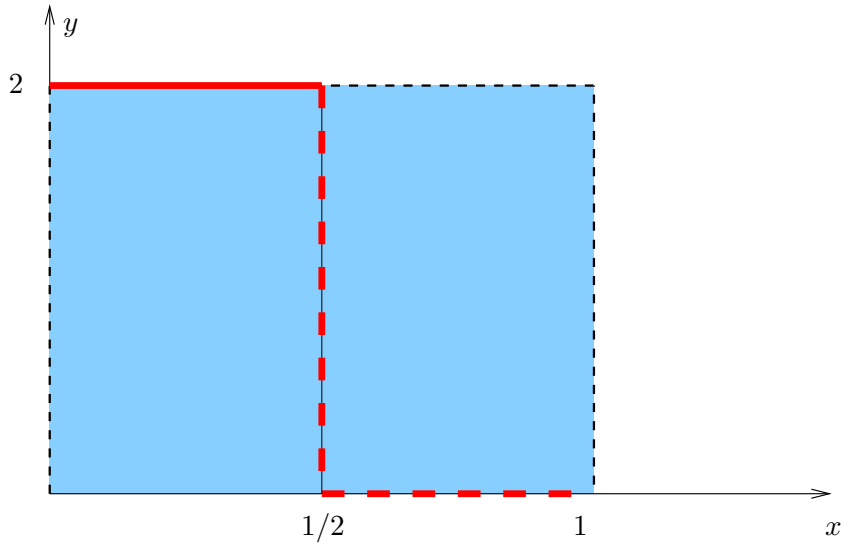


Figure 4.5: Sampling from uniform distribution over $(0, 1/2)$ using rejection sampling

The functions $f(x)$ and $g(x)$ with $\alpha = 2$ are displayed in Figure 4.6. The function $g(x)$ is proportional to the density of a Cauchy distribution with location parameter $\alpha - 1$ and scale parameter $\sqrt{2\,\alpha - 1}$, from which we can draw samples with inverse CDF method.

The following R codes implement rejection sampling for $f(x)$ with $g(x)$:

```
#log of a function which is always above the Gamma density function
log_g_gamma <- function(x,alpha)
{
   (alpha-1) * (log(alpha-1) - 1) - log( 1 + (x-(alpha-1))^2 / (2*alpha-1) )
}

#sampling from Gamma distribution with rejection sampling
sample_gamma_rej <- function(n,alpha)
{  sample_gamma <- rep(0,n)

   for(i in 1:n)
   {   rejected <- TRUE

       while(rejected)
       {  sample_gamma[i] <- rcauchy(1) * sqrt(2*alpha-1) + (alpha -1)
          U <- runif(1)
          rejected <- (log(U) > dgamma(sample_gamma[i],shape=alpha,log=TRUE) -
                       log_g_gamma(sample_gamma[i],alpha) )
       }
   }
   sample_gamma
}
```

Figure 4.7 shows the histogram and qqplot of 10000 random numbers drawn with R function `sample_gamma_rej`.

Appendix:

Figures 4.6 and 4.7 are produced by the following R codes, which are attached here for learning purpose.

```
source("gamma-rej.R")

postscript("draw-gamma.eps",paper="special",height=3.8,width=6.8,
           horizontal=FALSE)

par(mar=c(4,4,1,0))

x <- seq(0,10,length=100)

plot(x,log_g_gamma(x,2),type="l",ylab="log density",lty=2,ylim=c(-8,-1))

points(x,dgamma(x,shape=2,log=TRUE),type="l",lty=1,col="red")

dev.off()


postscript("sample-gamma-rej.eps",paper="special",height=3.3,width=6.8,
   horizontal=FALSE)

par(mar=c(4,4,1,1),mfrow=c(1,2))

rn_gamma_rej <- sample_gamma_rej(10000,2)

quantiles_rn_gamma_rej <- quantile(rn_gamma_rej,probs = seq(0,1,length=50))

quantiles_theory <- qgamma(seq(0,1,length=50),shape=2)

hist(rn_gamma_rej,main="")

plot(quantiles_theory,quantiles_rn_gamma_rej,ylim=c(0,6),pch=20,
     xlab="Theoretical Quantiles",ylab="Sample Quantiles")


dev.off()
```

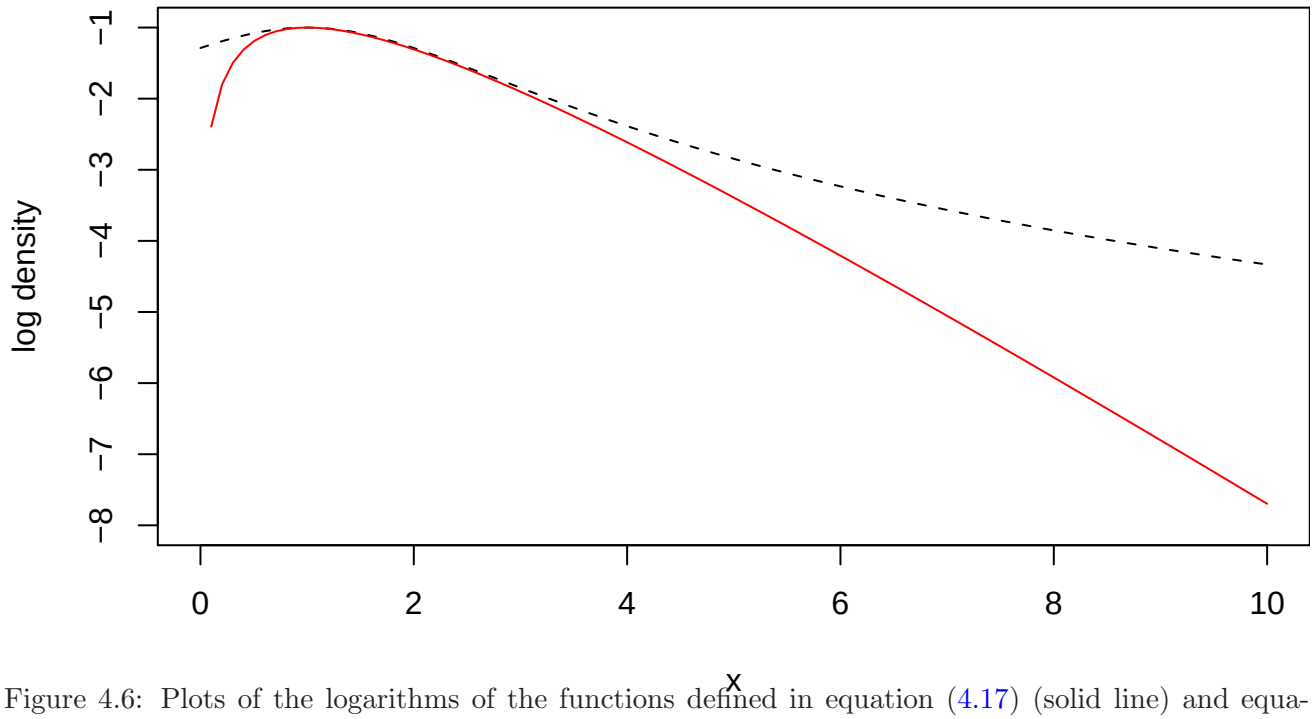


Figure 4.6: Plots of the logarithms of the functions defined in equation (4.17) (solid line) and equation (4.18) (dashed line), with $\alpha = 2$.

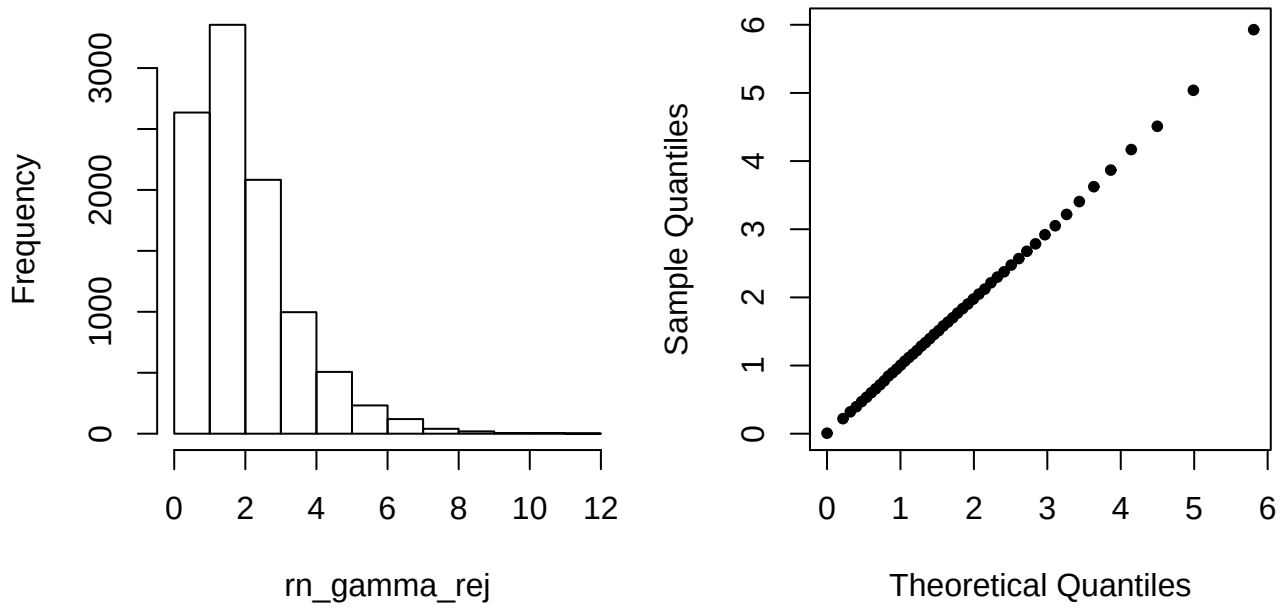


Figure 4.7: Plots testing 10000 random samples drawn from Gamma(2,1) with rejection sampling method.

4.3.1 Adaptive rejection sampling

Finding $g(x)$ for a $f(x)$ is tricky. Methods are developed to find the envolop function $g(x)$ for different types of $f(x)$. The following is one of such methods when $\log(f(x))$ is concave, using the fact that $\log(f(x))$ is dominated by the tangent function $u_n(x)$ at a number of abscissae $x_1, \ldots, x_n$. We initially set the abscissae and then take in new points when the target function $f(x)$ is evaluated at new points. We use $\exp(u_n(x))$ in the place of the envolop function $g(x)$. It is easy to sample from $\exp(u_n(x))$ since it is piece-wise exponential density functions and we can find its CDF analytically.

We also see that that the piece-wise linear function, denoted by $l_n(x)$, under the chords connecting the abscissae is always under $\log(f(x)))$. This function is called squeezing function. It is useful when the evaluation of $f(x)$ is costly, as we can decide to take a point in if we see that $\log(V)$ is already less than $l_n(x)$.

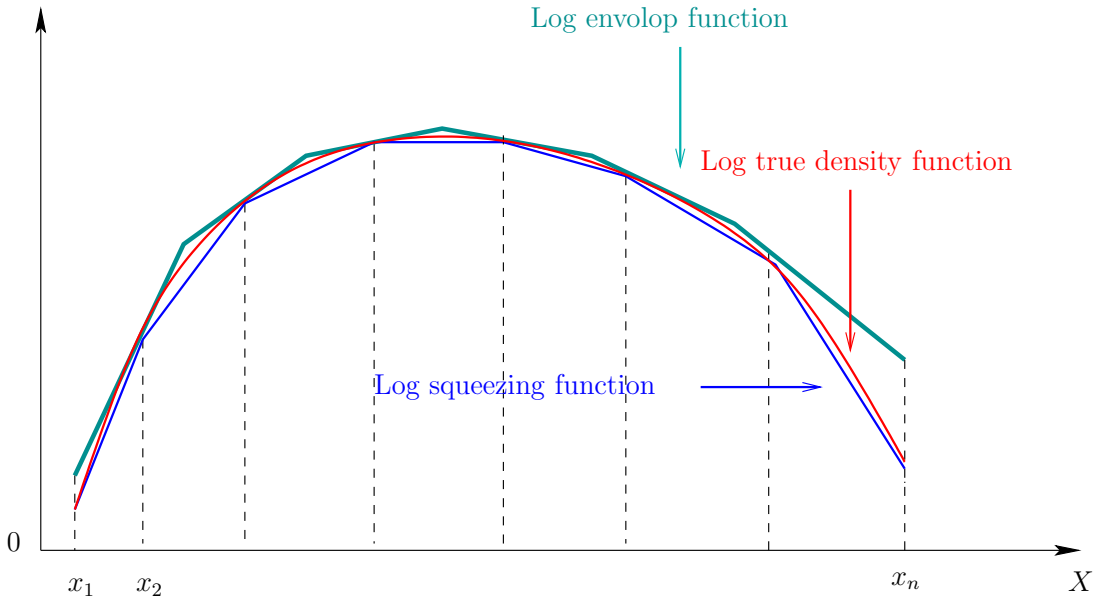


Figure 4.8: Graphical Illustration for Addaptive Rejection Sampling

Chapter 5

Monte Carlo estimation

5.1 Theoretical foundations

Monte Carlo estimation is justified theoretically by the *Law of Large Number and evaluated by* Central Limit Theorem.

Theorem 5.1.1 (Strong Law of Large Number) *If random variables $X_1, X_2, \ldots$, iid from a distribution with finit mean μ, then*

$$P\left(\frac{X_1 + \ldots + X_n}{n} \to \mu\right) = 1 \tag{5.1}$$

This theorem says that if we can simulate X_i long enough, the sample average will almost surely converge to the true mean.

In order to evaluate how quickly this average converge, we use the theorem:

Theorem 5.1.2 (Central Limit Theorem) *If random variables $X_1, X_2, \ldots$, iid from a distribution with finite mean μ and finite standard deviation σ, then*

$$\frac{\bar{X} - \mu}{\sigma/\sqrt{n}} \to^d N(0,1) \tag{5.2}$$

CLT can be used to calculate error bound given a confidence level. For instance, given confidence level $\alpha = 95\%$, we can conclude from CLT:

$$P(-1.96\,\sigma/\sqrt{n} \leq \bar{X} - \mu \leq 1.96\,\sigma/\sqrt{n}) \approx 0.95 \tag{5.3}$$

Obviously, we see that the width of error bound will decrease with n, in a rate of square root.

Let's demonstrate CLT by sampling from Gamma distribution, using the following R codes:

```
postscript("clt.eps",paper="special", height=9,width=7, horizontal=FALSE)

par(mfrow=c(3,2), mar=c(4,4,3,1))

sample_sizes <- c(4,4^2,4^3,4^4,4^5,4^6)

no_sim<- 10000

X <- rep(0,no_sim)

shape = 10

for(n in sample_sizes)
{
   for(i_sim in 1:no_sim)
      X[i_sim] <- mean(rgamma(n,shape=shape,scale=1) )

   hist(X,xlim=c(4,16),nclass=30,main=paste("n=",n))
}

dev.off()
```

5.2 A toy example: estimating π with Monte Carlo method

The expected value of some distribution is equal to the irrational constant π. We can use Monte Carlo method to estimate such expected values and consequently get an estimator of π if we do not know this value. We will discuss such an example.

Suppose random variables X, Y are uniformly distributed over rectangle $S = (-1, 1) \times (-1, 1)$, as displayed in Figure 5.2. The probability that (X, Y) lies in the ball $C = \{(x, y)|x^2 + y^2 \leq 1\}$ is $\pi/4$, in other words,

$$E(4 \times I((X, Y) \in C)) = \pi.$$

We can draw samples of (X, Y) uniformaly from S and count how much proportion of these samples fall into C, then π is estimated by:

$$\hat{\pi}_n = 4 \times \frac{\sum_{i=1}^{n} I(X_i, Y_i) \in C)}{n} \tag{5.4}$$

The following is an R function for estimating π using the above method:

```
# n is the number of samples drawn uniformly from the rectangle (-1,1)  * (-1,1)
# an estimate of pi is returned
```

The histogram of 10000 sample averages based on various numbers of samples are shown in Figure 5.1.

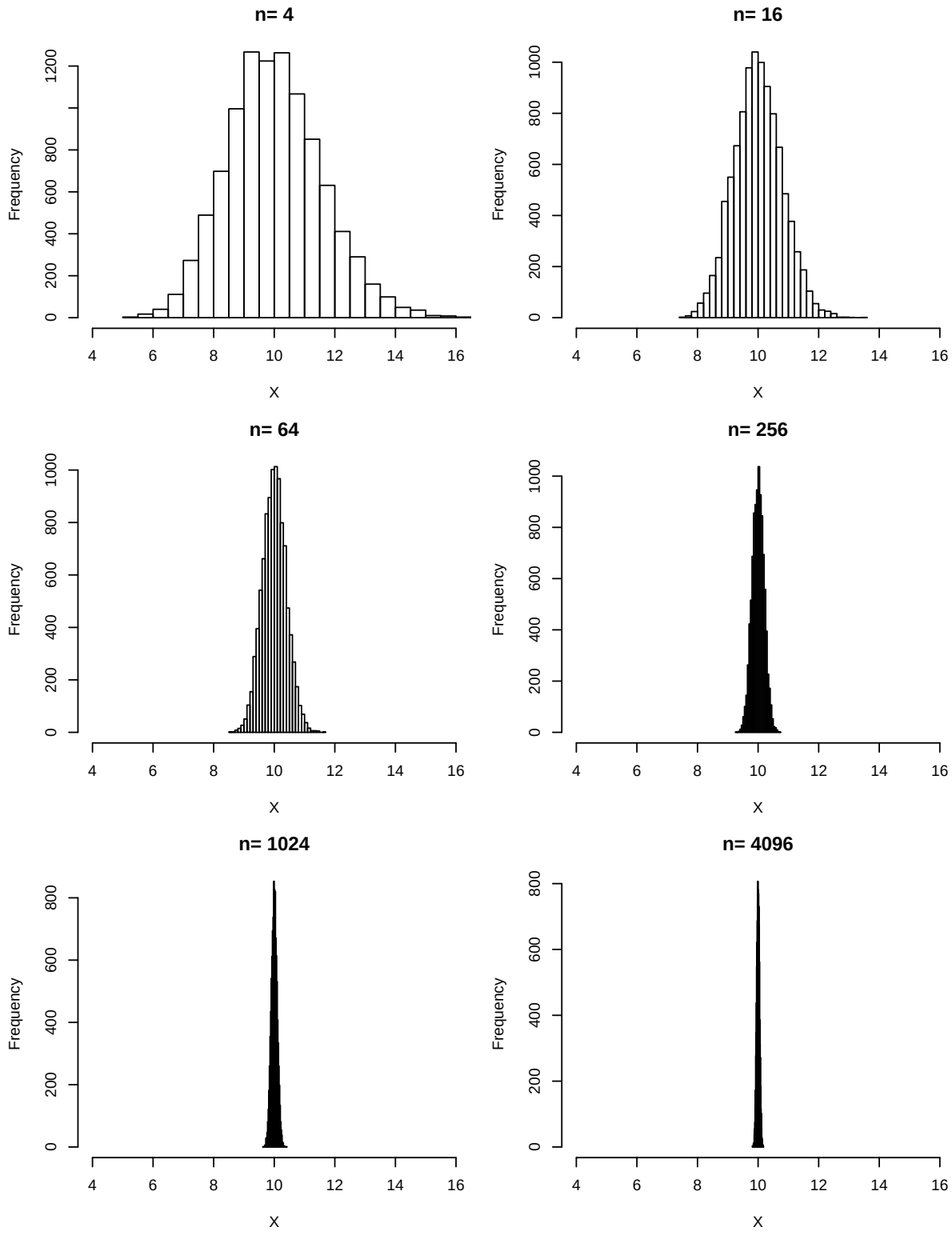
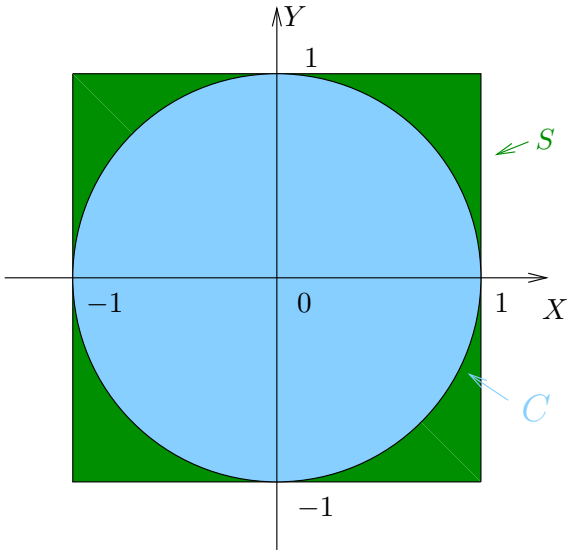


Figure 5.1: Demonstration of CLT using Gamma distribution with shape parameter = 10.



$(X, Y) \sim \text{Unif}(S)$

$E(4 \times I((X, Y) \in C)) = \pi$

Figure 5.2: A toy example of using Monte Carlo method to estimate π

```
pi_est_mc <- function(n)
{
     #X and Y are independent, both with marginal distribution unif(-1,1)
     X <- runif(n,-1,1)
     Y <- runif(n,-1,1)

     4 * mean(X^2 + Y^2 <= 1)
}
```

We test the above R function:

```
> source("est-pi.R")
>
> pi_est_mc(100)
[1] 3.12
>
> pi_est_mc(10000)
[1] 3.1232
>
> pi_est_mc(10000000)
[1] 3.140612
>
```

How to assess the error bound? The standard deviation of random variable $4 \times I((X, Y) \in C)$ is $4 \times \sqrt{\pi/4 \times (1 - \pi/4)} \approx 1.64$. The standard deviation of $\hat{\pi}_n$ is $1.64/\sqrt{n}$. For the last demonstration, the 95% confidence error bound is $1.96 \times 1.64/\sqrt{10^7} \approx 0.001$. When we report a Monte Carlo estimate, we usually also need to report the standard deviation. The number given above is within this 95%

confidence error bound, confirming the correctness of this analysis.

Question: How can we improve the above Monte Carlo estimation?

Chapter 6

Evaluating statistical methods using simulations

6.1 Evaluating point estimators using simulations

There are ususally many methods to estimate parameters in a statistical model. Comparing them analytically is often difficult or infeasible. Computer simuluation is an alternative, used ubiquitously in modern statistical research, due to its simplicity in implementation and interpretation. In practice, we may first use computer simulations to investigate or compare different statistical methods. The results may guide us to do more formal theoretical analysis.

6.1.1 Review of point estimation

We have a data set $x_1, \ldots, x_n$ available. We will model them as a realization of random variables $X_1, \ldots, X_n$ from some distribution. Taking parametric approach, we first assume a functional form for the density function or probability function, leaving some unknown parameters θ to be determined from the data, denoted by:

$$f(x_1, \ldots, x_n; \theta) \tag{6.1}$$

.

This assumed parametric distribution is called a statistical model for the data. A familiar example is the Gaussian model $N(\mu, \sigma^2)$, whose density function is:

$$f(x_1, \ldots, x_n; \mu, \sigma) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{(x_i - \mu)^2}{2\sigma^2}). \tag{6.2}$$

An estimator $\hat{\theta}_n$ for the unknown parameters θ is a function of data $X_1, \ldots, X_n$:

$$\hat{\theta}_n = T(X_1, \ldots, X_n) \tag{6.3}$$

There are many approaches to construct $\hat{\theta}_n$. One is called monent method, which simply equals the theoretical moments calculated from the assumed statistical model. Let

$$\alpha_k(\theta) = E(X^k), \text{ and } \hat{\alpha}_k = \frac{\sum_{i=1}^n X_i^k}{n}. \tag{6.4}$$

Suppose there are m elements in θ. The method of moment estimates θ by solving the following equations:

$$\begin{aligned} \alpha_1(\theta) &= \hat{\alpha}_1 && (6.5)\\ \alpha_2(\theta) &= \hat{\alpha}_2 && (6.6)\\ &\vdots \\ \alpha_m(\theta) &= \hat{\alpha}_m && (6.7) \end{aligned}$$

It can be shown that under appropriate regularity condtions, when $n \to \infty$, $\hat{\theta}_n$ will converge to θ in probability and $\sqrt{n}(\hat{\theta}_n - \theta) \to^d N(0, \Sigma)$. This approach is very straightforward and often easy in computation. However, it has been shown that is is worse asympotically than the MLE we are going to discuss next. This method can often be used in setting the initial value for some iterative methods which we will discuss later in this course.

Maximizing the likelihood function is another general approach for constructing an estimator. Simply, MLE $\hat{\theta}_n$ is the maximizer of the assumed probability distribution with regard to θ:

$$\hat{\theta}_n = \arg\max_{\theta} f(x_1, \ldots, x_n; \theta) \tag{6.8}$$

Obviouly, maximizing $f(x_1, \ldots, x_n; \theta)$ is equivalent to maximizing its logarithm:

$$\hat{\theta}_n = \arg\max_{\theta} \log(f(x_1, \ldots, x_n; \theta)) \tag{6.9}$$

Similarly, it can also be shown that under appropriate conditons, when $n \to \infty$, the MLE $\hat{\theta}_n$ will converge to θ in probability and $\sqrt{n}(\hat{\theta}_n - \theta) \to^d N(0, \Sigma)$. Additionally, it can be shown that MLE is optimal in terms of the asymptotical variance Σ.

Other than the MLE and Method of Moment estimators, there are also many natural estimator, for example unbiased estimator, ie, an estimator satisfying

$$E(\hat{\theta}_n) = \theta, \tag{6.10}$$

and some estimator based on Bayesian inference, which usually shrinks the estimator to some region where we believe the parameter θ is more likely to be.

The question is now how we choose from so many candidate estimators. *Decision theory* provides a framework for comparing estimators by comparing rish functions. Before we assess the estimators, we define our loss function $L(\theta, \hat{\theta})$, which measures how much loss occurs if the estimated value is different from the true value. Examples of loss functions inludes:

$$\text{Square error:} \quad L(\theta, \hat{\theta}) = (\hat{\theta} - \theta)^2 \tag{6.11}$$

$$\text{Absolute error:} \quad L(\theta, \hat{\theta}) = |\hat{\theta} - \theta| \tag{6.12}$$

$$\text{0-1 loss for discrete } \theta\text{:} \quad L(\theta, \hat{\theta}) = I(\theta \neq \hat{\theta}) \tag{6.13}$$

Estimators can be evaluated by looking at their expected loss, or average loss. In words, we look the average loss if we repeat the same estimation procedure many times. Formally, we need to calculate the risk function of an estimator, defined as follows:

$$R(\theta, \hat{\theta}_n) = E(L(\theta, \hat{\theta}_n)) \tag{6.14}$$

However, there is still much difficult in using this approach. There might not be a single estimator that is optimal for all value of θ. The optimal estimator might be different based on different loss functions. The calculation of the risk function $R(\theta, \hat{\theta}_n)$ is tedious or gives no close-form formula. Even when one can get a close-form formula for $R(\theta, \hat{\theta}_n)$, it may be quick and helpful to use computer simulation to investigate them in advance, which may give you hints how to do formal analysis, and also the results given by a simulation may be more direct and more convincing than a formal analysis involving many assumptions and approximations. In the next section, I will use several examples to demonstrate this.

6.1.2 Demonstrating examples

It is straightforward to use Monte Carlo methods to compute a risk function $R(\theta, \hat{\theta}_n)$. Given a value θ, we draw a certain number of samples from the assumed statistical model $f(x_1, \ldots, x_n; \theta)$, and compute the value of $L(\theta, \hat{\theta})$ for each estimate $\hat{\theta}$, finally average these values over all samples. We need to perform this procedure for many different values of θ and different n.

Example: (Bernolli models) Suppose $X_1, \ldots, X_n$ are iid from Bernolli(p). We want to estimate p. The following are two such estimators:

$$\hat{p}_1 = \bar{X} = \frac{X_1 + \ldots + X_n}{n} \tag{6.15}$$

$$\hat{p}_2 = \bar{X} = \frac{X_1 + \ldots + X_n + 1}{n + 2} \tag{6.16}$$

Obviously, when n is large, these two estimators yield very similar results. We want to see when n is small, how they differs. Of course, there are close-form formulae for the risk functions of these two estimators. We use this example only for demonstrating how to do simulations.

The following R functions are used to estimate MSE given a particular p and n for the previous two estimators.

```
p_est1 <- function(x) mean(x)

p_est2 <- function(x) (sum(x) + 1) / (n+2)

mse_est_mc <- function(n,p,no_sim,p_est)
{
   sq_error <- rep(0, no_sim)
   for(i_sim in seq(1,no_sim,by=1))
   {  sq_error[i_sim] <- (p_est( rbinom(n,1,p) ) - p)^2
   }
   list(mse=mean(sq_error), sd = sqrt(var(sq_error)/no_sim) )
}
```

We do simulation for $n = 10$ and $n = 100$ using the following R codes:

```
source("mse-bern.R")

postscript("risk-bern.eps",width=7,height=7,paper="special",horizontal=FALSE)

par( mfrow=c(2,2), mar=c(4,4,2,1) )

no_sim <- 10000

p_set <- seq(0,1,by = 0.05)

risk_est1 <- risk_est2 <- rep(0, length(p_set))
sd_risk_est1 <- sd_risk_est2 <- rep(0, length(p_set))

#############################################################################

n_set <- c(2,10,50,100)

for( n in n_set )
{

   for(i in seq(1,length(p_set),by=1) )
   {
       output_est <- mse_est_mc(n=n,p=p_set[i],no_sim=no_sim,p_est1)
       risk_est1[i] <- output_est$mse
```

```
        sd_risk_est1[i] <- output_est$sd

        output_est <- mse_est_mc(n=n,p=p_set[i],no_sim=no_sim,p_est2)
        risk_est2[i] <- output_est$mse
        sd_risk_est2[i] <- output_est$sd

    }

    plot(p_set, risk_est1,type="l",
         xlab="p",ylab="MSE",main=paste("n=",n))
    points(p_set, risk_est1 + 1.96*sd_risk_est1,type="l",lty=2)
    points(p_set, risk_est1 - 1.96*sd_risk_est1,type="l",lty=2)

    points(p_set, risk_est2,type="l",col=2)
    points(p_set, risk_est2 + 1.96*sd_risk_est2,type="l",lty=2,col=2)
    points(p_set, risk_est2 - 1.96*sd_risk_est2,type="l",lty=2,col=2)

}

dev.off()
```

The estimated risk functions for various p are displayed by Figure 6.1. From this figure, we can conclude that the estimator $\hat{p}_2$ performs better than the estimator $\hat{p}_1$ when p is near 0.5, and worse when p is close to 0 or 1.

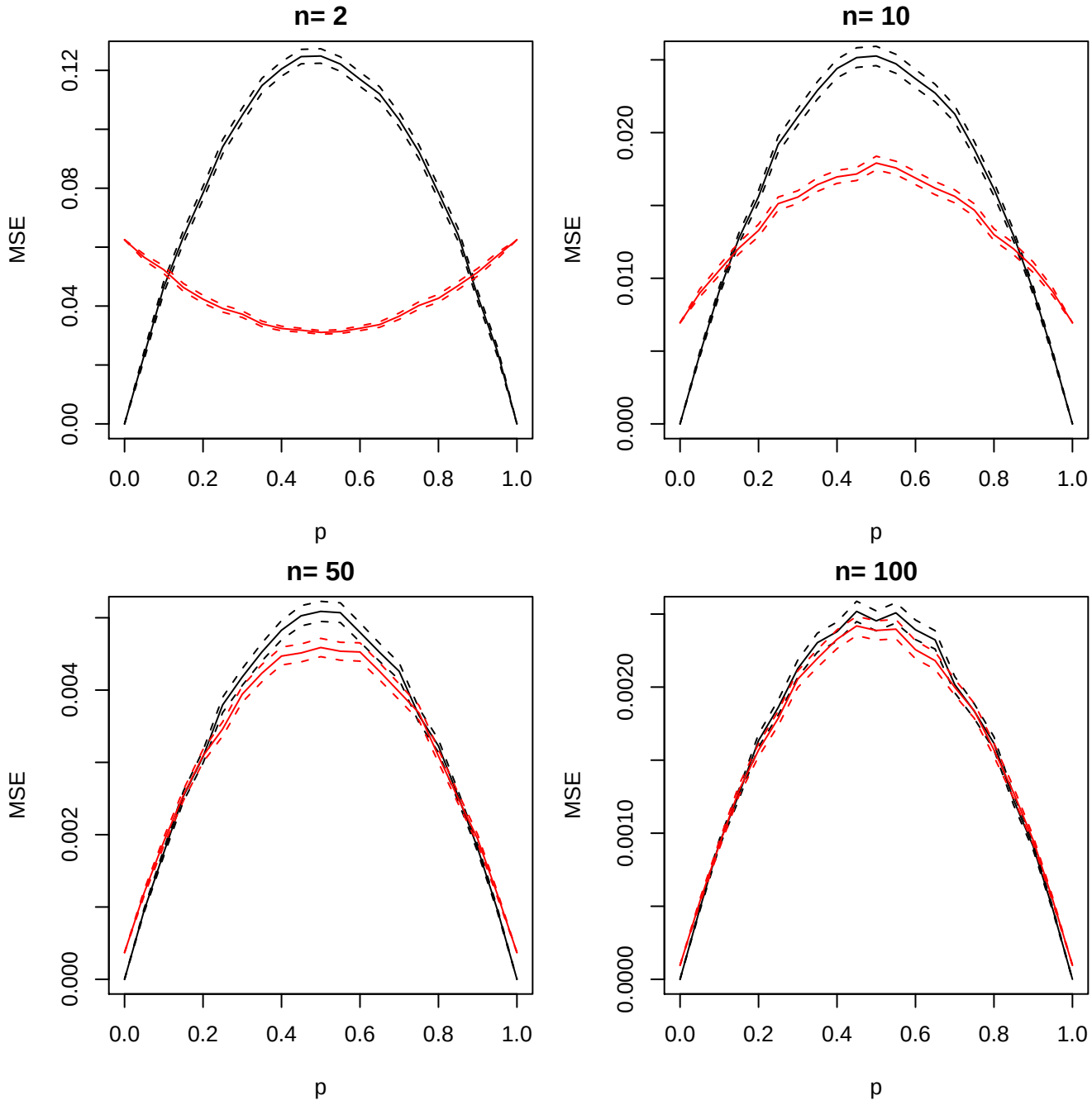


Figure 6.1: Estimated risk functions based on square error, of two estimators for proportions of Bernolli models. The dashed curves show the 95% Monte Carlo estimation error bounds. The black curves show for the estimator $\hat{p}_1$ in (6.15) and the red curves show for the estimator $\hat{p}_2$ in (6.16).

Example: (t Models) Suppose $X_1, \ldots, X_n$ are iid from $t(\mu, df)$. We want to estimate μ. The following are two reasonable estimators for μ:

$$\hat{\mu}_1 = \bar{X} = \frac{X_1 + \ldots + X_n}{n} \tag{6.17}$$

$$\hat{\mu}_2 = \text{median}(X_1, \ldots, X_n). \tag{6.18}$$

We will compare them in terms of absolute error. First note this expected loss is independent of the parameter μ. We therefore fix $\mu = 0$ in simulations. We will examine how the risk function depends on n and degree freedom.

The following R functions estimates the risk function for fixed degree freedom *df* and sample size n:

```
mu_est1 <- function(x) mean(x)

mu_est2 <- function(x) median(x)


t_abse_est_mc <- function(n,df,no_sim,mu_est)
{
   abs_error <- rep(0, no_sim)
   for(i_sim in seq(1,no_sim,by=1))
   {  abs_error[i_sim] <- abs(mu_est( rt(n,df) ))
   }
   list(abse=mean(abs_error), sd = sqrt(var(abs_error)/no_sim) )
}
```

The following R codes perform a simulation:

```
source("abs-t.R")

#simulate for square error

postscript("abse-t.eps",width=7,height=7,paper="special",horizontal=FALSE)

par( mfrow=c(2,2), mar=c(4,4,2,1) )

no_sim <- 5000

n_set <- seq(5,50,by=2)

risk_est1 <- risk_est2 <- rep(0, length(n_set))
sd_risk_est1 <- sd_risk_est2 <- rep(0, length(n_set))

##############################################################################
```

```
df_set <- c(1.5,2,10,50)

for( df in df_set )
{

   for(i_n in seq(1,length(n_set),by=1) )
   {
       output_est <- t_abse_est_mc(n=n_set[i_n],df=df,no_sim=no_sim,mu_est1)
       risk_est1[i_n] <- output_est$abse
       sd_risk_est1[i_n] <- output_est$sd

       output_est <- t_abse_est_mc(n=n_set[i_n],df=df,no_sim=no_sim,mu_est2)
       risk_est2[i_n] <- output_est$abse
       sd_risk_est2[i_n] <- output_est$sd

   }

   ylim <- c( min(c(risk_est1 - 1.96*sd_risk_est1,
                    risk_est2 - 1.96*sd_risk_est2)),
      max(c(risk_est1 + 1.96*sd_risk_est1,
            risk_est2 + 1.96*sd_risk_est2))
            )
   plot(n_set, risk_est1,type="l",ylim=ylim,
        xlab="n",ylab="Absolute Error",main=paste("df =",df))
   points(n_set, risk_est1 + 1.96*sd_risk_est1,type="l",lty=2)
   points(n_set, risk_est1 - 1.96*sd_risk_est1,type="l",lty=2)

   points(n_set, risk_est2,type="l",col=2)
   points(n_set, risk_est2 + 1.96*sd_risk_est2,type="l",lty=2,col=2)
   points(n_set, risk_est2 - 1.96*sd_risk_est2,type="l",lty=2,col=2)

}

dev.off()
```

The results are displayed by Figure 6.2, from which we see that when *df* is small, median is a better estimator than mean.

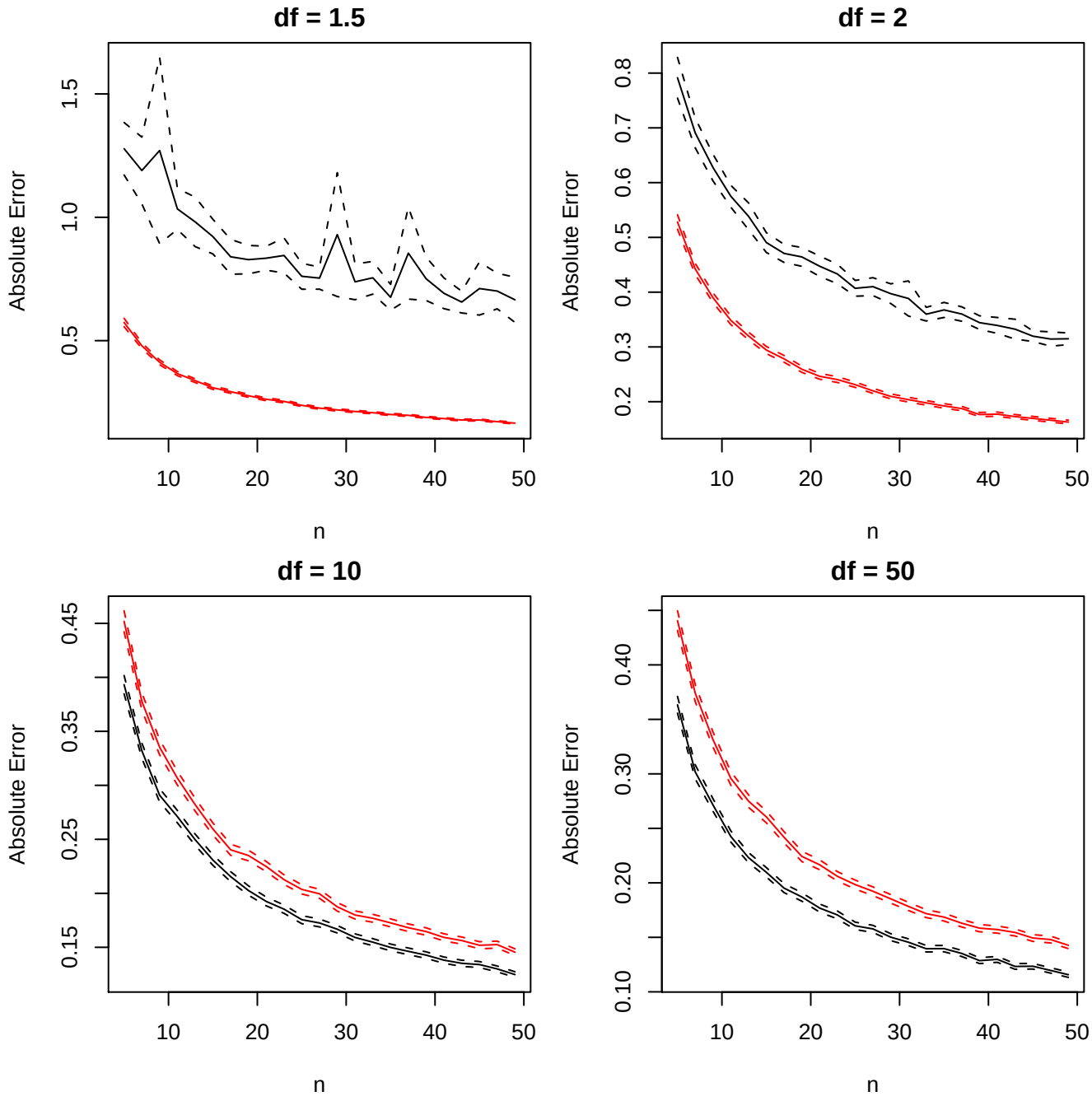


Figure 6.2: Estimated risk functions based on absolute error, of two estimators for modes of t distributions. The dashed curves show the 95% Monte Carlo estimation error bounds. The black curves show for the estimator $\hat{\mu}_1$ in (6.17) and the red curves show for the estimator $\hat{\mu}_2$ in (6.18).

Example: (Uniform Distributions) Suppose $X_1, \ldots, X_n$ are iid from unif$(0, \theta)$. We want to estimate θ. The following are two reasonable estimators for θ:

$$\hat{\theta}_1 = \max(X_1, \ldots, X_n) \tag{6.19}$$

$$\hat{\theta}_2 = \frac{n+1}{n} \max(X_1, \ldots, X_n) \tag{6.20}$$

We will compare $\hat{\theta}_1$ and $\hat{\theta}_1$ using both square and absolute error. Note that the expected values of these two estimators depends on θ in an obvious way. We fix $\theta = 1$. The results are displayed in Figure 6.3.

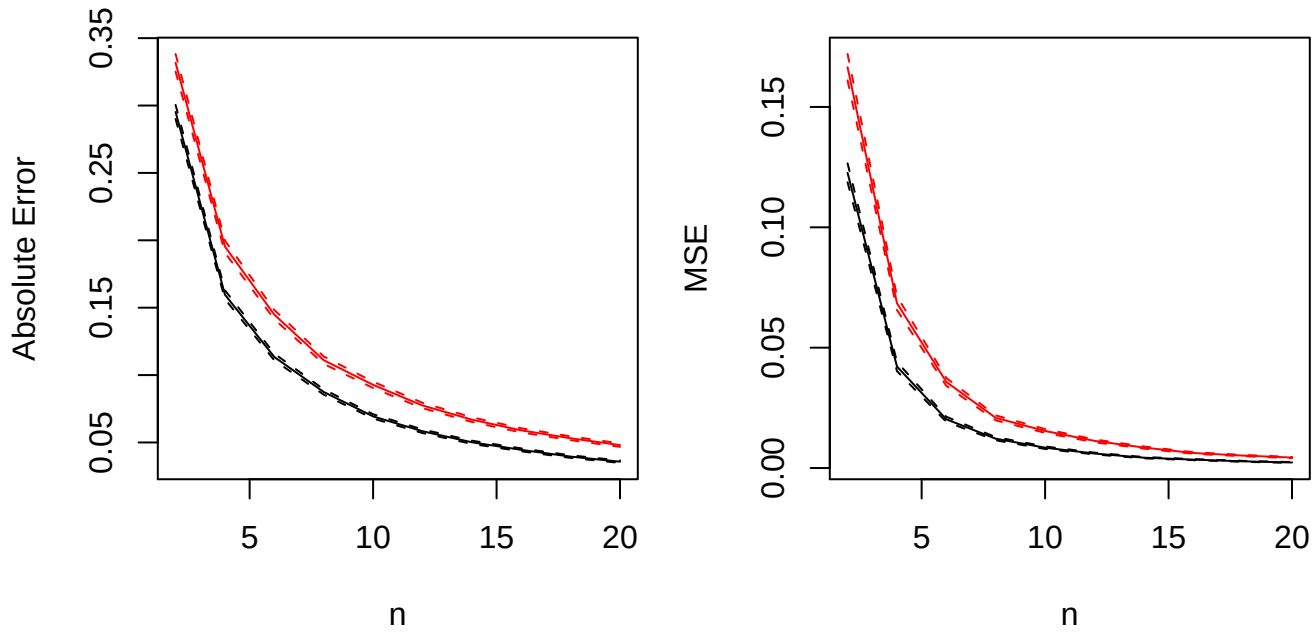


Figure 6.3: Estimated risk functions for estimating the upper limit of uniform distribution. The red curves display for $\hat{\theta}_1$ and the black curves display for $\hat{\theta}_2$

6.2 Evaluating testing methods using simulations

6.2.1 Review of hypothesis testing

Critical value

We are interested in arguing against a well-established hypothesis. This hypothesis is called **Null hypothesis**, denoted by H_0. Examples of such H_0 includes:

H_0 : Drug A is useless for a certain disease
H_0 : there is not relationship between variables y and x

At the same time, if we reject this null hypothesis, we also propose an alternative hypothesis, for example:

H_1 : Drug A improves the health condition of patients with a certain disease
H_1 : Drug A worsens the health condition of patients with a certain disease
H_1 : Drug A has effect on a certain disease (improve or worsen)
H_1 : there is relationship between variables y and x
H_1 : there is positive relationship between variables y and x
H_1 : there is negative relationship between variables y and x

Formally, we assume some parametric distribution on the data with unknown parameters. Then, based on this model, the null hypothesis and alternative hypothesis are phrased as the parameters are in different region of parameter space. Typical hypotheses includes:

$$\begin{aligned} &\theta = \theta_0 \quad \text{VS} \quad \theta \neq \theta_0 \text{ (two-sided test)} \\ &\theta > \theta_0 \quad \text{VS} \quad \theta < \theta_0 \text{ (1-sided test)} \\ &\theta < \theta_0 \quad \text{VS} \quad \theta > \theta_0 \text{ (1-sided test)} \end{aligned}$$

To see whether H_0 or H_1 is true, we will propose a test statistic T:

$$T = T(X_1, \ldots, X_n). \tag{6.21}$$

Typically, the value of T is larger, the H_1 is more supported, in other words, large T favours H_1.

Example: Suppose $X_1, \ldots, X_n$ are the measurements on n patients indicating how healthy they are after they are treated by a new drug A, and $Y_1, \ldots, Y_n$ are the same measurements on another n patients after they are treated by a placebo. In most situations, we can model these data using Gaussian

distribution:

$$X_1, \ldots, X_n \text{ iid} \sim \text{Normal}(\mu_x, \sigma^2) \tag{6.22}$$
$$Y_1, \ldots, Y_n \text{ iid} \sim \text{Normal}(\mu_y, \sigma^2) \tag{6.23}$$

We can use $H_0 : \mu_x - \mu_y = 0$ to phrase the null hypothesis that drug A is helpless to the patients with a certain disease, use $H_1 : \mu_x - \mu_y \neq 0$ to phrase that this drug is taking effect on a certain disease, and use $H_1 : \mu_x - \mu_y > 0$ to phrase that this drug is helpful for patients with the disease.

If the alternative hypothesis is $H_1 : \mu_x - \mu_y > 0$, a reasonable testing statistic is

$$T = \frac{\bar{X} - \bar{Y}}{\sqrt{2}\hat{\sigma}/\sqrt{n}}, \tag{6.24}$$

where $\hat{\sigma}$ is an estimator of σ, usually taken as $\sqrt{\frac{\sum(X_i - \bar{X})^2 + \sum(Y_i - \bar{Y})^2}{2n-2}}$.

If the alternative hypothesis is $H_1 : \mu_x - \mu_y \neq 0$, a reasonable testing statistic is

$$T = \frac{|\bar{X} - \bar{Y}|}{\sqrt{2}\hat{\sigma}/\sqrt{n}}. \tag{6.25}$$

A test procedure is that when T is larger than a critical value we are going to reject H_0 in favour of H_1. The difficult question is how large this critical value should be. 0.1? 1? 10? 100?

Statisticians made a criterion: Given a confidence level α (usually as small as for example 0.05), the critical value C_α is the value satisfying;

$$\max_{\theta \in H_0} P(T > C_\alpha | \theta) \leq \alpha \tag{6.26}$$

Remarks:

- Very often H_0 is simple and contains only a point in parameter space, for example $\theta = 0$.
- Theoretically, this guarantees that the probability we make wrong decision when H_0 is true is less than α. Explained in frequentist way, if we make decision using this C_α on 100 data sets with $\theta \in H_0$, only approximately $\alpha \times 100$ times we made wrong decision. If we use a very small α, say 0.05, it means we are very conservative on H_0, ie, we do not easily reject H_0.
- On the other hand, if we reject H_0 under a small α, it usually indicates strong evidence against H_0. This can be seen from the following probability based on Bayes rule:

$$P(\theta \in H_0|T > C_\alpha) = \frac{P(T > C_\alpha|\theta \in H_0)P(\theta \in H_0)}{P(T > C_\alpha|\theta \in H_0)P(\theta \in H_0) + P(T > C_\alpha|\theta \in H_1)P(\theta \in H_1)} \tag{6.27}$$

If $P(T > C_\alpha|\theta \in H_0)$ is very small, and $P(T > C_\alpha|\theta \in H_1)$ is usually much larger than $P(T > C_\alpha|\theta \in H_0)$ (note that H_1 favours larger T), the probability on the left of (6.27) will be very small. For example, plugging $P(T > C_\alpha|\theta \in H_0) = 0.05$,$P(T > C_\alpha|\theta \in H_1) = 0.5$, and $P(\theta \in H_0) = P(\theta \in H_1) = 0.5$ in (6.27) gives $P(\theta \in H_0|T > C_\alpha) = \frac{0.05\times0.5}{0.05\times0.5+0.5\times0.5} = \frac{0.05}{0.05+0.5} = 0.09$.

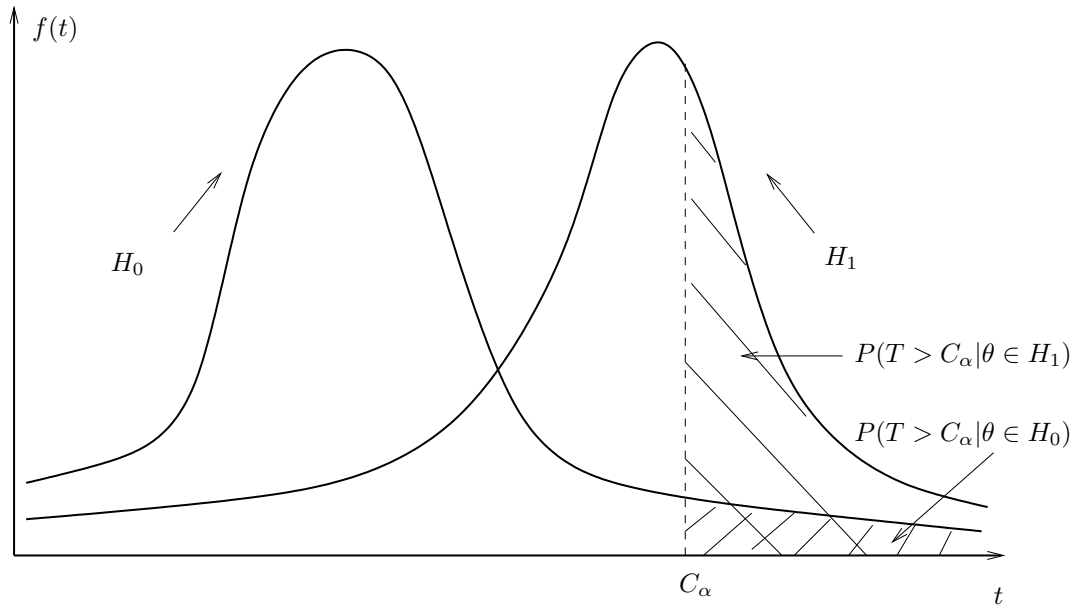


Figure 6.4: A typical picture of the density function of test statistic under H_0 and H_1.

Power of a test method

From the equation (6.27), we can see that $P(\theta \in H_0|T > C_\alpha)$ strongly depends on the ratio $P(T > C_\alpha|\theta \in H_1)$ to $P(T > C_\alpha|\theta \in H_0)$:

$$P(\theta \in H_0|T > C_\alpha) = \left(1 + \frac{P(T > C_\alpha|\theta \in H_1)P(\theta \in H_1)}{P(T > C_\alpha|\theta \in H_0)P(\theta \in H_0)}\right)^{-1} \tag{6.28}$$

When this ratio is larger, the test is more *reliable*. This ratio is illustrated by Figure 6.4. The probability $P(T > C_\alpha|\theta \in H_1)$ is called the power of a test T under confidence level α, in contrast, $P(T > C_\alpha|\theta \in H_0)$ is called the probability of type 1 error. The probability $P(T > C_\alpha|\theta \in H_1)$, i.e. 1-power, is called the probability of type 2 error. The following table is usually used to display these quantities:

	Accept H_0	Reject H_1
H_0 is true	OK	Type 1 error
H_1 is true	Type 2 error	OK

Naturally, different test procedures with the same probability of type 1 error are compared by the power. Given a confidence level α, if a test is more powerful than all others, it is called *most powerful* test. Unfortunately, for most test problems, for example the two-sided problems, such a most powerful testing procedure does not exist. In most practical problems, calculating power analytically is usually impossible. We often only investigate the power property of a given test for some particular $\theta \in H_1$.

P-value

There are many difficulties in using the above procedure with given confidence level:

- Different people may use different confidence level.
- People may want to know more information than the final decision whether the hypothesis is rejected or accepted. For example, the critical value C_α is 1.96, both 1.97 and 19.7 results in rejection, but the evidence given by 19.7 is obviously stronger than 1.97.
- The value of test statistic T is not standardized. Using the previous t test example, if another group of people change the unit in measuring, say using $X^* = 100\,X$. Obviously the critical value will be 100 times of the original one, for example the original critical value is 1.96, and the critical value for the 2nd group is 196. Now, if a group of people using the original unit report a value of the test statistic 1.99, but the second group of people get a value of 198 from another group of patients. It looks like $198 - 196 = 2$ is much larger than $1.99 - 1.96 = 0.03$, but actually 1.99 gives stronger evidence against the null hypothesis.

The solution is to report the probability of type 1 error if using the value t of the test statistic observed from the current data set as critical value:

$$p(t) = P(T \geq t | \theta \in H_0) \tag{6.29}$$

One can use it to decide whether to accept or reject the hypothesis based on any confidence level. This value is standardized for all tests. From it, we can also easily read information how strongly the data argues against the null hypothesis. For example, clearly we know p-value 0.005 disfavours the null hypothesis much more than 0.05. However, the difference of corresponding values of test statistic for p-value 0.005 and 0.05 may vary much for different problems.

Actually, the p-value $p(t)$ is just a monotone transformation of test statistic t. $p(T)$ itself is a test statistic. The difference is that its distribution is the same for all test problems.

Theorem 6.2.1 (uniformity of p-value) *Suppose T is a continuous random variable. Under null hypothesis H_0, $p(T)$ is uniformly distributed over interval (0,1).*

Proof: We have shown this result when we introduce generating random number by inverting CDF. Actually $p(T) = 1 - F(T)$, where $F(x)$ is the CDF of T under H_0. It is easy to see that $F(T) \sim \text{unif}(0,1)$: $P(F(T) \leq u) = P(T \leq F^{-1}(u)) = F(F^{-1}(u)) = u$.

This proof is illustrated by Figure 4.2.

A typical picture of the probability density function of $p(T)$ is displayed in Figure 6.5. Under H_0, it has unif$(0, 1)$ distribution, and under H_1, it is more likely to take small values.

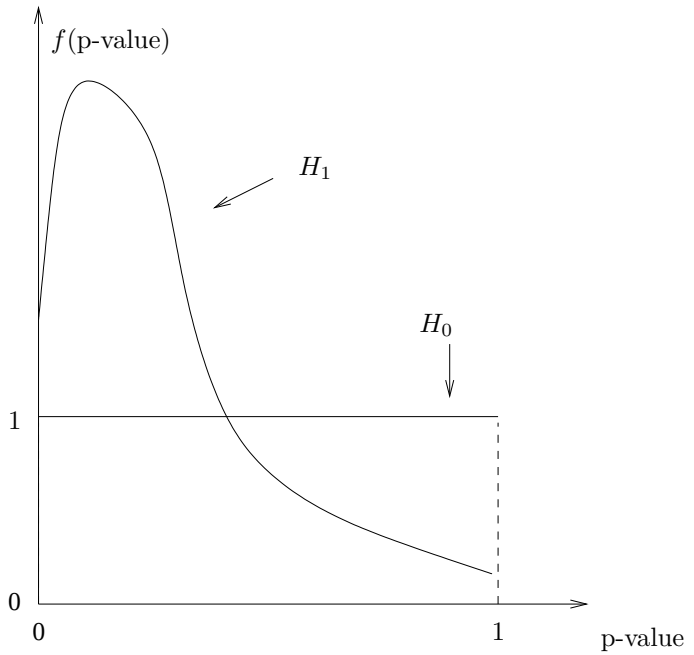


Figure 6.5: A typical density function of p-value under H_0 and H_1.

Very often, we can not find p-value exactly. People therefore use various methods to approximate the p-value, for example using the asymptotic distribution of T to replace the distribution of T with a fairly large number of sample size, or some other complicated procedures. Regardless how the p-value is produced, we can use this property of p-value to check whether a test procedure is valid. To do this, we simply sample T from the null hypothesis H_0, and draw the histogram of p-value.

6.2.2 Permutation test

Permutation test is a class of nonparametric tests. We can think of it as a test procedure for two-sided hypothesis.

One example of permutation test is to test whether the distribution for a group of data $X_1, \ldots, X_m$ is the distribution for the other group of data $Y_1, \ldots, Y_n$. Let $T(X_1, \ldots, X_m, Y_1, \ldots, Y_n)$ be some test statistic, for example,

$$T(X_1, \ldots, X_m, Y_1, \ldots, Y_n) = |\bar{X} - \bar{Y}| \tag{6.30}$$

If the distributions of X and Y are the same, the order of X_i and Y_i should not matter. We can permute these $m + n$ samples to approximate the distribution of T under null hypothesis. For each permuted sample, we compute the value of the test statistic T. We can then approximate $P(T > t_{\text{obs}})$:

$$\text{p-value} = \frac{1}{(m+n)!} \sum_{j=1}^{(m+n)!} I(T_j > t_{\text{obs}}) \tag{6.31}$$

In practice, $(m+n)!$ may be huge. We usually only permute these $m+n$ samples a pre-specified B times. The p-value is calculated by

$$\text{p-value} = \frac{1}{B} \sum_{j=1}^{B} I(T_j > t_{\text{obs}}) \tag{6.32}$$

Another example is to test whether there is relationship between $X_1, \ldots, X_n$ and $Y_1, \ldots, Y_n$. We may use some statistic to indicate the dependency, for example the absolute sample correlation:

$$T(X_1, \ldots, X_n, Y_1, \ldots, Y_n) = \left| \frac{\sum_{i=1}^n (X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^n (X_i - \bar{X})^2 \sum_{i=1}^n (Y_i - \bar{Y})^2}} \right| \tag{6.33}$$

If there is no relationship between X and Y, the order of $X_1, \ldots, X_n$ does not matter. We can permute $X_1, \ldots, X_n$ to obtain an approximate the distribution of T under null hypothesis, and calculate the p-value in the same way as in (6.32).

6.2.3 Demonstrating examples

Example: (t test on data from t distribution) When data is from Normal distribution, t test is exact. However, in practice, we often do not have reason to believe this assumption. We want to see how t test performs when this assumption is violated. In particular, we look at its performance when the samples are from t distribution with various degree of freedom.

The following are the R functions for performing the experiments:

```
# SIMULATE T TESTS ON DATA FROM A T DISTRIBUTION.  Simulates k data
# sets, each consisting of n data points that are drawn independently
# from the t distribution with df degrees of freedom.  For each data
# set, the p-value for a two-sided t test of the null hypothesis that
# the mean is mu (default 0) is computed.  The value returned by this
# function is the vector of these k p-values.

t.test.sim <- function (k, n, df, mu=0)
{
  pvalues <- numeric(k)

  for (i in 1:k)
  { x <- rt (n, df)
```

```
    pvalues[i] <- t.test.pvalue(x,mu)
  }

  pvalues
}


# FIND THE P-VALUE FOR A TWO-SIDED T TEST.  The data is given by the first
# argument, x, which must be a numeric vector.  The mean under the null
# hypothesis is given by the second argument, mu, which defaults to zero.
#
# Note:  This function is just for illustrative purposes.  The p-value
# can be obtained using the built-in t.test function with the expression:
#
#     t.test(x,mu=mu)$p.value

t.test.pvalue <- function (x, mu=0)
{
  if (!is.numeric(x) || !is.numeric(mu) || length(mu)!=1)
  { stop("Invalid argument")
  }

  n <- length(x)

  if (n<2)
  { stop("Can't do a t test with less than two data points")
  }

  t <- (mean(x)-mu) / sqrt(var(x)/n)

  2 * pt (-abs(t), n-1)
}
```

The results are shown by Figure 6.6.

Example: (Permutation test on correlation) We test the permutation test with X and Y drawn from Gaussian distribution.

```
# COMPUTE THE P-VALUE FOR A PERMUTATION TEST OF CORRELATION.  Tests the null
# hypothesis that the the vectors x and y are independent, versus the
# alternative that they are correlated (either positively or negatively).
# The vectors x and y are given as the first and second arguments; they
# must be of equal length.
#
# The p-value returned is computed by simulating permutations of how the
# elements of the vectors are paired up, with the simulation sample
# size being given as the third argument, n, which defaults to 999.  The
```

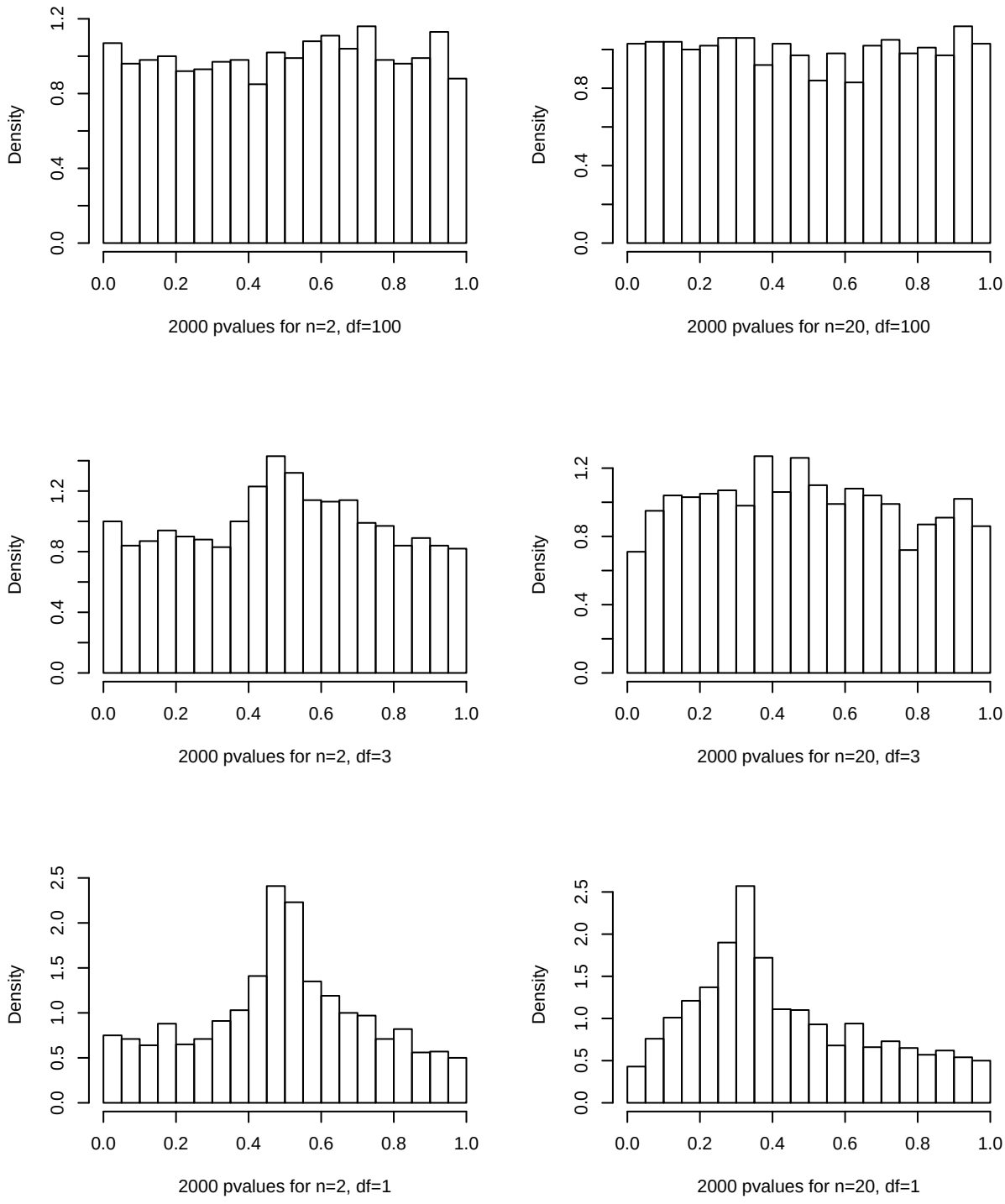


Figure 6.6: Results of t test on data from t distribution.

```
# p-values returned are integer multiples of 1/(n+1), and have the property
# that if the null hypothesis is true, the probability of obtaining a p-value
# of k/(n+1) or smaller is equal to k/(n+1), unless there is exact equality
# for the correlations obtained with different permutations, in which case the
# probability may differ slightly from this.

perm.cor.test <- function (x, y, n=999)
{
  real.abs.cor <- abs(cor(x,y))

  number.as.big <- 0
  for (i in 1:n)
  { if (abs(cor(x,sample(y))) >= real.abs.cor)
    { number.as.big <- number.as.big + 1
    }
  }

  (number.as.big + 1) / (n + 1)
}


# TEST ON NORMALLY-DISTRIBUTED DATA, COMPARED TO TEST BASED ON NORMAL DIST.

test.perm.norm <- function(no.sim,no.perm)
{
  pvalue <- rep(0,no.sim)

  for(i in 1:no.sim)
  {
     x <- rnorm(20)
     y <- rnorm(20)

     pvalue[i] <- perm.cor.test(x,y,no.perm)
  }

  pvalue
}
```

With `no.perm=100`, the histogram of 2000 p-values is shown:

Example:(power regression) In practice, people tend to use some "preprocessing" procedure to exacerbate the evidence against the hypothesis (in order to for example get their paper published). The preprocessing procedure may be very complicated and then hide this tendency. We will talk about such an example.

When the relationship of a response variable, y, to a predictor variable, x, appears to be non-linear, it

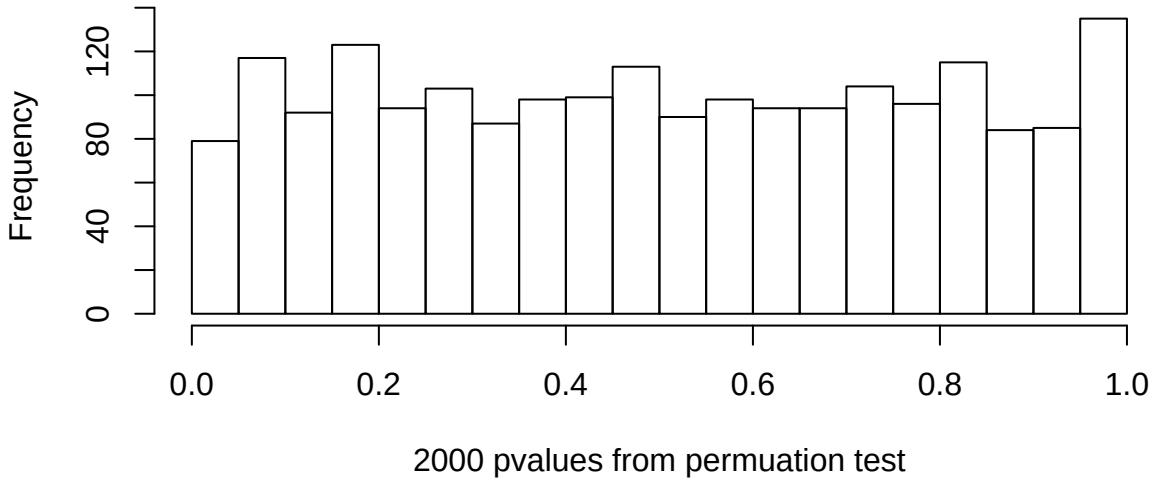


Figure 6.7: Histogram of 2000 p-values by permutation test on correlation

is common to consider power transformations of x, in an attempt to find a model that fits better. One might, for instance, attempt to model y as $a + b\,x^2 + e$, for some values of the regression coefficients a and b, with e being the random residual. Or if the data looks different, one might instead model y in terms of $x^{1/2}$. In this context, using the p-value for the regression that was done as an indication of the strength of evidence against the null hypothesis of no relationship would be naive, as it fails to account for the fact that the transformation of x was based on an examination of the data. However, one can obtain a valid p-value from a permutation test.

We do the experiments using the following R functions:

```
# REGRESSION USING BEST POWER TRANSFORMATION.  Takes as arguments a
# vector of predictor values and a vector of response values (of equal
# length).  The predictor values must be non-negative.  Finds the
# power transformation for the predictors that produces the highest
# correlation (in absolute value) with the response, chosing the power
# from the powers argument (which defaults to seq(0.1,2.0,by=0.1).
# Returns the pvalue

pvalue.lm.pow <- function (x, y, powers=seq(0.1,2.0,by=0.1))
{
  if (!is.vector(x,"numeric") || !is.vector(y,"numeric")
   || length(x)!=length(y))
  { stop("Arguments should be numeric vectors of equal length")
  }
```

```
  if (any(x<0))
  { stop("Predictors must be non-negative")
  }

  n <- length(x)

  # Find the power that produces the highest correlation with the response.

  best.r <- 0

  for (p in powers)
  { r <- cor(x^p,y)
    if (abs(r)>=abs(best.r))
    { power <- p
      best.r <- r
    }
  }

  # Return the best power and the linear model using that power.

  xp <- x^power
  #return naive pvalue
  coef(summary(lm(y~xp)))[2,4]
}

# TEST VALIDITY OF THE NAIVE P-VALUES.  Simulates the results of
# naively interpreting pvalues for the regression on the best
# power of the predictor variable (from pvalue.lm.pow) as a real pvalue.
# The arguments of this function are the vector of predictor variables
# to use (x), the number of datasets to simulate (N), and possible
# further arguments that are passed on to pvalue.lm.pow.  N datasets of n
# cases are generated in which x is as specified and the corresponding
# y values are generated independently from the standard normal
# distribution (without reference to x).  The result is the vector of
# N p-values obtained from the models that pvalue.lm.pow chooses for these
# datasets.  Since the null hypothesis of no relationship is true,
# these p-values should be uniformly distributed between 0 and 1, if
# they are valid.

test.lm.power <- function (x, N, ...)
{
  n <- length(x)

  # Simulate N datasets and record the naive p-value found for each.
```

```
  pvalues <- rep(0,N)

  for (k in 1:N)
  {
    y <- rnorm(n)

    pvalues[k] <- pvalue.lm.pow(x,y,...)
  }

  # Return the vector of N p-values that were obtained.

  pvalues
}

# FIND PERMUTATION PVALUE FOR REGRESSION USING BEST POWER TRANSFORM.
# Takes as arguments the vectors of predictors (x) and responses (y),
# as for pvalue.lm.pow, the number of permutations to use in finding the
# p-value (default is 999), and possible further arguments that are
# passed on to pvalue.lm.pow.  Returns the p-value from the permutation
# test, which is (roughly) the fraction of times that reg.pow applied
# to a randomly shuffled data sets gives a smaller p-value than reg.pow
# gives when applied to the actual dataset.  In detail, the pvalue is
# (count+1)/(perms+1), with count being the number of smaller p-values
# obtained from permuted datasets.  If the null hypothesis of no
# relationship is true, these p-values will be uniformly distributed
# over the possible values (ie, roughly uniform between 0 and 1).

pvalue.perm.lm.pow <- function (x, y, perms=999, ...)
{
  # Find the naive p-value using pvalue.lm.pow for the actual data.

  actual <- pvalue.lm.pow(x,y,...)

  # Count how many times pvalue.lm.pow applied to a random permutation finds
  # a model with as small a naive p-value as for the actual data.

  count <- 0

  for (k in 1:perms)
  { pvalue <- pvalue.lm.pow(x,sample(y),...)
    if (pvalue<=actual)
    { count <- count + 1
    }
  }

  # Return the p-value from the permutation test, based on the count found.
```

```
  (count+1) / (perms+1)
}

# TEST VALIDITY OF THE PERMUATION PVALUES. The method is the same as for
#test.lm.power. When power.sim is not 0, it can be used to calculate the power
#of the test

test.perm.power <- function (x, power.sim, N=100, resid.std.dev=1, ...)
{
  n <- length(x)

  pvalues <- rep(0,N)

  for (k in 1:N)
  { y <- x^power.sim + rnorm(n,0,resid.std.dev)
    pvalues[k] <- pvalue.perm.lm.pow(x,y,...)
  }
  pvalues
}
```

The histograms of 1000 p-values produced by null hypothesis using both naive and permutation methods and those produced by alternative (power=1.5, 0.5) using permutation test:

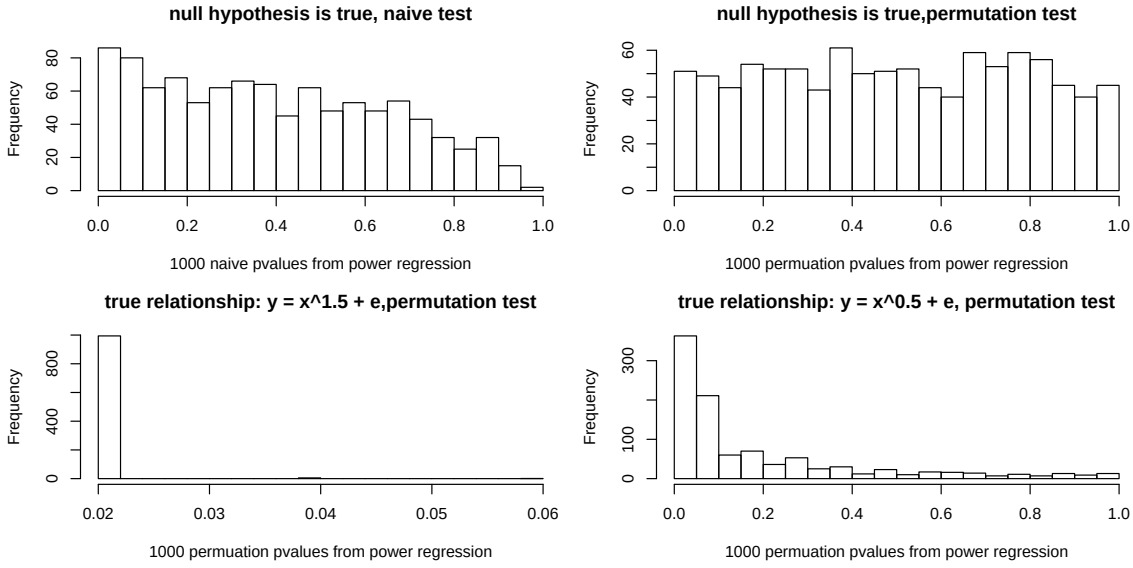


Figure 6.8: Histogram of 1000 p-values for power regression example

Chapter 7

Numerical methods for maximizing likelihood functions

7.1 Review of maximum likelihood estimation (MLE)

If we have specified a parametric statistical models for data $\mathcal{D}$ as follows:

$$f(\mathcal{D}|\theta), \tag{7.1}$$

we can estimate the unknown θ using the value that appear most likely to have generated the data $\mathcal{D}$. Formally, $\hat{\theta}_{\text{mle}}$ is defined as:

$$\hat{\theta}_{\text{mle}} = \arg\max_{\theta} \ \log(f(\mathcal{D}|\theta)) \tag{7.2}$$

The function $\log(f(\mathcal{D}|\theta))$ is usually called **log-likelihood function**.

This is a generally applicable estimation procedure and therefore used very extensively. For most practical problems, there isn't closed-form for $\hat{\theta}_{\text{mle}}$. We will introduce some numerical methods. Prior to the algorithmic discussion, we give it a brief review, illustrating and demonstrating its correctnesss.

We first look at a simple example.

Example: Suppose $X_1, \ldots, X_n$ are iid from Cauchy(θ,1), with the following density:

$$f(x_1, \ldots, x_n|\theta) = \prod_{i=1}^{n} \frac{1}{1 + (x_i - \theta)^2} \tag{7.3}$$

The MLE for θ in (7.3):

$$\hat{\theta}_{\text{mle}} = \arg\max_{\theta} \sum_{i=1}^{n} -\log(1 + (x_i - \theta)^2) \tag{7.4}$$

In order to demonstrate MLE, we simulate this procedure a certain number of times to see what is going on. We use a value, denoted as θ_*, to generate data and then draw the log-likelihood functions based on each simulated data sets. We do this simulation for various n. The following is the R function and R codes for performing a simulation:

```
sim_loglike_cc <- function(no_sim,n,true_theta=0)
{
    thetas <- seq(true_theta-5,true_theta+5,length=100)

    loglikes <- matrix(0,no_sim,100)

    for(i in 1:no_sim) {
        x <- rcauchy(n,true_theta)
        loglike_at_true <-   mean(dcauchy(x,true_theta,log=TRUE))
        loglikes[i,] <- sapply(thetas, function(theta) mean(dcauchy(x,theta,log=TRUE)) )   -
                            loglike_at_true
    }

    plot(thetas,loglikes[1,],xlab='theta',ylab='log likelihood',type='l',lty=2,
         ylim=c(min(loglikes),max(loglikes)),
         main=paste('sample size =',n) )
    abline(v=true_theta)

    for(i in seq(2,no_sim,by=1)) lines(thetas,loglikes[i,],lty=2)

}

postscript('loglike-cc.eps',paper='special',height=6,width=7,horiz=FALSE)

par(mfrow=c(2,2),mar=c(4,4,2,1))

sim_loglike_cc(10,2,0)

sim_loglike_cc(10,20,0)

sim_loglike_cc(10,200,0)

sim_loglike_cc(10,2000,0)

dev.off()
```

The results are shown by Figure 7.1

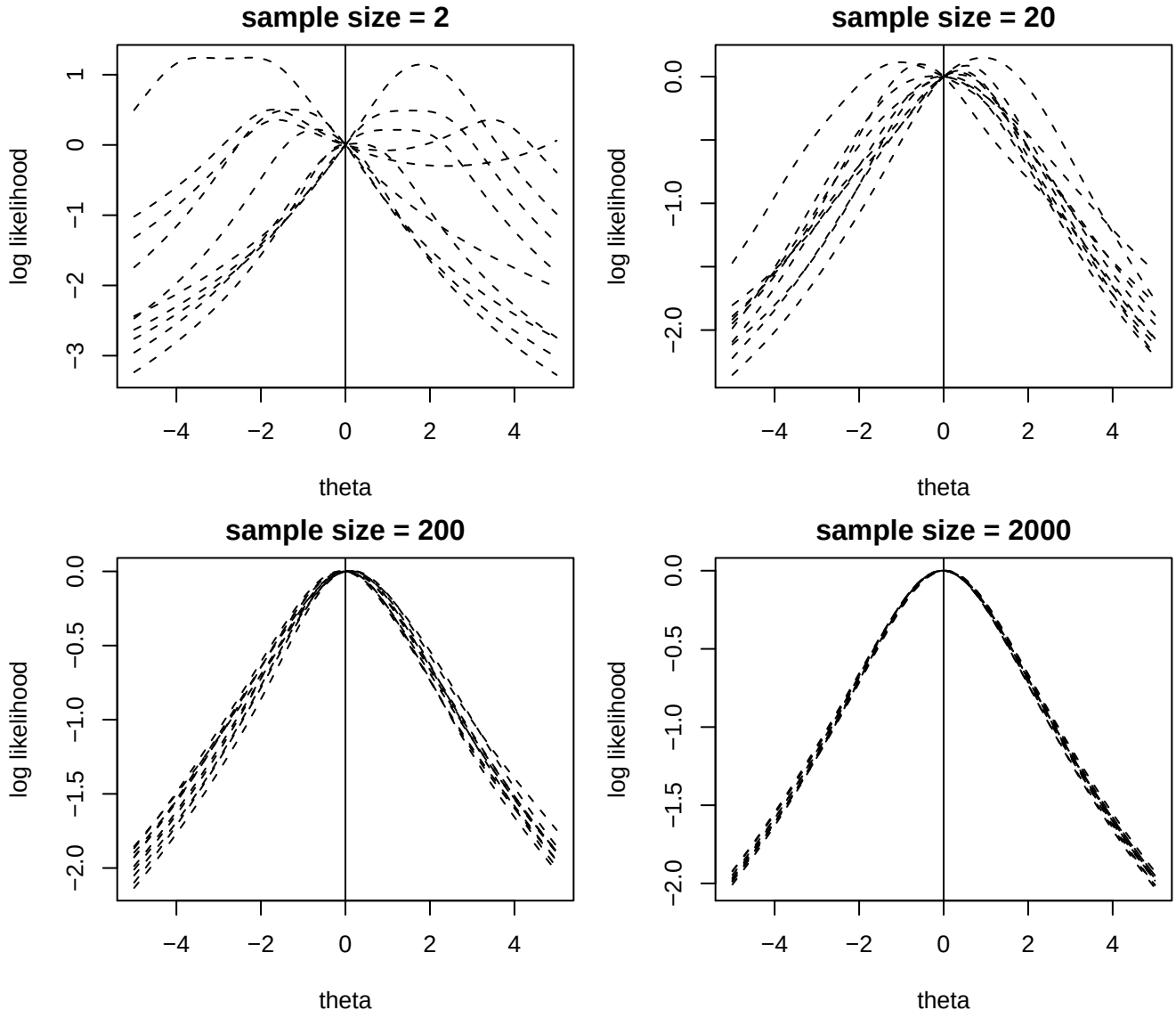


Figure 7.1: Scaled log likelihood functions of Cauchy distribution with $\theta_* = 0$, based on simulated data sets.

Let's show $\hat{\theta}_{\text{mle}}$ will converge to θ_* when n goes to ∞ (when $\mathcal{D}$ is iid samples $X_1, \ldots, X_n$): It is easy to see that maximizing log likelihood function $\log(f(\mathcal{D}|\theta))$ is equivalent to maximizing:

$$M_n(\theta) = \frac{1}{n} \sum_i \log \frac{f(X_i|\theta)}{f(X_i|\theta_*)} \tag{7.5}$$

By law of large number, for each θ, $M_n(\theta)$ will converge to

$$\begin{aligned} E\left(\log \frac{f(X_i|\theta)}{f(X_i|\theta_*)}\right) &= \int \log \frac{f(x|\theta)}{f(x|\theta_*)} f(x|\theta_*)\, dx & (7.6)\\ &\leq \int \left(\frac{f(x|\theta)}{f(x|\theta_*)} - 1\right) f(x|\theta_*)\, dx & (7.7)\\ &= 0 & (7.8) \end{aligned}$$

Under some regularity conditions (satisfied by most statistical models used in practice), the MLE $\hat{\theta}_{\text{mle}}$ for iid samples has the following properties:

1. The MLE is consistent: $\hat{\theta}_{\text{mle}} \to \theta_*$ when $n \to \infty$

2. The MLE is asymptotically normal:

$$\hat{\theta}_{\text{mle}} \to N(\theta_*, (n\, I(\theta_*))^{-1}) \tag{7.9}$$

   where

$$I(\theta_*) = -E\left(\frac{\Delta^2}{\Delta\, \theta^2} \log(f(X|\theta))\Big|_{\theta_*}\right) = E\left(\left(\frac{d^2}{d\theta_i d\theta_j} \log(f(X|\theta))\right)_{p\times p}\right) \tag{7.10}$$

3. Plugin $\hat{\theta}_{\text{mle}}$ in $I(\theta_*)$, the fact still holds:

$$\hat{\theta}_{\text{mle}} \to N(\theta_*, (n\, I(\hat{\theta}_{\text{mle}}))^{-1}) \tag{7.11}$$

   $I(\hat{\theta}_{\text{mle}})$ is called the **expected fisher information**. In contrast, $I(\theta_*)$ can also be estimated by:

$$-\frac{1}{n} \sum_i \frac{\Delta^2}{\Delta\, \theta^2} \log(f(X_i|\theta))\Big|_{\hat{\theta}_{\text{mle}}} \tag{7.12}$$

   This is called **the observed fisher information**, n times of which is just the minus of the second derivative of the log likelihood function at $\hat{\theta}_{\text{mle}}$. The obvious advantage of this estimation of standard deviation is that we do not need to compute the expected value in (7.10), which might be difficult or impossible.

4. The MLE is asymptotically optimal.

7.2 Univariate problems

7.2.1 Root finding algorithms

We may find $\hat{\theta}_{\text{mle}}$ by looking for root of the derivative of log likelihood function:

$$\frac{d \log(\mathcal{D}|\theta)}{d\theta} = 0. \tag{7.13}$$

Simple iteration (fixed-point method)

If the equation we want to solve can be written as

$$x = f(x), \tag{7.14}$$

we can use simple iteration method. The algorithm starts with a rough guess x_0, then iterates as follows:

$$x_n = f(x_{n-1}) \tag{7.15}$$

It is illustrated by Figure 7.2.

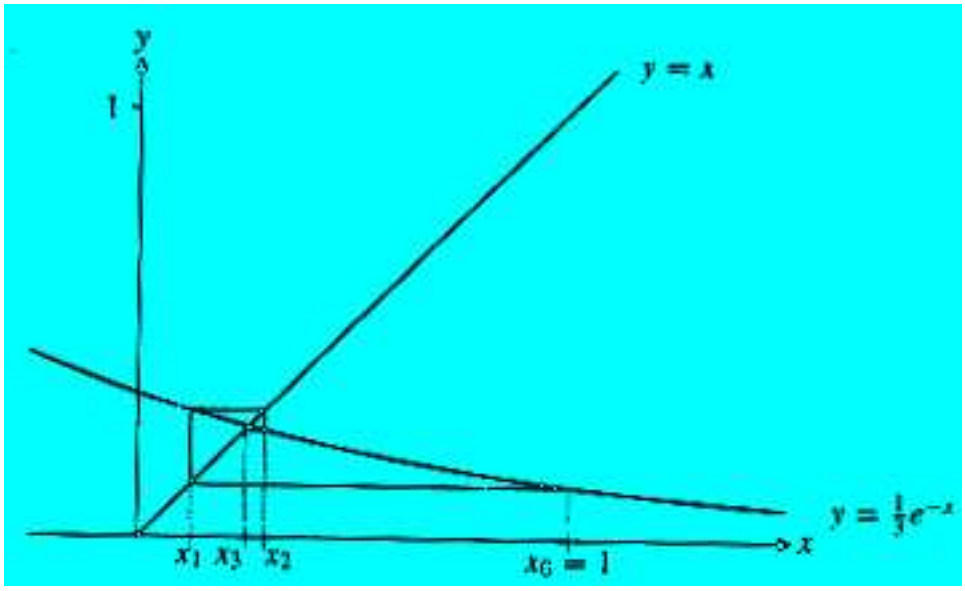


Figure 7.2: Illustration of simple iteration

Let's look at some animation.

Bisection method

The bisection method is a root-finding algorithm which works by repeatedly dividing an interval in half and then selecting the subinterval in which a root exists.

Suppose we want to solve the equation $f(x) = 0$. Given two points a and b such that $f(a)$ and $f(b)$ have opposite signs, we know by the intermediate value theorem that f must have at least one root in the interval $[a, b]$ as long as f is continuous on this interval. The bisection method divides the interval in two by computing $c = (a + b)/2$. There are now two possibilities: either $f(a)$ and $f(c)$ have opposite signs, or $f(c)$ and $f(b)$ have opposite signs. The bisection algorithm is then applied recursively to the sub-interval where the sign change occurs.

If f is a continuous function on the interval $[a, b]$ and $f(a)f(b) < 0$, then the bisection method converges to a root of f. In fact, the absolute error is halved at each step. After n steps, the maximum absolute error is

$$\frac{|b - a|}{2^{n+1}} \tag{7.16}$$

The bisection method is less efficient than Newton's method but it is much less prone to odd behavior.The method converges linearly, which is quite slow. On the positive side, the method is guaranteed to converge if $f(a)$ and $f(b)$ have different sign.

This method is illustrated by Figure 7.3.

Let's look at some animation.

Newton-Raphson method

The idea of the method is as follows: one starts with an initial guess which is reasonably close to the true root, then the function is approximated by its tangent line (which can be computed using the tools of calculus), and one computes the x-intercept of this tangent line (which is easily done with elementary algebra). This x-intercept will typically be a better approximation to the function's root than the original guess, and the method can be iterated.

Suppose $f : [a, b] \to R$ is a differentiable function defined on the interval $[a, b]$ with values in the real numbers R. The formula for converging on the root can be easily derived. Suppose we have some current approximationx_n. Then we can derive the formula for a better approximation, x_{n+1} by referring to the diagram on the right. We know from the definition of the derivative at a given point that it is the slope of a tangent at that point.

$$f'(x_n) = \frac{\Delta \mathrm{y}}{\Delta \mathrm{x}} = \frac{f(x_n) - 0}{x_n - x_{n+1}} = \frac{0 - f(x_n)}{x_{n+1} - x_n} \tag{7.17}$$

Let's look at an animation.

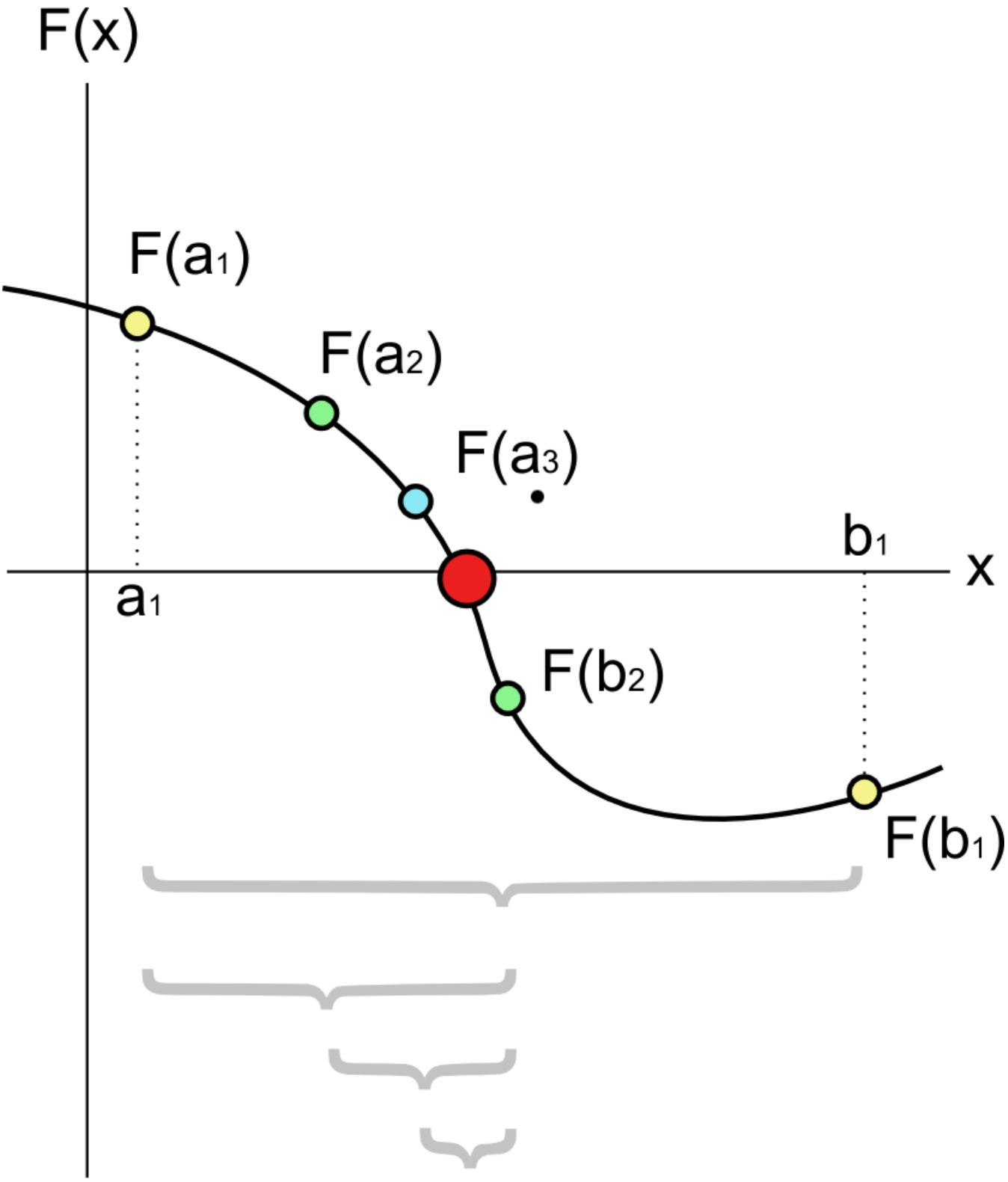


Figure 7.3: Bisection method for solving $F(x) = 0$

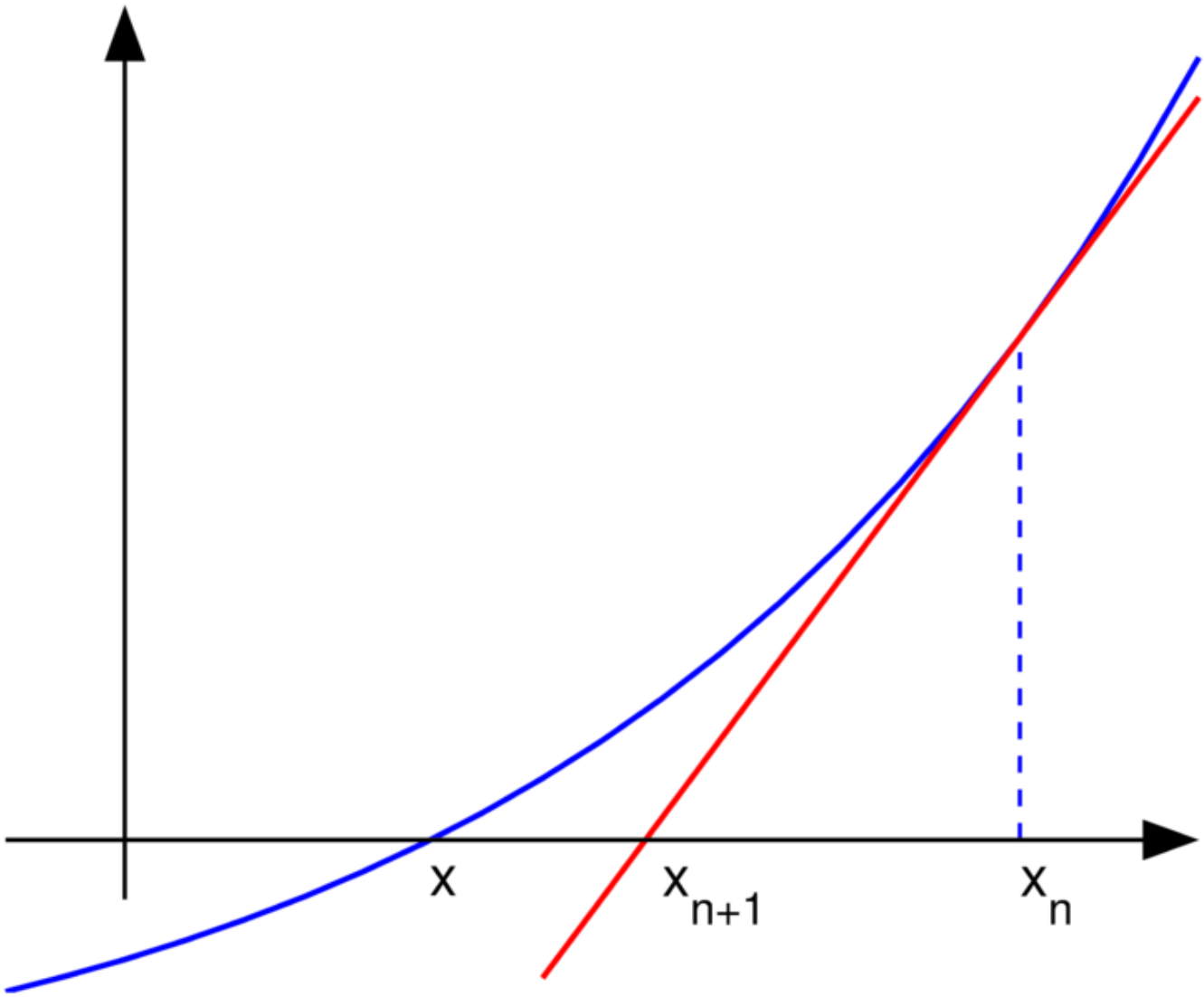


Figure 7.4: An illustration of one iteration of Newton's method (the function f is shown in blue and the tangent line is in red). We see that x_{n+1} is a better approximation than x_n for the root x of the function f.

Secant method

The secant method is defined by the recurrence relation

$$x_{n+1} = x_n - \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} f(x_n). \tag{7.18}$$

As can be seen from the recurrence relation, the secant method requires two initial values, x_0 and x_1, which should ideally be chosen to lie close to the root.

Given x_{n1} and x_n, we construct the line through the points $(x_{n1}, f(x_{n1}))$ and $(x_n, f(x_n))$, as demonstrated in the picture on the right. Note that this line is a secant or chord of the graph of the function f. In point-slope form, it can be defined as

$$y - f(x_n) = \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}} (x - x_n). \tag{7.19}$$

We now choose x_{n+1} to be the root of this line, so x_{n+1} is chosen such that

$$f(x_n) + \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}} (x_{n+1} - x_n) = 0. \tag{7.20}$$

Solving this equation gives the recurrence relation for the secant method.

The iterates xn of the secant method converge to a root of f, if the initial values x0 and x1 are sufficiently close to the root. The order of convergence is 1.618. This result only holds under some technical conditions, namely that f be twice continuously differentiable and the root in question be simple (i.e., that it not be a repeated root). If the initial values are not close to the root, then there is no guarantee that the secant method converges.

The recurrence formula of the secant method can be derived from the formula for Newton's method

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{7.21}$$

by using the finite difference approximation

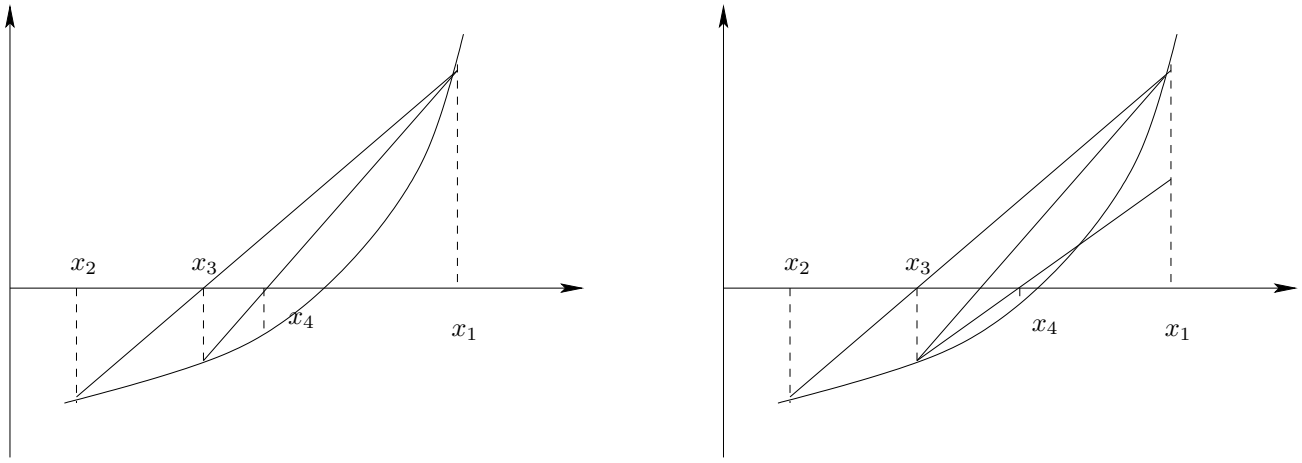
$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}. \tag{7.22}$$

If we compare Newton's method with the secant method, we see that Newton's method converges faster (order 2 against 1.618). However, Newton's method requires the evaluation of both f and its derivative at every step, while the secant method only requires the evaluation of f. Therefore, the secant method

may well be faster in practice.

Let's look at some animation.

Illinois method



Secant method Illinois method

Figure 7.5: Illinois method

The order of convergence of illinois method is 1.442. But it is safer.

Fisher's scoring method

Used only in statistics. Replacing $-f'(\theta_m)$ (derivative of score function) with the **expected fisher information** $n\,I(\theta_m)$.

$$\theta_{m+1} = \theta_m + (n\,I(\theta_m))^{-1}\,s(\theta_m) \tag{7.23}$$

A demonstrating example

Suppose we only observe nonzero iid samples $X_1, \ldots, X_n$ from a poission distribution, whose probability function is:

$$P(x_1, \ldots, x_n|\theta) = \prod_{i=1}^{n} \frac{\exp(-\lambda)\lambda^{x_i}}{x_i!} \frac{1}{1-\exp(-\lambda)} \tag{7.24}$$

The log likelihood function is

$$l(\lambda) = -\lambda n + \sum x_i \log(\lambda) - n \log(1 - \exp(-\lambda)) \tag{7.25}$$

The score function $s(\lambda) = l'(\lambda)$:

$$s(\lambda) = \frac{\sum x_i}{\lambda} - \frac{n}{1 - \exp(-\lambda)} \tag{7.26}$$

The equation $s(\lambda) = 0$ is equivalent to:

$$\lambda = \bar{x}\,(1 - \exp(-\lambda)) \tag{7.27}$$

The derivative of score function:

$$s'(\lambda) = -\frac{\sum x_i}{\lambda^2} + n\,\frac{\exp(-\lambda)}{(1 - \exp(-\lambda))^2} \tag{7.28}$$

The fisher information based on n sample is:

$$n\,I(\lambda) = -E(s'(\lambda)) \tag{7.29}$$

$$= \frac{1}{\lambda\,(1 - \exp(-\lambda))} - n\,\frac{\exp(-\lambda)}{(1 - \exp(-\lambda))^2} \tag{7.30}$$

```
# MAXIMUM LIKELIHOOD ESTIMATION FOR POISSON DATA WITH ZEROS UNOBSERVED.
#
# For all three functions,
# the arguments are the vector of observations, the number of iterations to
# do, and the initial guess (defaulting to the sample mean).  The result is
# a data frame with one row for each iteration (including the initial guess),
# with the columns being the estimate for lambda at that iteration and the
# log likelihood for that value of lambda (minus the factorial terms that
# don't involve lambda).


# COMPUTE LOG LIKELIHOOD.  The arguments are the data vector and a value for
# lambda (or vector of values).  The result is the log probability of the data
# given that value for lambda, omitting the factorial terms (or a vector of
# log probabilities if lambda is a vector).

nzp.log.likelihood <- function (n, lambda)
{
  sum(n) * log(lambda) - length(n) * (lambda + log(1-exp(-lambda)))
}
```

```
# FIND MLE BY SIMPLE ITERATION.

nzp.simple.iteration <- function (n, r, lambda0=mean(n))
{
  mean.n <- mean(n)

  lambda <- rep(0,r+1)
  lambda[1] <- lambda0

  for (i in 1:r)
  { lambda[i+1] <- mean.n * (1 - exp(-lambda[i]))
  }

  data.frame (lambda=lambda, log.lik=nzp.log.likelihood(n,lambda))
}


# FIND MLE BY NEWTON-RAPHSON ITERATION.

nzp.newton.raphson <- function (n, r, lambda0=mean(n))
{
  mean.n <- mean(n)

  lambda <- rep(0,r+1)
  lambda[1] <- lambda0

  for (i in 1:r)
  { e <- exp(-lambda[i])
    lambda[i+1] <- lambda[i] -
       (mean.n/lambda[i] - 1/(1-e)) / (e/(1-e)^2 - mean.n/lambda[i]^2)
  }

  data.frame (lambda=lambda, log.lik=nzp.log.likelihood(n,lambda))
}


# FIND MLE BY THE METHOD OF SCORING.

nzp.method.of.scoring <- function (n, r, lambda0=mean(n))
{
  mean.n <- mean(n)

  lambda <- rep(0,r+1)
  lambda[1] <- lambda0
```

```
  for (i in 1:r)
  { e <- exp(-lambda[i])
    lambda[i+1] <- lambda[i] -
       (mean.n/lambda[i] - 1/(1-e)) / ((e/(1-e) - 1/lambda[i]) / (1-e))
  }

  data.frame (lambda=lambda, log.lik=nzp.log.likelihood(n,lambda))
}
```

The results of some experiments:

```
TESTS OF FUNCTIONS TO FIND THE MLE FOR THE POISSON MODEL WITHOUT ZEROS.

> n<-c(1,2,1,1,3)

> nzp.simple.iteration(n,15)
     lambda   log.lik
1  1.600000 -3.112386
2  1.276966 -2.794171
3  1.153788 -2.729644
4  1.095297 -2.712646
5  1.064895 -2.707666
6  1.048378 -2.706129
7  1.039191 -2.705642
8  1.034015 -2.705485
9  1.031077 -2.705434
10 1.029404 -2.705418
11 1.028448 -2.705412
12 1.027902 -2.705411
13 1.027589 -2.705410
14 1.027410 -2.705410
15 1.027308 -2.705410
16 1.027249 -2.705410

> nzp.simple.iteration(n,15,0.1)
      lambda   log.lik
1  0.1000000 -7.159838
2  0.1522601 -6.031974
3  0.2259762 -5.037551
4  0.3236207 -4.215401
5  0.4423606 -3.593492
6  0.5719713 -3.173899
7  0.6969413 -2.926302
8  0.8030295 -2.799296
9  0.8832484 -2.742050
```

```
10 0.9384996 -2.718835
11 0.9740570 -2.710123
12 0.9959229 -2.707020
13 1.0089882 -2.705951
14 1.0166597 -2.705590
15 1.0211177 -2.705469
16 1.0236926 -2.705429

> nzp.simple.iteration(n,15,10)
      lambda    log.lik
1  10.000000 -31.579092
2   1.599927  -3.112294
3   1.276942  -2.794155
4   1.153778  -2.729641
5   1.095292  -2.712645
6   1.064893  -2.707666
7   1.048376  -2.706129
8   1.039190  -2.705642
9   1.034014  -2.705485
10  1.031077  -2.705434
11  1.029404  -2.705418
12  1.028448  -2.705412
13  1.027902  -2.705411
14  1.027589  -2.705410
15  1.027410  -2.705410
16  1.027308  -2.705410

> nzp.newton.raphson(n,15)
      lambda   log.lik
1  1.6000000 -3.112386
2  0.7787627 -2.822769
3  0.9714403 -2.710607
4  1.0244661 -2.705422
5  1.0271638 -2.705410
6  1.0271701 -2.705410
7  1.0271701 -2.705410
8  1.0271701 -2.705410
9  1.0271701 -2.705410
10 1.0271701 -2.705410
11 1.0271701 -2.705410
12 1.0271701 -2.705410
13 1.0271701 -2.705410
14 1.0271701 -2.705410
15 1.0271701 -2.705410
16 1.0271701 -2.705410
```

```
> nzp.newton.raphson(n,15,0.1)
      lambda   log.lik
1  0.1000000 -7.159838
2  0.1914009 -5.446287
3  0.3504920 -4.047049
4  0.5886078 -3.133477
5  0.8480668 -2.763504
6  0.9985621 -2.706757
7  1.0264615 -2.705411
8  1.0271697 -2.705410
9  1.0271701 -2.705410
10 1.0271701 -2.705410
11 1.0271701 -2.705410
12 1.0271701 -2.705410
13 1.0271701 -2.705410
14 1.0271701 -2.705410
15 1.0271701 -2.705410
16 1.0271701 -2.705410

> nzp.newton.raphson(n,15,10)
        lambda   log.lik
1     10.00000 -31.57909
2    -42.65225       NaN
3    -85.30450       NaN
4   -170.60901       NaN
5   -341.21801       NaN
6   -682.43603       NaN
7  -1364.87206       NaN
8          NaN       NaN
9          NaN       NaN
10         NaN       NaN
11         NaN       NaN
12         NaN       NaN
13         NaN       NaN
14         NaN       NaN
15         NaN       NaN
16         NaN       NaN
Warning messages:
1: NaNs produced in: log(x)
2: NaNs produced in: log(x)

> nzp.method.of.scoring(n,15)
     lambda   log.lik
1  1.600000 -3.112386
2  1.057311 -2.706856
3  1.027271 -2.705410
```

```
4  1.027170 -2.705410
5  1.027170 -2.705410
6  1.027170 -2.705410
7  1.027170 -2.705410
8  1.027170 -2.705410
9  1.027170 -2.705410
10 1.027170 -2.705410
11 1.027170 -2.705410
12 1.027170 -2.705410
13 1.027170 -2.705410
14 1.027170 -2.705410
15 1.027170 -2.705410
16 1.027170 -2.705410

> nzp.method.of.scoring(n,15,0.1)
     lambda   log.lik
1  0.100000 -7.159838
2  1.162915 -2.733133
3  1.029150 -2.705416
4  1.027171 -2.705410
5  1.027170 -2.705410
6  1.027170 -2.705410
7  1.027170 -2.705410
8  1.027170 -2.705410
9  1.027170 -2.705410
10 1.027170 -2.705410
11 1.027170 -2.705410
12 1.027170 -2.705410
13 1.027170 -2.705410
14 1.027170 -2.705410
15 1.027170 -2.705410
16 1.027170 -2.705410

> nzp.method.of.scoring(n,15,10)
      lambda    log.lik
1  10.000000 -31.579092
2   1.596112  -3.107480
3   1.056944  -2.706821
4   1.027269  -2.705410
5   1.027170  -2.705410
6   1.027170  -2.705410
7   1.027170  -2.705410
8   1.027170  -2.705410
9   1.027170  -2.705410
10  1.027170  -2.705410
11  1.027170  -2.705410
```

```
12  1.027170  -2.705410
13  1.027170  -2.705410
14  1.027170  -2.705410
15  1.027170  -2.705410
16  1.027170  -2.705410

> n<-c(1,2,1,1,1)

> nzp.simple.iteration(n,15)
      lambda   log.lik
1  1.2000000 -3.114159
2  0.8385669 -2.418128
3  0.6812044 -2.183208
4  0.5927914 -2.077887
5  0.5366615 -2.023899
6  0.4983636 -1.993971
7  0.4709712 -1.976512
8  0.4507253 -1.965963
9  0.4354010 -1.959423
10 0.4235938 -1.955291
11 0.4143723 -1.952642
12 0.4070941 -1.950925
13 0.4013021 -1.949802
14 0.3966627 -1.949062
15 0.3929270 -1.948573
16 0.3899063 -1.948247

> nzp.simple.iteration(n,15,0.1)
      lambda   log.lik
1  0.1000000 -2.554668
2  0.1141951 -2.458051
3  0.1294993 -2.371321
4  0.1457578 -2.294629
5  0.1627595 -2.227898
6  0.1802454 -2.170816
7  0.1979217 -2.122846
8  0.2154791 -2.083259
9  0.2326138 -2.051178
10 0.2490486 -2.025644
11 0.2645495 -2.005673
12 0.2789380 -1.990310
13 0.2920958 -1.978675
14 0.3039636 -1.969990
15 0.3145347 -1.963591
16 0.3238458 -1.958932
```

```
> nzp.simple.iteration(n,15,10)
       lambda    log.lik
1  10.0000000 -36.184262
2   1.1999455  -3.114041
3   0.8385473  -2.418096
4   0.6811942  -2.183194
5   0.5927852  -2.077880
6   0.5366574  -2.023896
7   0.4983607  -1.993969
8   0.4709691  -1.976511
9   0.4507237  -1.965962
10  0.4353998  -1.959422
11  0.4235929  -1.955290
12  0.4143715  -1.952642
13  0.4070935  -1.950925
14  0.4013017  -1.949802
15  0.3966623  -1.949062
16  0.3929267  -1.948573

> nzp.newton.raphson(n,15)
         lambda   log.lik
1     1.2000000 -3.114159
2    -0.7903745       NaN
3    -2.5059333       NaN
4    -6.6397071       NaN
5   -13.5648162       NaN
6   -27.1321084       NaN
7   -54.2642168       NaN
8  -108.5284336       NaN
9  -217.0568672       NaN
10 -434.1137344       NaN
11 -868.2274687       NaN
12          NaN       NaN
13          NaN       NaN
14          NaN       NaN
15          NaN       NaN
16          NaN       NaN
Warning messages:
1: NaNs produced in: log(x)
2: NaNs produced in: log(x)

> nzp.newton.raphson(n,15,0.1)
      lambda   log.lik
1  0.1000000 -2.554668
2  0.1742741 -2.189137
3  0.2692156 -2.000372
```

```
4  0.3467844 -1.951042
5  0.3742121 -1.947603
6  0.3764256 -1.947584
7  0.3764380 -1.947584
8  0.3764380 -1.947584
9  0.3764380 -1.947584
10 0.3764380 -1.947584
11 0.3764380 -1.947584
12 0.3764380 -1.947584
13 0.3764380 -1.947584
14 0.3764380 -1.947584
15 0.3764380 -1.947584
16 0.3764380 -1.947584

> nzp.newton.raphson(n,15,10)
        lambda   log.lik
1     10.00000 -36.18426
2    -63.61565       NaN
3   -127.23131       NaN
4   -254.46262       NaN
5   -508.92523       NaN
6  -1017.85047       NaN
7          NaN       NaN
8          NaN       NaN
9          NaN       NaN
10         NaN       NaN
11         NaN       NaN
12         NaN       NaN
13         NaN       NaN
14         NaN       NaN
15         NaN       NaN
16         NaN       NaN
Warning messages:
1: NaNs produced in: log(x)
2: NaNs produced in: log(x)
> nzp.method.of.scoring(n,15)
      lambda   log.lik
1  1.2000000 -3.114159
2  0.4513577 -1.966260
3  0.3772372 -1.947587
4  0.3764381 -1.947584
5  0.3764380 -1.947584
6  0.3764380 -1.947584
7  0.3764380 -1.947584
8  0.3764380 -1.947584
9  0.3764380 -1.947584
```

```
10 0.3764380 -1.947584
11 0.3764380 -1.947584
12 0.3764380 -1.947584
13 0.3764380 -1.947584
14 0.3764380 -1.947584
15 0.3764380 -1.947584
16 0.3764380 -1.947584

> nzp.method.of.scoring(n,15,0.1)
      lambda   log.lik
1  0.1000000 -2.554668
2  0.3887130 -1.948136
3  0.3764599 -1.947584
4  0.3764380 -1.947584
5  0.3764380 -1.947584
6  0.3764380 -1.947584
7  0.3764380 -1.947584
8  0.3764380 -1.947584
9  0.3764380 -1.947584
10 0.3764380 -1.947584
11 0.3764380 -1.947584
12 0.3764380 -1.947584
13 0.3764380 -1.947584
14 0.3764380 -1.947584
15 0.3764380 -1.947584
16 0.3764380 -1.947584

> nzp.method.of.scoring(n,15,10)
       lambda    log.lik
1  10.0000000 -36.184262
2   1.1959483  -3.105436
3   0.4507237  -1.965962
4   0.3772239  -1.947586
5   0.3764381  -1.947584
6   0.3764380  -1.947584
7   0.3764380  -1.947584
8   0.3764380  -1.947584
9   0.3764380  -1.947584
10  0.3764380  -1.947584
11  0.3764380  -1.947584
12  0.3764380  -1.947584
13  0.3764380  -1.947584
14  0.3764380  -1.947584
15  0.3764380  -1.947584
16  0.3764380  -1.947584
```

7.2.2 Golden sector method

Finding root of the derivative of a function does not guarantee finding the miximizer. There is method called "golden sector" search.

Let $\alpha = 1 - 0.618$. In the golden section search, given an interval $[x_1, x_3]$ thought to contain a minimum, initially interior points are located at $x_0 = x_1 + \alpha(x_3 - x_1)$ and $x_2 = x_3 - \alpha(x_3 - x_1)$. Then $f(x_0)$ and $f(x_2)$ are evaluated. If $f(x_0) < f(x_2)$ then the new interval is $[x_1, x_2]$ and the next point added is $x_1 + \alpha(x_2 x_1)$, and if $f(x_0) > f(x_2)$ then the new interval is $[x0, x3]$ and the next point added is $x_3 - \alpha(x_3 - x_0)$. The algorithm continues in this fashion until the width of the interval determines the solution to sucient precision. Note that it is not necessary to evaluate f at the endpoints of the initial interval, and if this interval turns out not to contain a local minimum, then the algorithm will converge to one of the endpoints.

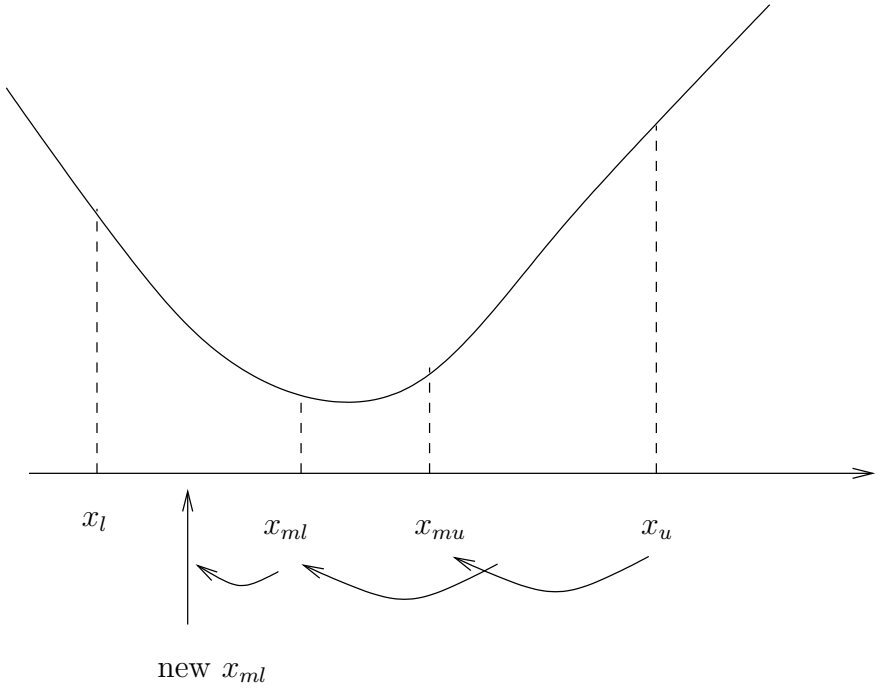


Figure 7.6: Golden sector search

A function implementing the golden section search is given below. Here f is a function whose first argument is the variable of the minimization (x above), `brack.int` is the bracketing interval (a vector of length 2), and ... are additional arguments to f. The iteration terminates when the relative length of the interval is $<$ `eps`.

```
golden <- function(f,brack.int,eps=1e-4,...) {
 g <- (3-sqrt(5))/2
 xl <- min(brack.int)
```

```
  xu <- max(brack.int)
  tmp <- g*(xu-xl)
  xmu <- xu-tmp
  xml <- xl+tmp
  fl <- f(xml,...)
  fu <- f(xmu,...)
  while(abs(xu-xl)>(1.e-5+abs(xl))*eps) {
     if (fl<fu) {
       xu <- xmu
       xmu <- xml
       fu <- fl
       xml <- xl+g*(xu-xl)
       fl <- f(xml,...)
     } else {
       xl <- xml
       xml <- xmu
       fl <- fu
       xmu <- xu-g*(xu-xl)
       fu <- f(xmu,...)
     }
   }
   if (fl<fu) xml else xmu
}
```

To illustrate the use of this function, the multinomial log likelihood

$$1997\log(2+\theta) + 1810\log(1-\theta) + 32\log(\theta), \tag{7.31}$$

where $0 < \theta < 1$, will be maximized. The probabilities of the three categories are actually $p_1 = (2+\theta)/4, p_2 = (1-\theta)/2$ and $p_3 = \theta/4$, but the constant divisors have been dropped from the likelihood. Since the function is designed to locate a minimum, the function f evaluates the negative of the log likelihood.

```
> f <- function(theta) -1997*log(2+theta)-1810*log(1-theta)-32*log(theta)
> golden(f,c(.001,.999))
[1] 0.03571247
```

7.3 Descent methods for multivariate problems

In this section, we assume we want to minimize a multiviriate function $f(x)$. When there are more than 1 parameters, the situation is more complicated. However, we can reduce multivariate problems into univariate problem if we can find a directon D such that

$$g(\lambda; D, x_n) = f(x_n + \lambda\, D) \tag{7.32}$$

is a descreasing function of λ. Then we can move x_n by a small positive step λ to obtain a point where the function f is smaller. This requirement is equivalent to finding direction D such that

$$g'(0; D, x_n) < 0 \tag{7.33}$$

That is,

$$\sum_i \frac{d}{dx_i} f(x_n) \cdot D_i < 0 \tag{7.34}$$

Written in the form of inner product and gradient of f at x_n:

$$D^t \cdot \nabla f(x_n) < 0 \tag{7.35}$$

After specifying D, we can find λ by minimizing $g(\lambda; D, x_n)$ with respect to λ. We may do this using the methods introduced in previous section.

From the Cauchy-Schwarz inequality,

$$|D^t \cdot \nabla f(x_n)|/|D|_2 \leq |\nabla f(x_n)|_2 \tag{7.36}$$

and taking $D = \nabla f(x_n)$ gives equality, so $\nabla f(x)$ (normalized) is the direction of most rapid increase in f at x_n, and $-\nabla f(x_n)$ is the direction of most rapid decrease. This suggests considering $D = -\nabla f(x_n)$ in the above algorithm. This is called the method of steepest descent. It turns out to perform poorly for most functions, though.

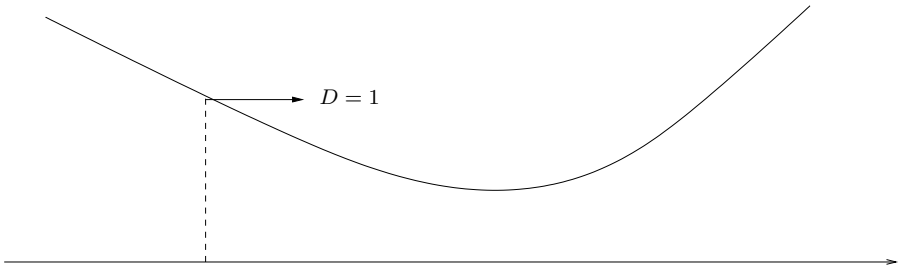


Figure 7.7: Illustration of descent method in 1-dimensional problem

7.3.1 Newton-Raphson methods for multivariate problems

If we let $D = -(\nabla^2 f(x_n))^{-1}\nabla f(x_n)$ and each time let $\lambda = 1$, it is the simple N-R mehod:

$$x_{n+1} = x_n - (\nabla^2 f(x_n))^{-1}\nabla f(x_n) \tag{7.37}$$

The matrix $\nabla^2 f(x_n))$ is usually called Hessian matrix. Applied to maximize log-likelihood function, $-\nabla^2 f(x_n))$ is called the observed fisher information matrix at x_n based on all data points, and $\nabla f(x_n)$ is the score function.

Note that this direction is not gauranteed to satisfy $D^t\nabla f(x_n) < 0$, unless the hessian matrix is positive definite, which can be seen from the following equation:

$$D^t\nabla f(x_n) = -\nabla f(x_n)^t(\nabla^2 f(x_n))^{-1}\nabla f(x_n) \tag{7.38}$$

Example: (MLE for logistic regression models). We model a binary variable y and another predictor variable z as follows:

$$P(y = 1|z) = p(z) = \frac{\exp(\mu(z))}{1 + \exp(\mu(z)} = \frac{1}{1 + \exp(-\mu(z))} \tag{7.39}$$

where $\mu(z) = \beta_0 + \beta_1 z$ is the linear function of x, linking the relationship between y and z. If we observe n data points $(y_1, z_1), \ldots, (y_n, z_n)$, the negative log-likelihood function of β_0 and β_1 is:

$$f(\beta_0, \beta_1) = -\sum_i [y_i\mu(z_i) - \log(1 + \exp(\mu(z_i)))], \tag{7.40}$$

We first compute the gradient of f from the following derivatives:

$$\begin{aligned}
\frac{d}{d\beta_0} f(\beta_0, \beta_1) &= -\sum_i \left(y_i - \frac{\exp(\mu(z_i))}{1 + \exp(\mu(z_i))} \right) && (7.41)\\
&= -\sum_i (y_i - p_i), && (7.42)\\
\frac{d}{d\beta_1} f(\beta_0, \beta_1) &= -\sum_i \left(y_i z_i - z_i \frac{\exp(\mu(z_i))}{1 + \exp(\mu(z_i))} \right) && (7.43)\\
&= -\sum_i (y_i - p_i) z_i && (7.44)
\end{aligned}$$

where $p_i = p(z_i)$.

Then we can find the hessian matrix by calculating the following derivatives:

$$\frac{d^2}{d\beta_0^2} f(\beta_0, \beta_1) = \sum_i p_i(1 - p_i) \tag{7.45}$$

$$\frac{d^2}{d\beta_0 \, d\beta_1} f(\beta_0, \beta_1) = \sum_i z_i p_i(1 - p_i) \tag{7.46}$$

$$\frac{d^2}{d\beta_1^2} f(\beta_0, \beta_1) = \sum_i z_i^2 \, p_i(1 - p_i) \tag{7.47}$$

The hessian matrix is:

$$\nabla^2 f(\beta_0, \beta_1) = \sum_i \begin{pmatrix} 1 & z_i \\ z_i & z_i^2 \end{pmatrix} p_i(1 - p_i) \tag{7.48}$$

From the above expression, we see $\nabla^2 f(\beta_0, \beta_1)$ is positive definite unless $z_1 = \ldots = z_n$. Thus, this is a well-behaved problem.

Next, we program it and do some experiments with R:

```
> # function to compute logistic regression -log likelihood, -score and
> # information. b=parameters, r=binary response, z=covariate
> lik_score_info <- function(b,r,z) {
+         u <- b[1]+b[2]*z
+         u2 <- exp(u)
+         l <- -sum(u*r-log(1+u2))
+         p <- u2/(1+u2)
+         s <- -c(sum(r-p),sum(z*(r-p)))
+         v <- matrix(c(sum(p*(1-p)),sum(z*p*(1-p)),0,sum(z*z*p*(1-p))),2,2)
+         v[1,2] <- v[2,1]
+         list(neg.loglik=l,neg.score=s,inf=v)
+ }
>
> #finding MLE with newton-raphson method
> mle_logistic_nr <- function(b0, no_iter, r, z, debug=FALSE)
+ {
+     result_nr <- matrix(0,no_iter+1, 3)
+     colnames(result_nr) <- c('beta0','beta1','neg_loglike')
+     result_nr[1,] <- c(b0, 0)
+
+     for( i in 1:no_iter + 1)
+     {
+          q <- lik_score_info(result_nr[i-1,1:2],r,z)
+          result_nr[i,1:2] <-   result_nr[i-1,1:2] - solve(q$inf,q$neg.score)
+          result_nr[i-1,3] <- q$neg.loglik
+          if(debug) print(result_nr[i-1,])
+     }
```

```
+     result_nr[-(no_iter+1),]
+ }
>
> gen_logistic_data <- function(b,n)
+ {
+     z <- (runif(n)>.1)*1
+     emu <- exp(b[1]+z*b[2])
+     p <- emu/(1+emu)
+     r <- (runif(n)<p)*1
+
+     list(z=z,r=r)
+ }
>
> data <- gen_logistic_data(c(0,2),100)
> #starting from (0,0)
> mle_logistic_nr(c(0,0),15,data$r,data$z)
          beta0    beta1 neg_loglike
 [1,] 0.0000000 0.000000    69.31472
 [2,] 0.3333333 1.348485    34.93728
 [3,] 0.3364714 1.929911    32.69673
 [4,] 0.3364722 2.099086    32.58475
 [5,] 0.3364722 2.111996    32.58421
 [6,] 0.3364722 2.112067    32.58421
 [7,] 0.3364722 2.112067    32.58421
 [8,] 0.3364722 2.112067    32.58421
 [9,] 0.3364722 2.112067    32.58421
[10,] 0.3364722 2.112067    32.58421
[11,] 0.3364722 2.112067    32.58421
[12,] 0.3364722 2.112067    32.58421
[13,] 0.3364722 2.112067    32.58421
[14,] 0.3364722 2.112067    32.58421
[15,] 0.3364722 2.112067    32.58421
> #starting from (0,-2)
> mle_logistic_nr(c(0,-2),15,data$r,data$z,debug=TRUE)
      beta0       beta1 neg_loglike
     0.0000    -2.0000    181.4874
      beta0       beta1 neg_loglike
  0.3333333   5.2981007  47.8851504
       beta0        beta1  neg_loglike
   0.3364714  -16.0589452 1281.6707124
```

7.3.2 Improving Newton-Raphson method

Some improvements over simple N-R methods:

- Backtracking

$$x_{n+1}(\lambda) = x_n - \lambda[\nabla^2 f(x_n)]^{-1}\nabla f(x_n), \tag{7.49}$$

  Starting from $\lambda = 1$, if the proposed point does not yield a better function value, go back to use a smaller λ.

- Positive definite modification on the hessian matrix

$$\nabla^2 f(x_n) + \alpha I \tag{7.50}$$

- At the begining, use more stable methods, for example use smaller λ and larger α in the above modification, at the latter stage, use N-R iterations.

- Replacing $\nabla^2 f(x_n)$ with another positive definite matrix to avoid computing it. The BFGS algorithm uses:

$$A_{n+1} = A_n + \frac{1}{y^t s} y^t y - \frac{1}{s^t A_n s} A_n s^t s A_n \tag{7.51}$$

  where $y = \nabla f(x_{n+1} - f(x_n)$ and $s = x_{n+1} - x_n$.

- Fisher scoring method: replacing $\nabla^2 f(x_n)$ with $E(\nabla^2 f(x_n))$

7.4 EM algorithm

Regarding the classical discussion of EM algorithm, please read the extra handwritten pages.

Neal and Hinton (1993) provides a new view of EM algorithm, which unifies both E step and M step as maximizing a new function. Here we discuss this view for more general situations than that used in Neal and Hinton (1993).

Let x be the observed data, y be the unobserved data, and θ be the parameter needed to model the joint distribution of x and y. We define a funtional (function of function) involving θ and a unknown distribution about y, represented by a probability density function or a probability mass function $\tilde{P}(y)$, as follows:

$$F(\tilde{P}(\cdot), \theta) = \int \tilde{P}(y) \log(P(x, y|\theta))dy - \int \tilde{P}(y) \log(\tilde{P}(y))dy \tag{7.52}$$

When y is discrete, the integral in the above equation is replaced by summation. We will show that the E-step is equivalent to maximizing F with θ fixed and the M-step is equivalent to maximizing F with $\tilde{P}(\cdot)$ fixed.

First of all, we can rewrite F in (7.52) in another way:

$$\begin{aligned} F(\tilde{P}(\cdot),\theta) &= \int \tilde{P}(y)\log(P(x,y|\theta))dy - \int \tilde{P}(y)\log(P(y|x,\theta))dy + \\ &\quad \int \tilde{P}(y)\log(P(y|x,\theta))dy - \int \tilde{P}(y)\log(\tilde{P}(y))dy \qquad (7.53) \\ &= \int \tilde{P}(y)\log\left(\frac{P(x,y|\theta)}{P(y|x,\theta)}\right)dy - \int \tilde{P}(y)\log\left(\frac{\tilde{P}(y)}{P(y|x,\theta)}\right)dy \qquad (7.54) \\ &= \log\left(P(x|\theta)\right) - \int \tilde{P}(y)\log\left(\frac{\tilde{P}(y)}{P(y|x,\theta)}\right)dy \qquad (7.55) \\ &= L(\theta;x) - \mathrm{KL}(\tilde{P}(\cdot)||P(\cdot|x,\theta)) \qquad (7.56) \end{aligned}$$

where $L(\theta;x)$ is the log likelihood function of θ based on the observed data only, and $\mathrm{KL}(p(\cdot)||q(\cdot))$ is called the Kullback-Leibsler divergence between two distributions $p(\cdot)$ and $q(\cdot)$.

We now maximize $F(\tilde{P}(\cdot),\theta)$ using alternative procedure. Fixing θ at some value, say, $\theta^{(0)}$, the $\tilde{P}(\cdot)$ that maximizes $F(\tilde{P}(\cdot),\theta^{(0)})$ is the $\tilde{P}(\cdot)$ that minimizes $\mathrm{KL}(\tilde{P}(\cdot)||P(\cdot|x,\theta^{(0)}))$ since the first term in (7.56) has nothing with $\tilde{P}(\cdot)$. This optimization will lead to choosing $P(\cdot|x,\theta^{(0)})$ as the $\tilde{P}(\cdot)$, since it is well-known that $\mathrm{KL}(p(\cdot)||q(\cdot))$ is minimized to 0 when $p=q$, shown by the following inequality:

$$\begin{aligned} \mathrm{KL}(p(\cdot)||q(\cdot)) &= \int p(y)\log(p(y)/q(y))dy \qquad (7.57) \\ &= -\int p(y)\log(q(y)/p(y))dy \qquad (7.58) \\ &\geq -\int p(y)(q(y)/p(y)-1)dy, \quad \text{since } \log(x)\leq x-1 \qquad (7.59) \\ &= 0 \qquad (7.60) \end{aligned}$$

This step is equivalent to E-step. Next, fixing $\tilde{P}(\cdot)$ at $P(\cdot|x,\theta^{(0)})$, from the equation (7.52), the θ that maximizes $F(\tilde{P}(\cdot),\theta)$, denoted by $\theta^{(1)}$, is the θ maximizing the first term of (7.52):

$$\theta^{(1)} = \underset{\theta}{\mathrm{argmax}} \int P(y|x,\theta^{(0)})\ \log(P(x,y|\theta))dy \qquad (7.61)$$

This step is equivalent to M-step.

From equation (7.56), it is easy to see that if the θ^* and $\tilde{P}^*(\cdot)$ maximizes $F(\tilde{P}(\cdot),\theta)$, then θ^* maximizes $L(\theta;x)$, as desired.

Below we present two examples programmed with R.

Examples: (mixture of normal distributions)

```
>
> log_like_obs <- function(theta,Y)
+ {
+     sum(log(theta[1]*dnorm(Y,theta[2],1) +
```

```
+            (1 - theta[1]) * dnorm(Y,theta[3],1)))
+ }
>
> em_mixnorm <- function(theta0,Y,no_iters)
+ {
+    result <- matrix(0,no_iters + 1,4)
+    colnames(result) <- c("p","u1","u0","log_lik")
+    result[1,1:3] <- theta0
+    result[1,4] <- log_like_obs(theta0,Y)
+    for(i in 1:no_iters + 1) {
+       like1_weighted <- dnorm(Y,result[i-1,2],1) * result[i-1,1]
+       like0_weighted <- dnorm(Y,result[i-1,3],1) * (1-result[i-1,1])
+       weighs <-   like1_weighted / (like1_weighted + like0_weighted)
+       #update p
+       result[i,1] <- mean(weighs)
+       #update u1
+       result[i,2] <- sum(Y*weighs)/sum(weighs)
+       result[i,3] <- sum(Y*(1-weighs))/sum(1-weighs)
+       result[i,4] <- log_like_obs(result[i,1:3],Y)
+    }
+    result
+ }
>
> gen_mixnorm <- function(theta,n)
+ {
+     Z <- 1*(runif(n) < theta[1])
+     Y <- rep(0,n)
+     for(i in 1:n){
+        if(Z[i]==1) Y[i] <- rnorm(1,theta[2],1)
+        else Y[i] <- rnorm(1,theta[3],1)
+     }
+     Y
+ }
>
> data <- gen_mixnorm(c(0.3,3,0),200)
>
> em_mixnorm(c(0.5,-1,1),data,25)
              p         u1       u0   log_lik
 [1,] 0.5000000 -1.0000000 1.000000 -441.4642
 [2,] 0.3858329 -0.7025300 1.616792 -405.6139
 [3,] 0.4435772 -0.6246891 1.795431 -398.9955
 [4,] 0.4984340 -0.5288000 1.964832 -392.9841
 [5,] 0.5489261 -0.4404694 2.136471 -387.7061
 [6,] 0.5933116 -0.3633110 2.305150 -383.4104
 [7,] 0.6304278 -0.2979683 2.461680 -380.2310
 [8,] 0.6599359 -0.2445888 2.597551 -378.1130
```

```
 [9,] 0.6822955 -0.2027597 2.707746 -376.8435
[10,] 0.6985296 -0.1713546 2.791708 -376.1529
[11,] 0.7099053 -0.1486858 2.852427 -375.8066
[12,] 0.7176598 -0.1328558 2.894616 -375.6435
[13,] 0.7228400 -0.1220853 2.923110 -375.5701
[14,] 0.7262516 -0.1148975 2.941991 -375.5381
[15,] 0.7284766 -0.1101662 2.954348 -375.5245
[16,] 0.7299184 -0.1070813 2.962369 -375.5188
[17,] 0.7308486 -0.1050826 2.967551 -375.5164
[18,] 0.7314471 -0.1037931 2.970887 -375.5154
[19,] 0.7318315 -0.1029635 2.973030 -375.5150
[20,] 0.7320781 -0.1024307 2.974406 -375.5148
[21,] 0.7322362 -0.1020889 2.975287 -375.5148
[22,] 0.7323375 -0.1018698 2.975852 -375.5147
[23,] 0.7324024 -0.1017294 2.976214 -375.5147
[24,] 0.7324439 -0.1016395 2.976446 -375.5147
[25,] 0.7324705 -0.1015819 2.976594 -375.5147
[26,] 0.7324875 -0.1015451 2.976689 -375.5147
>
```

Examples: (censored poisson data)

```
> # AN EM ALGORITHM FOR FINDING THE MLE FOR A CENSORED POISSON MODEL.
> #
> # The data consists of n observed counts, whose mean is m, plus c counts
> # that are observed to be less than 2 (ie, 0 or 1), but whose exact value
> # is not known.  The counts are assumed to be Poisson distributed with
> # unknown mean, lambda.
> #
> # The function below finds the maximum likelihood estimate for lambda given
> # the data, using the EM algorithm started from the specified guess at lambda
> # (default being the mean count with censored counts set to 1), run for the
> # specified number of iterations (default 20).  The log likelihood is printed
> # at each iteration.  It should never decrease.
>
> EM.censored.poisson <- function (n, y_bar, c, lambda0=(n*y_bar+c)/(n+c),
+ iterations=20)
+ {
+   # Set initial guess, and print it and its log likelihood.
+
+   lambda <- lambda0
+
+   cat (0, lambda, log.likelihood(n,y_bar,c,lambda), "\n")
+
+   # Do EM iterations.
+
```

```
+   for (i in 1:iterations)
+   {
+     # The E step: Figure out the distribution of the unobserved data.  For
+     # this model, we need the probability that an unobserved count that is
+     # either 0 or 1 is actually equal to 1, which is p1 below.
+
+     y_mis <- lambda / (1+lambda)
+
+     # The M step: Find the lambda that maximizes the expected log likelihood
+     # with unobserved data filled in according to the distribution found in
+     # the E step.
+
+     lambda <- (n*y_bar + c*y_mis) / (n+c)
+
+     # Print the new guess for lambda and its log likelihood.
+
+     cat (i, lambda, log.likelihood(n,y_bar,c,lambda), "\n")
+   }
+
+   # Return the value for lambda from the final EM iteration.
+
+   lambda
+ }
>
> log.likelihood <- function (n, y_bar, c, lambda)
+ {
+   n*y_bar*log(lambda) - (n+c)*lambda + c*log(1+lambda)
+ }
>
> y <- rpois(200,3)
>
> c<- sum(y < 2)
> y_bar <- mean(y[y >=2])
>
> EM.censored.poisson(length(y),y_bar,c)
0 3.189612 127.8333
1 3.155807 127.8756
2 3.155532 127.8756
3 3.15553 127.8756
4 3.155530 127.8756
5 3.155530 127.8756
6 3.155530 127.8756
7 3.155530 127.8756
8 3.155530 127.8756
9 3.155530 127.8756
10 3.155530 127.8756
```

```
11 3.155530 127.8756
12 3.155530 127.8756
13 3.155530 127.8756
14 3.155530 127.8756
15 3.155530 127.8756
16 3.155530 127.8756
17 3.155530 127.8756
18 3.155530 127.8756
19 3.155530 127.8756
20 3.155530 127.8756
[1] 3.155530
>
```

Chapter 8

Numerical methods for Bayesian inference

8.1 Review of Bayesian inference

See handwritten pages.

See handwritten pages for therectical discussion for numerical quadrature, laplace approximation, and importance sampling.

8.2 Numerical quadrature

We demonstrated the midpoint rule with a simple example.

Example (Normal data with log-norm prior for σ). The data $X_1, \ldots, X_n$ are modelled as IID samples from a normal distribution given the value of the mean parameter μ and the logarithm of the standard deviation σ, denoted by w. The likelihood function of μ and w based on data $X_1, \ldots, X_n$ is therefore:

$$L(\mu, w) = \prod_{i=1}^{n} \phi(x_i; \mu, e^w) \tag{8.1}$$

where $\phi(x; \mu, \sigma)$ is the density function of Normal distribution with mean μ and σ.

We assign normal priors to parameter μ and w with fixed $\mu_0, \sigma_\mu, w_0, \sigma_w$:

$$\mu \sim N(\mu_0, \sigma_\mu) \tag{8.2}$$
$$w \sim N(w_0, \sigma_w) \tag{8.3}$$

The density function of the prior for μ and w is therefore:

$$P(\mu, w) = \phi(\mu; \mu_0, \sigma_u)\, \phi(w; w_0, \sigma_w) \tag{8.4}$$

The posterior distribution of μ and w can be found using Bayes rule:

$$P(\mu, w \mid X_1, \ldots, X_n) = \frac{L(\mu, w)\, P(\mu, w)}{\int_\mu \int_w L(\mu, w)\, P(\mu, w) d\mu dw} \tag{8.5}$$

We will want to compute the normalizing constant $\int_\mu \int_w L(\mu, w)\, P(\mu, w) d\mu dw$, which is needed to compare this model with other models. Since this integral is intractable, we will use midpoint rule to approximate it. The range of μ and w are both infinite, we therefore apply the following change of variable to μ and w before applying midpoint rule:

$$\mu(\mu_t) = \log(\mu_t) - \log(1 - \mu_t) \tag{8.6}$$

$$w(w_t) = \log(w_t) - \log(1 - w_t) \tag{8.7}$$

We then compute the following integral for computing the normalizing constant:

$$\int_{\mu_t} \int_{w_t} L(\mu(\mu_t), w(w_t))\, P(\mu(\mu_t), w(w_t))\, \frac{1}{\mu_t(1-\mu_t)} \frac{1}{w_t(1-w_t)} d\mu_t dw_t \tag{8.8}$$

Below are R functions used to approximate equation (8.8).

```
## the function for computing log likelihood of normal data
log_lik <- function(x,mu,w)
{   sum(dnorm(x,mu,exp(w),log=TRUE))
}

## the function for computing log prior
log_prior <- function(mu,w, mu_0,sigma_mu,w_0,sigma_w)
{   dnorm(mu,mu_0,sigma_mu,log=TRUE) + dnorm(w,w_0,sigma_w,log=TRUE)
}

## the function for computing the unormalized log posterior
## given transformed mu and w
log_post_tran <- function(x, mu_t, w_t, mu_0,sigma_mu,w_0,sigma_w)
{
    #log likelihood
    log_lik(x,logi(mu_t), logi(w_t)) +
    #log prior
    log_prior(logi(mu_t), logi(w_t), mu_0,sigma_mu,w_0,sigma_w) +
    #log derivative of transformation
    log_der_logi(mu_t) + log_der_logi(w_t)
}
```

```
## the logistic function for transforming (0,1) value to (-inf,+inf)
logi <- function(x)
{  log(x) - log(1-x)
}

## the log derivative of logistic function
log_der_logi <- function(x)
{  -log(x) - log(1-x)
}

## the generic function for approximating 1-D integral with midpoint rule
## the logarithms of the function values are passed in
## the log of the integral result is returned

## log_f  --- a function computing the logarithm of the integrant function
## range  --- the range of integral varaible, a vector of two elements
## n      --- the number of points at which the integrant is evaluated
## ...    --- other parameters needed by log_f
log_int_mid <- function(log_f, range, n,...)
{   if(range[1] >= range[2])
        stop("Wrong ranges")
    h <- (range[2]-range[1]) / n
    v_log_f <- sapply(range[1] + (1:n - 0.5) * h, log_f,...)
    log_sum_exp(v_log_f) + log(h)
}

## a function computing the sum of numbers represented with logarithm
## lx     --- a vector of numbers, which are the log of another vector x.
## the log of sum of x is returned
log_sum_exp <- function(lx)
{   mlx <- max(lx)
    mlx + log(sum(exp(lx-mlx)))
}

## a function computing the normalization constant
log_mar_gaussian_mid <- function(x,mu_0,sigma_mu,w_0,sigma_w,n)
{
    ## function computing the normalization constant of with mu_t fixed
    log_int_gaussian_mu <- function(mu_t)
    {   log_int_mid(log_f=log_post_tran,range=c(0,1),n=n,
                    x=x,mu_t=mu_t,mu_0=mu_0,sigma_mu=sigma_mu,
                    w_0=w_0,sigma_w=sigma_w)
    }

    log_int_mid(log_f=log_int_gaussian_mu,range=c(0,1), n=n)
```

```
}

## we use Monte Carlo method to debug the above function
log_mar_gaussian_mc <- function(x,mu_0,sigma_mu,w_0,sigma_w,iters_mc)
{
    ## draw samples from the priors
    mus <- rnorm(iters_mc,mu_0,sigma_mu)
    ws <- rnorm(iters_mc,w_0,sigma_w)
    one_log_lik <- function(i)
    {  log_lik(x,mus[i],ws[i])
    }
    v_log_lik <- sapply(1:iters_mc,one_log_lik)
    log_sum_exp(v_log_lik) - log(iters_mc)
}
```

We do some experiments to test and use the function `log_mar_gaussian`:

```
> source("gaussian-midpoint.R")
> ## debugging the program
> x <- rnorm(50)
> log_mar_gaussian_mid(x,0,1,0,1,100)
[1] -75.74747
> log_mar_gaussian_mc(x,0,1,0,1,100000)
[1] -75.74272
> x <- rnorm(10) # another debug
> log_mar_gaussian_mid(x,0,1,0,1,100)
[1] -15.77959
> log_mar_gaussian_mc(x,0,1,0,1,100000)
[1] -15.77239
>
> ## looking at the convergence
> x <- rnorm(100)
> for(i in seq(10,90,by=10))
+ { cat("n = ",i,",")
+   cat(" Estimated Log Marginal Likelihood =",
+       log_mar_gaussian_mid(x,0,1,0,1,i),"\n")
+ }
n =  10 , Estimated Log Marginal Likelihood = -153.9522
n =  20 , Estimated Log Marginal Likelihood = -152.8829
n =  30 , Estimated Log Marginal Likelihood = -152.9467
n =  40 , Estimated Log Marginal Likelihood = -152.9536
n =  50 , Estimated Log Marginal Likelihood = -152.9537
n =  60 , Estimated Log Marginal Likelihood = -152.9537
n =  70 , Estimated Log Marginal Likelihood = -152.9537
n =  80 , Estimated Log Marginal Likelihood = -152.9537
n =  90 , Estimated Log Marginal Likelihood = -152.9537
```

```
>
> ## looking at the log marginal likelihood of different models
> x <- rnorm(100)
> log_mar_gaussian_mid(x,mu_0=0,sigma_mu=1,w_0=0,sigma_w=1,100)
[1] -147.9318
> log_mar_gaussian_mid(x,mu_0=-5,sigma_mu=1,w_0=0,sigma_w=1,100)
[1] -160.2222
> log_mar_gaussian_mid(x,mu_0=0,sigma_mu=100,w_0=0,sigma_w=1,100)
[1] -152.5316
> log_mar_gaussian_mid(x,mu_0=0,sigma_mu=0.1,w_0=0,sigma_w=1,100)
[1] -145.9879
> log_mar_gaussian_mid(x,mu_0=0,sigma_mu=0.01,w_0=0,sigma_w=1,100)
[1] -146.5134
> log_mar_gaussian_mid(x,mu_0=0,sigma_mu=0.001,w_0=0,sigma_w=1,100)
[1] -342.224
> log_mar_gaussian_mid(x,mu_0=0,sigma_mu=0.1,w_0=0,sigma_w=10,100)
[1] -148.2878
>
```

8.3 Laplace approximation

Example: We will apply laplace approximation to normal models with log-normal prior for the standard deviation. Below are the R functions for computing the log of the marginal likelihood with laplace approximation.

```
## the generic function for finding laplace approximation of integral of 'f'
## neg_log_f    --- the negative log of the intergrand function
## p0           --- initial value in searching mode
## ...          --- other arguments needed by neg_log_f
## the log of the integral is returned
log_int_lap <- function(neg_log_f,p0,...)
{   ## looking for the mode and hessian of the log likehood function
    result_min <- nlm(f=neg_log_f,p=p0, hessian=TRUE,...)
    hessian <- result_min$hessian
    neg_log_like_mode <- result_min$minimum
    ## computing the integral based on Gaussian approximation
    - neg_log_like_mode + 0.5 * ( sum(log(2*pi) - log(svd(hessian)$d) ) )
}

## the function for computing log likelihood of normal data
log_lik <- function(x,mu,w)
{   sum(dnorm(x,mu,exp(w),log=TRUE))
}
```

```
## the function for computing log prior
log_prior <- function(mu,w, mu_0,sigma_mu,w_0,sigma_w)
{   dnorm(mu,mu_0,sigma_mu,log=TRUE) + dnorm(w,w_0,sigma_w,log=TRUE)
}

## the function for computing the negative log of likelihood * prior
neg_log_post <- function(x, theta, mu_0,sigma_mu,w_0,sigma_w)
{   - log_lik(x,theta[1], theta[2]) -
    log_prior(theta[1],theta[2],mu_0,sigma_mu,w_0,sigma_w)
}


## the function for computing the negative log of exp(w) * likelihood * prior
neg_log_wpost <- function(x, theta, mu_0,sigma_mu,w_0,sigma_w)
{   - theta[2] - log_lik(x,theta[1], theta[2]) -
    log_prior(theta[1],theta[2],mu_0,sigma_mu,w_0,sigma_w)
}

## approximating the log of integral of likelihood * prior
log_mar_gaussian_lap <- function(x,mu_0,sigma_mu,w_0,sigma_w)
{   log_int_lap(neg_log_post,p0=c(mean(x),log(sqrt(var(x)))),
                x=x,mu_0=mu_0,sigma_mu=sigma_mu,w_0=w_0,sigma_w=sigma_w)
}

## approximating the log of the integral of exp(w) * likelihood * prior
log_mean_sigma_unnorm <- function(x,mu_0,sigma_mu,w_0,sigma_w)
{   log_int_lap(neg_log_wpost,p0=c(mean(x),log(sqrt(var(x)))),
                x=x,mu_0=mu_0,sigma_mu=sigma_mu,w_0=w_0,sigma_w=sigma_w)
}

## estimating the mean of the posterior distribution of sigma
postmean_sigma_lap <- function(x,mu_0,sigma_mu,w_0,sigma_w)
{  exp(log_mean_sigma_unnorm(x,mu_0,sigma_mu,w_0,sigma_w) -
       log_mar_gaussian_lap(x,mu_0,sigma_mu,w_0,sigma_w)
      )
}
```

We tested the function `log_gaussian_mar_lap` on two data sets and compared with numerical quadrature and Monte Carlo method:

```
> source("gaussian-lap.R")
> source("gaussian-midpoint.R")
>
> ## debugging the program
> x <- rnorm(50)
> log_mar_gaussian_mid(x,0,10,0,5,100)
```

```
[1] -85.32002
> log_mar_gaussian_mc(x,0,10,0,5,100000)
[1] -85.45555
> log_mar_gaussian_lap(x,0,10,0,5)
[1] -85.33843
>
>
> x <- rnorm(500)
> log_mar_gaussian_mid(x,0,10,0,5,100)
[1] -715.267
> log_mar_gaussian_mc(x,0,10,0,5,100000)
[1] -715.2394
> log_mar_gaussian_lap(x,0,10,0,5)
[1] -715.2687
>
> ## finding posterior mean of sigma by laplace approximation
> x <- rnorm(500,0,3)
> postmean_sigma_lap(x,0,10,0,5)
[1] 2.81234
>
> x <- rnorm(50,0,0.1)
> postmean_sigma_lap(x,0,10,0,5)
[1] 0.09680444
>
```

8.4 Importance sampling

Example: (estimating small probability) Suppose X has distribution $N(0,1)$. We want to estiamte $P(X \in A)$ for an interval A. A naive way is to draw samples from $N(0,1$ and then count the fraction of those samplings in A. Another way is to calculate the following integral:

$$P(X \in A) = \int \phi(x)\, I(x \in A)\, dx \tag{8.9}$$

$$= \int \phi(x)\, \frac{I(x \in A)}{\text{length}(A)}\, dx\, \text{length}(A) \tag{8.10}$$

where $\phi(x)$ is the density function of $N(0,1)$.

The integral above suggests an approach of using importance sampling for estimating $P(x \in A)$. I.e., we can sampling from Uniform distribution over A and then find the average of $\phi(x)$.

Belows are the R functions and some experiment results:

```
> ## estimating the probability P(X in A) for X ~ N(0,1)
> ## by sampling from N(0,1)
```

```
> est_normprob_mc <- function(A,iters_mc)
+ {
+      X <- rnorm(iters_mc)
+      mean((X > A[1]) * (X<A[2]))
+ }
>
> ## estimating the probability P(X in A) for X ~ N(0,1)
> ## by sampling from Unif(A[1],A[2])
> est_normprob_imps <- function(A, iters_mc)
+ {
+      X <- runif(iters_mc,A[1],A[2])
+      mean(dnorm(X))*(A[2]-A[1])
+ }
>
> A <- c(-2,2)
> probs_mc <- replicate(1000,est_normprob_mc(A,100))
> probs_imps <- replicate(1000,est_normprob_imps(A,100))
> var(probs_mc)
[1] 0.0003960376
> var(probs_imps)
[1] 0.002222247
>
> A <- c(1.5,2)
> probs_mc <- replicate(1000,est_normprob_mc(A,100))
> probs_imps <- replicate(1000,est_normprob_imps(A,100))
> var(probs_mc)
[1] 0.0004126101
> var(probs_imps)
[1] 1.327551e-06
>
> A <- c(1.9,2)
> probs_mc <- replicate(1000,est_normprob_mc(A,100))
> probs_imps <- replicate(1000,est_normprob_imps(A,100))
> var(probs_mc)
[1] 6.002002e-05
> var(probs_imps)
[1] 1.084326e-09
>
> A <- c(1.95,2)
> probs_mc <- replicate(1000,est_normprob_mc(A,100))
> probs_imps <- replicate(1000,est_normprob_imps(A,100))
> var(probs_mc)
[1] 3.236987e-05
> var(probs_imps)
[1] 6.752362e-11
>
```

Example: We apply the importance sampling to the normal models with log-normal prior for the standard deviation for finding the log marginal likelihood. We will use the following R functions, which is saved in a file namded `gaussian-imps.R`:

```
## computing the log probability density function of multivariate normal
## x     --- a vector, the p.d.f at x will be computed
## mu   --- the mean vector of multivariate normal distribution
## A     --- the inverse covariance matrix of multivariate normal distribution
log_pdf_mnormal <- function(x, mu, A)
{   0.5 * ( -length(mu)*log(2*pi) + sum(log(svd(A)$d)) - t(x-mu) %*% A %*% (x-mu) )

}
## the function for computing log likelihood of normal data
log_lik <- function(x,mu,w)
{   sum(dnorm(x,mu,exp(w),log=TRUE))
}

## the function for computing log prior
log_prior <- function(mu,w, mu_0,sigma_mu,w_0,sigma_w)
{   dnorm(mu,mu_0,sigma_mu,log=TRUE) + dnorm(w,w_0,sigma_w,log=TRUE)
}

## the function for computing the negative log of likelihood * prior
neg_log_post <- function(x, theta, mu_0,sigma_mu,w_0,sigma_w)
{   - log_lik(x,theta[1], theta[2]) -
    log_prior(theta[1],theta[2],mu_0,sigma_mu,w_0,sigma_w)
}


## computing the log marginal likelihood using importance sampling with
## the posterior distribution approximated by the Gaussian distribution at
## its mode
log_mar_gaussian_imps <- function(x,mu_0,sigma_mu,w_0,sigma_w,iters_mc)
{   result_min <- nlm(f=neg_log_post,p=c(mean(x),log(sqrt(var(x)))),
                      hessian=TRUE,
                      x=x,mu_0=mu_0,sigma_mu=sigma_mu,w_0=w_0,sigma_w=sigma_w)
    hessian <- result_min$hessian
    mu <- result_min$estimate

    ## finding the multiplier for sampling from multivariate normal
    Sigma <- t( chol(solve(hessian)) )
    ## draw samples from N(mu, Sigma %*% Sigma')
    thetas <- Sigma %*% matrix(rnorm(2*iters_mc),2,iters_mc) + mu

    ## values of log approximate p.d.f. at samples
    log_pdf_mnormal_thetas <- apply(thetas,2,log_pdf_mnormal,mu=mu,A=hessian)
```

```
    ## values of log true p.d.f. at samples
    log_post_thetas <- - apply(thetas,2,neg_log_post,x=x, mu_0=mu_0,
                               sigma_mu=sigma_mu,w_0=w_0,sigma_w=sigma_w)

    ## averaging the weights, returning its log
    log_sum_exp(log_post_thetas-log_pdf_mnormal_thetas) - log(iters_mc)
}

## a function computing the sum of numbers represented with logarithm
## lx     --- a vector of numbers, which are the log of another vector x.
## the log of sum of x is returned
log_sum_exp <- function(lx)
{   mlx <- max(lx)
    mlx + log(sum(exp(lx-mlx)))
}

## we use Monte Carlo method to debug the above function
log_mar_gaussian_mc <- function(x,mu_0,sigma_mu,w_0,sigma_w,iters_mc)
{
    ## draw samples from the priors
    mus <- rnorm(iters_mc,mu_0,sigma_mu)
    ws <- rnorm(iters_mc,w_0,sigma_w)
    one_log_lik <- function(i)
    {  log_lik(x,mus[i],ws[i])
    }
    v_log_lik <- sapply(1:iters_mc,one_log_lik)
    log_sum_exp(v_log_lik) - log(iters_mc)
}
```

We run some experiments to test the function `log_mar_gaussian_imps`:

```
> source("gaussian-imps.R")
> ## debugging the program
> x <- rnorm(50)
> log_mar_gaussian_imps(x,0,1,0,5,100)
[1] -65.18838
> log_mar_gaussian_mc(x,0,1,0,5,10000)
[1] -65.3026
> x <- rnorm(10) # another debug
> log_mar_gaussian_imps(x,0,1,0,5,100)
[1] -15.63691
> log_mar_gaussian_mc(x,0,1,0,5,10000)
[1] -15.52662
>
> ## comparing importance sampling with Gaussian approximation with naive monte carlo
> x <- rnorm(200)
```

```
> v_log_mar_imps <- replicate(1000, log_mar_gaussian_imps(x,0,1,0,5,100))
> v_log_mar_mc <- replicate(1000, log_mar_gaussian_mc(x,0,1,0,5,100))
>
> var(v_log_mar_imps)
[1] 0.0002001579
> var(v_log_mar_mc)
[1] 224.3225
>
> postscript("comp-imps-naivemc.eps", width=10, height=4.5, horizont=FALSE)
> par(mfrow=c(1,2))
> xlim <- c(min(c(v_log_mar_imps,v_log_mar_mc)),max(c(v_log_mar_imps,v_log_mar_mc)))
> hist(v_log_mar_imps,main="Sampling from approximate Gaussian",xlim=xlim)
> hist(v_log_mar_mc,main="Sampling from the prior",xlim=xlim)
> dev.off()
null device
          1
>
```

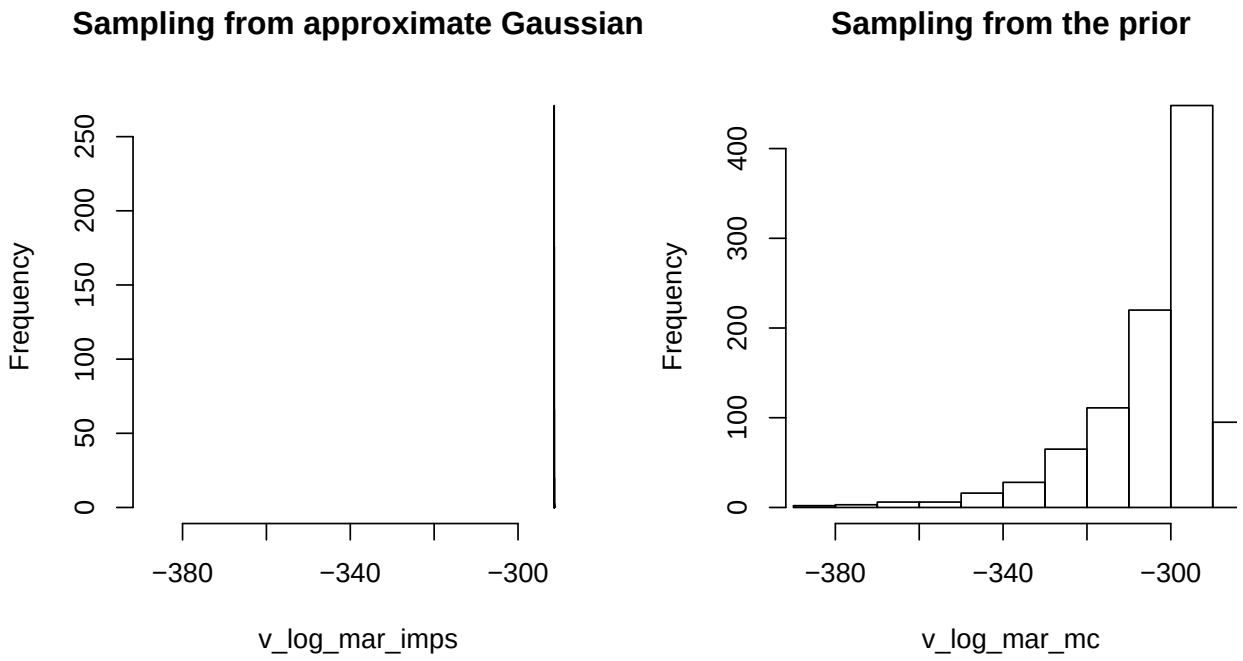


Figure 8.1: Demonstration of importance sampling with an example of estimating the log marginal likelihood function of normal models with log-normal prior for the standard deviation.

8.5 Markov chain Monte Carlo

See handwritten pages for therectical discussion for Markov chain Monte Carlo.

Example: We demonstrate the convergence theorem of Markov chain with a toy example. The target distribution is uniform over $S = (0, 1, \ldots, n-1)$. The Markov chain is defined by the transition probability below:

$$T(j|i) = \begin{cases} 0.45 & \text{for } j = (i+1) \bmod n \\ 0.45 & \text{for } j = (i-1) \bmod n \\ 0.10 & \text{for } j = i \end{cases} \tag{8.11}$$

We demonstrate the convergence theorem by simulating a certain number of Markov chains given the same initial value. We will exam the convergence by looking at the distribution of the state in last iteration. The following is the R functions as well as the R script for doing experiment:

```
## generic function for plotting a probability mass function
plot_pmf <- function(x,p,...)
{
   if(length(x) != length(p))
      stop("The lengths of 'p' and 'x' are different")
   if( sum(p) != 1) p <- p/sum(p)

   xlim <- c(min(x),max(x))
   ylim <- c(0,max(p))
   plot(c(x[1],x[1]),c(0,p[1]),type='l',
        xlim=xlim,ylim=ylim,xlab='x',ylab='p',...)
   lapply( seq(2,length(x),by=1),
           function(i) points(c(x[i],x[i]),c(0,p[i]), type='l') ) -> nothing
}



## simulating a Markov chain with the uniform distribution over discrete
## space (0,2,...,n-1) and with transition probability 0.5 to its adjacent
## points in cyclic way
## no_sim       --- number of chains simulated
## n            --- number of points in state space
## initial_value   --- initial value of Markov chain
## iters_mc     --- length of a Markov chain
demo_converge_mc <- function(no_sim, n, initial_value, iters_mc,...)
{
    forward <- function(i)
    {  if( i == n - 1) 0
       else i + 1
    }
```

```
    backward <- function(i)
    {  if( i == 0 ) n - 1
       else i - 1
    }

    chain <- c(initial_value,rep(0, iters_mc))
    ## simulating one chain starting from initial_value
    one_sim <- function()
    {   for(i in seq(2,iters_mc+1) )
        {   u <- runif(1)
            if(u < 0.45)  chain[i] <<- forward(chain[i-1])
            else if( u > 0.55 ) chain[i] <<- backward(chain[i-1])
                 else chain[i] <<- chain[i-1]
        }
        chain[length(chain)]
    }
    ## simulating no_sim chains of length iters_mc and recording the last state
    states_last_iter <- replicate(no_sim,one_sim())

    ## computing the frequency of states_last_iter
    p <- sapply(seq(0,n-1), function(x) sum(states_last_iter==x) )

    ## plotting the PMF
    plot_pmf(seq(0,n-1),p,
             main=paste("Number of chains:",
             no_sim,",","Initial:",initial_value,",","length:",iters_mc))
    ## plot the trace of the last chain
    plot(chain,xlab="Iteration",ylab="Markov chain",
         main="A Markov chain trace",type='b',pch=20)
}

postscript("demo-converge-mc-1.eps",width=7,height=8.5,paper="special",
           horizontal=F)
par(mfrow=c(2,2),cex=0.7)

demo_converge_mc(5000, 10, 0, 5)

demo_converge_mc(5000, 10, 0, 10)

dev.off()

postscript("demo-converge-mc-2.eps",width=7,height=8.5,paper="special",
           horizontal=F)
par(mfrow=c(2,2),cex=0.7)

demo_converge_mc(5000, 10, 0, 20)
```

```
demo_converge_mc(5000, 10, 0, 50)

dev.off()

postscript("demo-converge-mc-3.eps",width=7,height=8.5,paper="special",
           horizontal=F)
par(mfrow=c(2,2),cex=0.7)

demo_converge_mc(5000, 10, 4, 100)

demo_converge_mc(5000, 10, 7, 100)

dev.off()
```

The plots produced from the above experiments are shown next.

Example: In this example we use Gibbs sampling with Metropolis step to draw samples for some bivariate normal distributions and mixture of bivariate normal distributions. We also sample from the mixture biviariate normal distribution with multivariate Metropolis method. The generic functions for performing Gibbs sampling and Metropolis sampling are given below. They are followed the R commands for performing some experiments. The pictures are attached afterward.

```
## a generic function for Metropolis sampling with Gaussian proposal
## iters            --- length of Markov chain
## log_f            --- the log of the density function
## p                --- the number of variables being sampled
## x0               --- the initial value
## stepsizes        --- the standard deviations of Gaussian proposal
##                  stepsizes[i] for the 'i'th  variable
## iters_per.iter   --- for each transition specified by 'iters',
##                  the Markov chain sampling is run 'iters_per.iter' times
##                  this argument is used to avoid saving the whole Markov chain
## ...              --- other arguments needed to compute log_f

## a matrix of (iters + 1) * p is returned, with each row for an iteration

met_gaussian <- function(log_f,iters, p, x0, stepsizes, iters_per.iter=1, ...)
{   ## creating a matrix to save the Markov chain, with each row for an iteration
    chain <- matrix(0,iters+1,p)
    chain[1,] <- x0
    old_log_f <- log_f(x0,...)

    ## doing one transition
    one_transition <- function(i)
    {   chain[i,] <<- chain[i-1,]
        i_inside <- 0
```

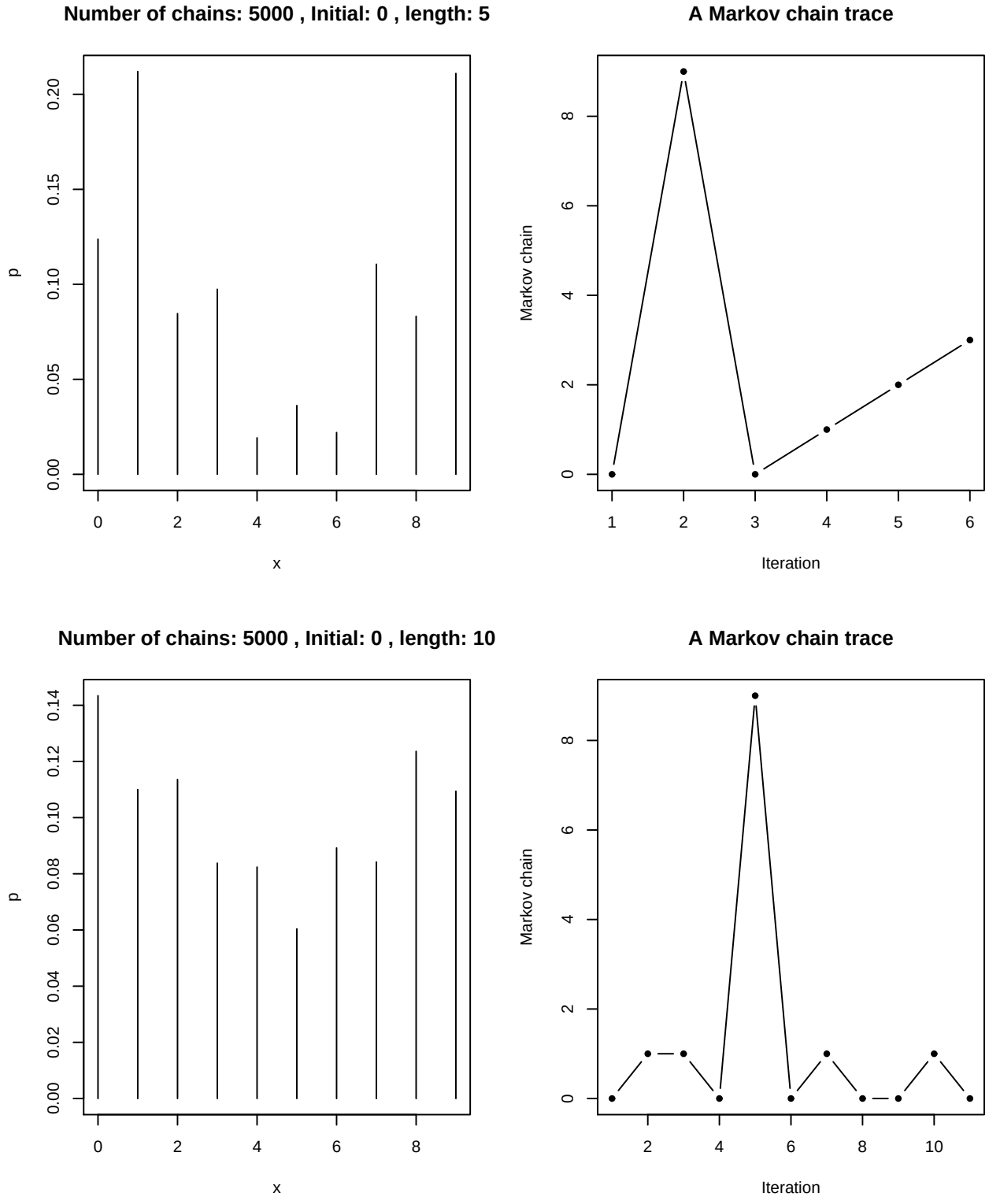


Figure 8.2: Distribution of the state in the last iteration of Markov chains of length 5 and 10, and the traces of two particular Markov chains.

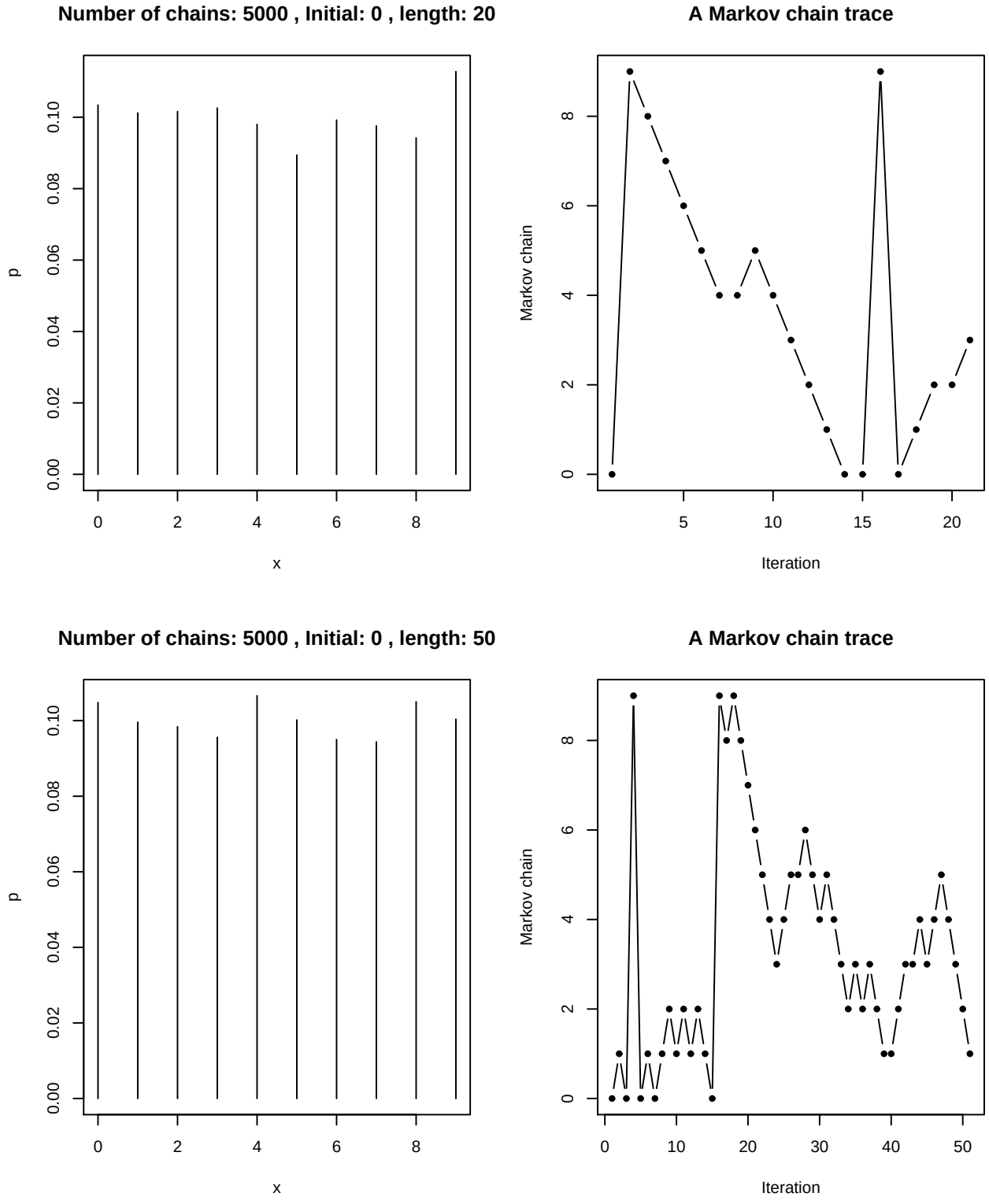


Figure 8.3: Distribution of the state in the last iteration of Markov chains of length 20 and 50, and the traces of two particular Markov chains.

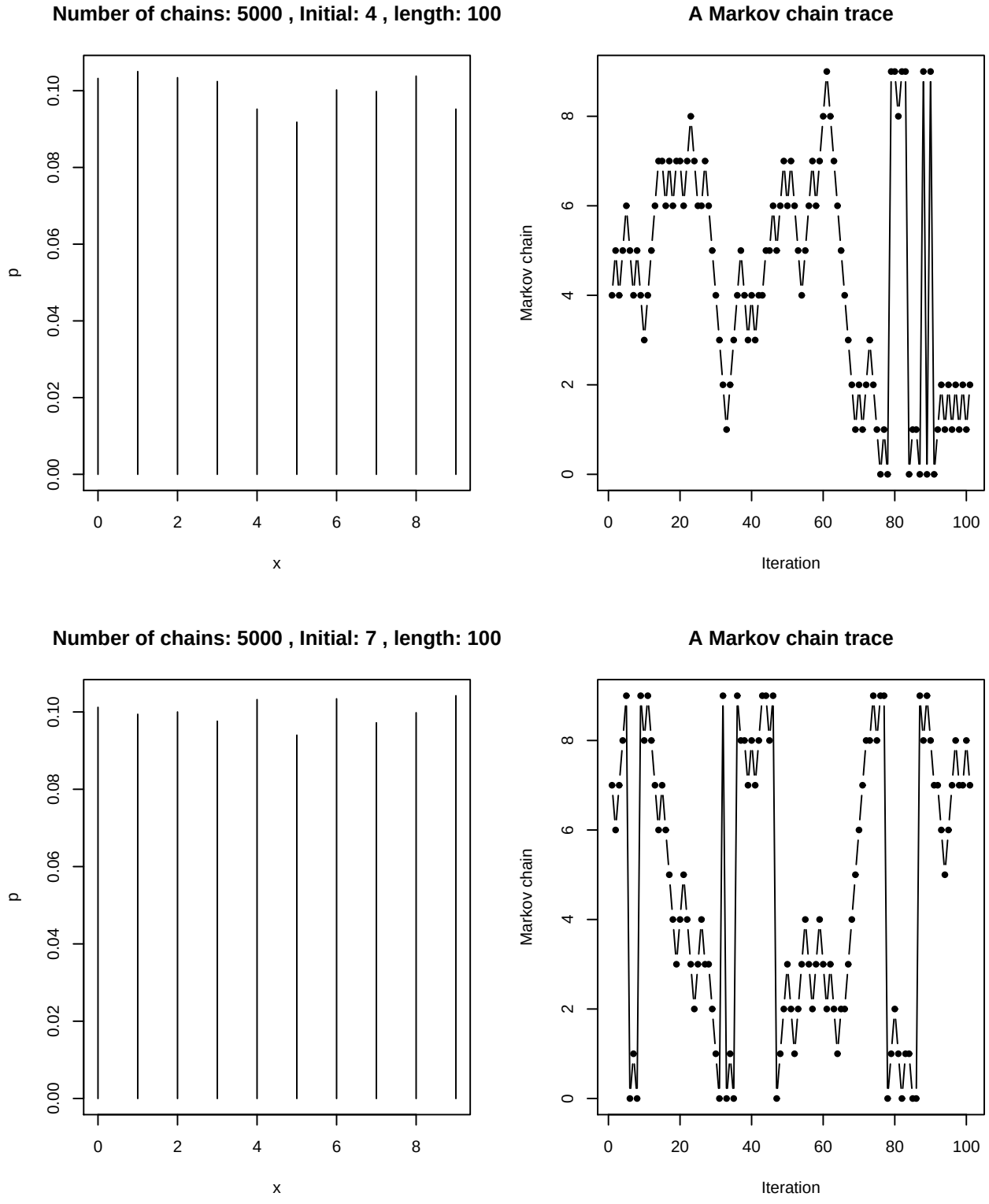


Figure 8.4: Distribution of the state in the last iteration of Markov chains of length 100, and the traces of two particular Markov chains.

```
        repeat{
                i_inside <- i_inside + 1
                ## propose a point
                x_prop <- rnorm(p) * stepsizes + chain[i,]
                ## decide whether to accept it
                new_log_f <- log_f(x_prop,...)
                if(log(runif(1)) < new_log_f - old_log_f)
                {  chain[i,] <<- x_prop
                   old_log_f <<- new_log_f
                }
                if(i_inside == iters_per.iter) break
        }
    }
    ## perform iters Metropolis updating
    sapply(seq(2,iters+1),one_transition)
    ## return the whole chain
    chain
}

## a generic function for Gibbs sampling with Metropolis steps
## iters_mc         --- iterations of Gibbs sampling
## iters_met        --- iterations of Metropolis for each 1-dimensional sampling
## log_f            --- the log of the density function
## p                --- the dim of variables sampled
## x0               --- the initial value
## stepsizes_met    --- a vector of length p, with
##                  stepsizes_met[i] being the standard deviation of Gaussian
##                  proposal for updating 'i'th parameter
## iters_per.iter   --- for each transition specified by 'iters',
##                  the Markov chain sampling is run 'iters_per.iter' times
##                  this argument is used to avoid saving the whole Markov chain
## ...              --- extra arguments needed to compute log_f

## a matrix of (iters_mc + 1) * p is returned, with each row for an iteration

gibbs_met <- function(log_f,p,x0,iters_mc,iters_met,stepsizes_met,
                      iters_per.iter=1,...)
{
   chain <- matrix(0, iters_mc + 1, p)
   chain[1,] <- x0

   ## update only 'i_par' th parameter in 'iter' iteration
   gibbs_update_per.iter_per.par <- function(iter, i_par)
   {   log_f_condition <- function(x_i)
       {   x <- chain[iter,]
           x[i_par] <- x_i
```

```
          log_f(x,...)
      }
      ## updating parameter with index i_par with Metropolis method
      chain[iter,i_par] <<-
             met_gaussian(log_f_condition,iters_met,1,
                    chain[iter,i_par],stepsizes_met[i_par])[iters_met+1]
   }
   ## gibbs sampling for iteration 'iter' for all parameters
   gibbs_update_per.iter <- function(iter) {
        ## copy the states in 'iter-1' iteration to this iteration
        chain[iter,] <<- chain[iter-1,]
        replicate(iters_per.iter,
                  sapply(1:p,gibbs_update_per.iter_per.par,iter=iter) )
   }
   ## perform iters_mc gibbs sampling for all parameters
   sapply( seq(2,iters_mc+1), gibbs_update_per.iter )

   chain
}

> source("gibbs-met.R")
> ###########################################################################
> ##     demonstration by sampling from bivariate normal distributions
> ###########################################################################
>
> ## the function computing the log density function of multivariate normal
> ## x     --- a vector, the p.d.f at x will be computed
> ## mu   --- the mean vector of multivariate normal distribution
> ## A     --- the inverse covariance matrix of multivariate normal distribution
> log_pdf_mnormal <- function(x, mu, A)
+ {  0.5 * ( -length(mu)*log(2*pi)+sum(log(svd(A)$d))-sum(t(A*(x-mu))*(x-mu)) )
+ }
>
> ###########################################################################
> ## sampling from a bivariate normal distribution with correlation 0.1,
> ## both marginal standard deviations 1, mean vector (0,5)
> A = solve(matrix(c(1,0.1,0.1,1),2,2))
> mc_mvn <- gibbs_met(log_f=log_pdf_mnormal,2,x0=c(0,0),iters_mc=1000,iters_met=5,
+                     stepsizes_met=c(0.5,0.5), mu=c(0,5), A = A)
>
> postscript("mc_mvn_lowcor.eps",width=7,height=8,horiz=FALSE)
>
> par(mfrow=c(2,2), oma=c(0,0,1,0))
>
> ## looking at the trace of Markov chain in the first 100 iterations
> plot(mc_mvn[1:100,1],mc_mvn[1:100,2],type="b",pch=20,
```

```
+      main="Markov chain trace of both variables")
>
> ## looking at the trace of Markov chain for a variable
> plot(mc_mvn[,1],type="b",pch=20, main="Markov chain trace of the 1st variable")
>
> ## looking at the QQ plot of the samples for a variable
> qqnorm(mc_mvn[-(1:50),1],main="Normal QQ plot of the 1st variable")
>
> ## looking at the ACF of the samples for a variable
> acf(mc_mvn[,1],main="ACF plot of the 1st variable")
>
> title(main="Gibbs sampling for a bivariate normal with correlation 0.1",
+       outer=TRUE)
>
> dev.off()
null device
          1
>
> ## checking the correlation of samples
> cat("The sample correlation is",cor(mc_mvn[-(1:50),1],mc_mvn[-(1:50),2]),"\n")
The sample correlation is 0.05248102
>
> ###########################################################################
> ## sampling from a bivariate normal distribution with correlation 0.9,
> ## both marginal standard deviations 1, mean vector (0,5)
> A = solve(matrix(c(1,0.9,0.9,1),2,2))
> mc_mvn <- gibbs_met(log_f=log_pdf_mnormal,2,x0=c(0,0),iters_mc=1000,iters_met=5,
+                     stepsizes_met=c(0.5,0.5), mu=c(0,5), A = A)
>
> postscript("mc_mvn_highcor.eps",width=7,height=8,horiz=FALSE)
>
> par(mfrow=c(2,2), oma=c(0,0,1,0))
>
> ## looking at the trace of Markov chain in the first 100 iterations
> plot(mc_mvn[1:100,1],mc_mvn[1:100,2],type="b",pch=20,
+      main="Markov chain trace of both variables")
>
> ## looking at the trace of Markov chain for a variable
> plot(mc_mvn[,1],type="b",pch=20, main="Markov chain trace of the 1st variable")
>
> ## looking at the QQ plot of the samples for a variable
> qqnorm(mc_mvn[-(1:50),1],main="Normal QQ plot of the 1st variable")
>
> ## looking at the ACF of the samples for a variable
> acf(mc_mvn[,1],main="ACF plot of the 1st variable")
>
```

```
> title(main="Gibbs sampling for a bivariate normal with correlation 0.9",
+       outer=TRUE)
>
> dev.off()
null device
          1
>
> ## checking the correlation of samples
> cat("The sample correlation is",cor(mc_mvn[-(1:50),1],mc_mvn[-(1:50),2]),"\n")
The sample correlation is 0.9060792
>
>
> ##############################################################################
> ##     demonstration by sampling from mixture bivariate normal distributions
> ##############################################################################
>
> ## the function computing the log density function of mixture multivariate normal
> ## x     --- a vector, the p.d.f at x will be computed
> ## mu1,mu2   --- the mean vector of multivariate normal distribution
> ## A1,A2     --- the inverse covariance matrix of multivariate normal distribution
> ## mixture proportion is 0.5
> log_pdf_twonormal <- function(x, mu1, A1, mu2, A2)
+ {  log_sum_exp( c(log_pdf_mnormal(x,mu1,A1), log_pdf_mnormal(x,mu2,A2)) )
+ }
>
> log_sum_exp <- function(lx)
+ {  ml <- max(lx)
+    ml + log(sum(exp(lx-ml)))
+ }
>
>
> ##############################################################################
> ## set parameters defining a mixture bivariate distribution
> A1 <- solve(matrix(c(1,0.1,0.1,1),2,2))
> A2 <- solve(matrix(c(1,0.1,0.1,1),2,2))
> mu1 <- c(0,0)
> mu2 <-c(4,4)
>
> ## performing Gibbs sampling
> mc_mvn <- gibbs_met(log_f=log_pdf_twonormal,2,x0=c(0,0),iters_mc=4000,iters_met=5,
+                     stepsizes_met=c(0.5,0.5),mu1=mu1,mu2=mu2,A1=A1,A2=A2)
>
> postscript("mc_mvn_closemix.eps",width=7,height=8,horiz=FALSE)
>
> par(mfrow=c(2,2), oma=c(0,0,2,0))
>
```

```
> ## looking at the trace of Markov chain in the first 100 iterations
> plot(mc_mvn[,1],mc_mvn[,2],type="b",pch=20,
+      main="Markov chain trace of both variables")
>
> ## looking at the trace of Markov chain for a variable
> plot(mc_mvn[,1],type="b",pch=20, main="Markov chain trace of the 1st variable")
>
> ## looking at the trace of Markov chain for a variable
> plot(mc_mvn[,2],type="b",pch=20, main="Markov chain trace of the 2nd variable")
>
> ## looking at the ACF of the samples for a variable
> acf(mc_mvn[,1],main="ACF plot of the 1st variable")
>
> title(main="Gibbs sampling for a mixture of two bivariate normal distributions
+ with locations (0,0) and (4,4)", outer=TRUE)
>
> dev.off()
null device
          1
>
> ## checking the correlation of samples
> cat("The sample correlation is",cor(mc_mvn[-(1:50),1],mc_mvn[-(1:50),2]),"\n")
The sample correlation is 0.8172255
>
>
> ###############################################################################
> ## set parameters defining a mixture bivariate distribution
> A1 <- solve(matrix(c(1,0.1,0.1,1),2,2))
> A2 <- solve(matrix(c(1,0.1,0.1,1),2,2))
> mu1 <- c(0,0)
> mu2 <-c(6,6)
>
> ## performing Gibbs sampling
> mc_mvn <- gibbs_met(log_f=log_pdf_twonormal,2,x0=c(0,0),iters_mc=4000,iters_met=5,
+                     stepsizes_met=c(0.5,0.5),mu1=mu1,mu2=mu2,A1=A1,A2=A2)
>
> postscript("mc_mvn_farmix.eps",width=7,height=8,horiz=FALSE)
>
> par(mfrow=c(2,2), oma=c(0,0,2,0))
>
> ## looking at the trace of Markov chain in the first 100 iterations
> plot(mc_mvn[,1],mc_mvn[,2],type="b",pch=20,
+      main="Markov chain trace of both variables")
>
> ## looking at the trace of Markov chain for a variable
> plot(mc_mvn[,1],type="b",pch=20, main="Markov chain trace of the 1st variable")
```

```
>
> ## looking at the trace of Markov chain for a variable
> plot(mc_mvn[,2],type="b",pch=20, main="Markov chain trace of the 2nd variable")
>
> ## looking at the ACF of the samples for a variable
> acf(mc_mvn[,1],main="ACF plot of the 1st variable")
>
> title(main="Gibbs sampling for a mixture of two bivariate normal distributions
+ with locations (0,0) and (6,6)", outer=TRUE)
>
> dev.off()
null device
          1
>
> ## checking the correlation of samples
> cat("The sample correlation is",cor(mc_mvn[-(1:50),1],mc_mvn[-(1:50),2]),"\n")
The sample correlation is 0.9094759
>
>
> ###########################################################################
> ## Sampling from mixture bivariate normal distributions with Metropolis method
> ###########################################################################
>
> ###########################################################################
> ## set parameters defining a mixture bivariate distribution
> A1 <- solve(matrix(c(1,0.1,0.1,1),2,2))
> A2 <- solve(matrix(c(1,0.1,0.1,1),2,2))
> mu1 <- c(0,0)
> mu2 <-c(6,6)
>
> ## performing Gibbs sampling
> mc_mvn <- met_gaussian(log_f=log_pdf_twonormal,p=2,x0=c(0,0),iters=4000,
+                   iters_per.iter=10,stepsizes=c(1,1),
+                   mu1=mu1,mu2=mu2,A1=A1,A2=A2)
>
> postscript("mc_mvn_farmix_met.eps",width=7,height=8,horiz=FALSE)
>
> par(mfrow=c(2,2), oma=c(0,0,2,0))
>
> ## looking at the trace of Markov chain in the first 100 iterations
> plot(mc_mvn[,1],mc_mvn[,2],type="b",pch=20,
+      main="Markov chain trace of both variables")
>
> ## looking at the trace of Markov chain for a variable
> plot(mc_mvn[,1],type="b",pch=20, main="Markov chain trace of the 1st variable")
>
```

```
> ## looking at the trace of Markov chain for a variable
> plot(mc_mvn[,2],type="b",pch=20, main="Markov chain trace of the 2nd variable")
>
> ## looking at the ACF of the samples for a variable
> acf(mc_mvn[,1],main="ACF plot of the 1st variable")
>
> title(main="Sampling with Metropolis method for a mixture of two bivariate normal
+ distributions with locations (0,0) and (6,6)", outer=TRUE)
>
> dev.off()
null device
          1
>
> ## checking the correlation of samples
> cat("The sample correlation is",cor(mc_mvn[-(1:50),1],mc_mvn[-(1:50),2]),"\n")
The sample correlation is 0.9056469
>
>
>
> ##############################################################################
> ##     demonstration by sampling from a toy distribution
> ##############################################################################
>
> log_pdf_ring <- function(x,r1,r2)
+ {   d <- sqrt(sum(x^2))
+     if( d < r2 & d > r1 ) 0
+     else -Inf
+ }
>
> ##############################################################################
> mc_ring <- gibbs_met(log_f=log_pdf_ring,2,x0=c(2.3,0),iters_mc=1000,iters_met=5,
+                      stepsizes_met=c(0.5,0.5), r1=2, r2=2.5)
>
> postscript("mc_ring.eps",width=7.5,height=4.5,horiz=FALSE)
>
> par(mfrow=c(1,2))
>
> ## looking at the trace of Markov chain in the first 100 iterations
> plot(mc_ring[,1],mc_ring[,2],type="b",pch=20,
+      main="Markov chain trace of both variables")
>
>
> ## looking at the ACF of the samples for a variable
> acf(mc_ring[,1],main="ACF plot of the 1st variable")
>
> dev.off()
```

```
null device
          1
>
> ## checking the correlation of samples
> cat("The sample correlation is",cor(mc_ring[-(1:50),1],mc_ring[-(1:50),2]),"\n")
The sample correlation is -0.00767716
>
```

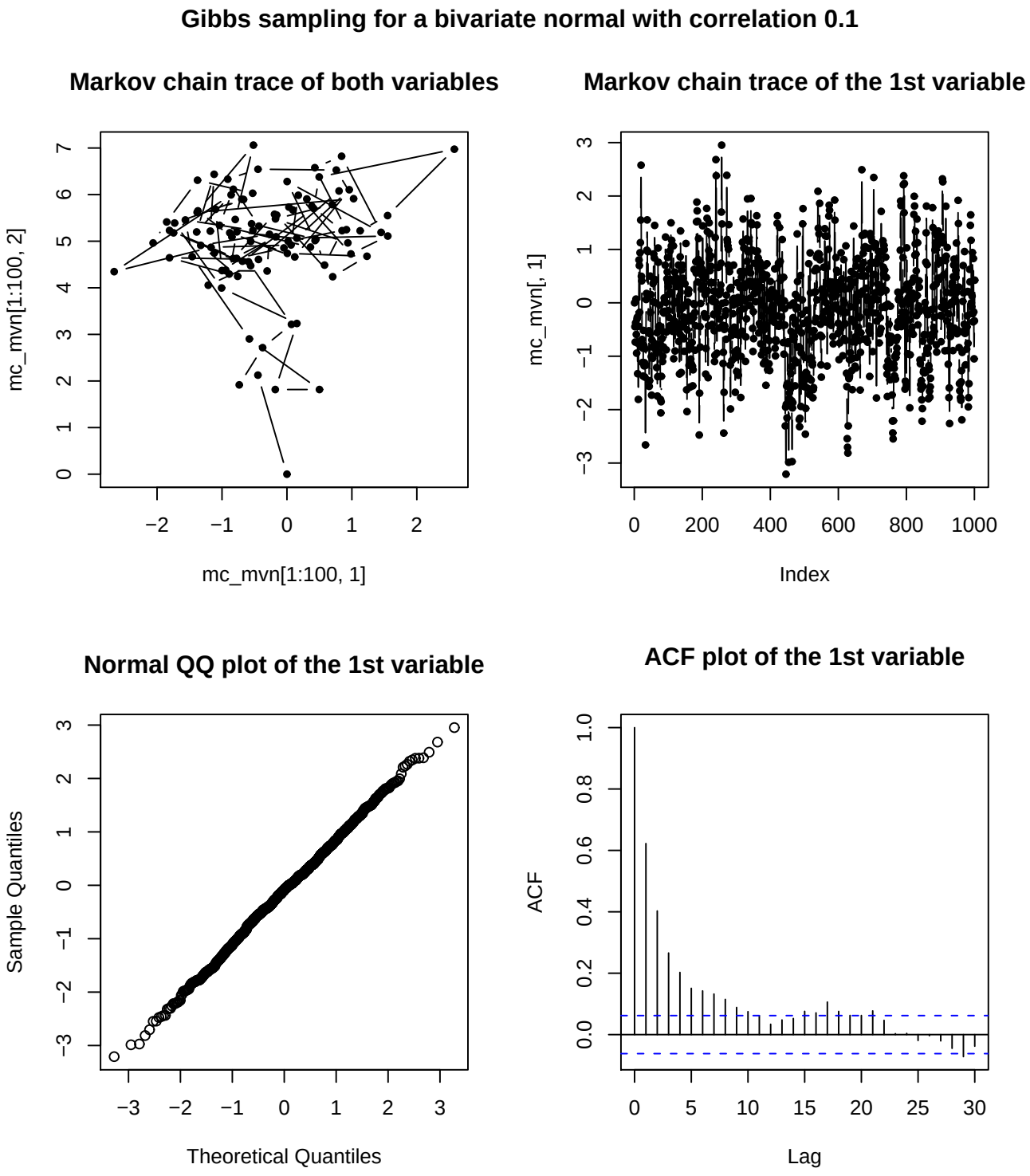


Figure 8.5: The plots displaying the samples drawn from a bivariate normal distribution with correlation 0.1, using Gibbs sampling with each variable sampled by Metropolis method with a Gaussian proposal distribution.

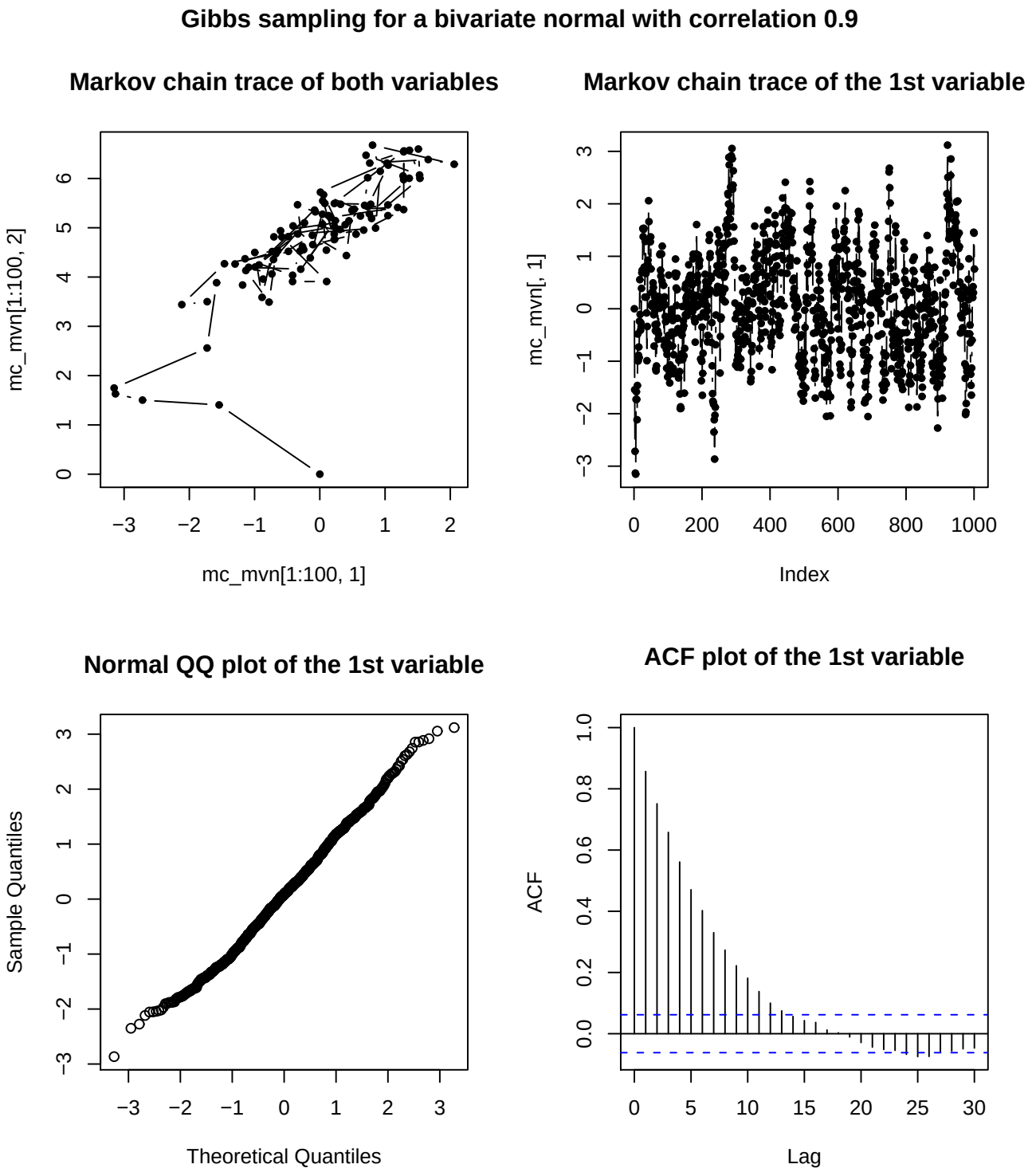


Figure 8.6: The plots displaying the samples drawn from a bivariate normal distribution with correlation 0.9, using Gibbs sampling with each variable sampled by Metropolis method with a Gaussian proposal distribution.

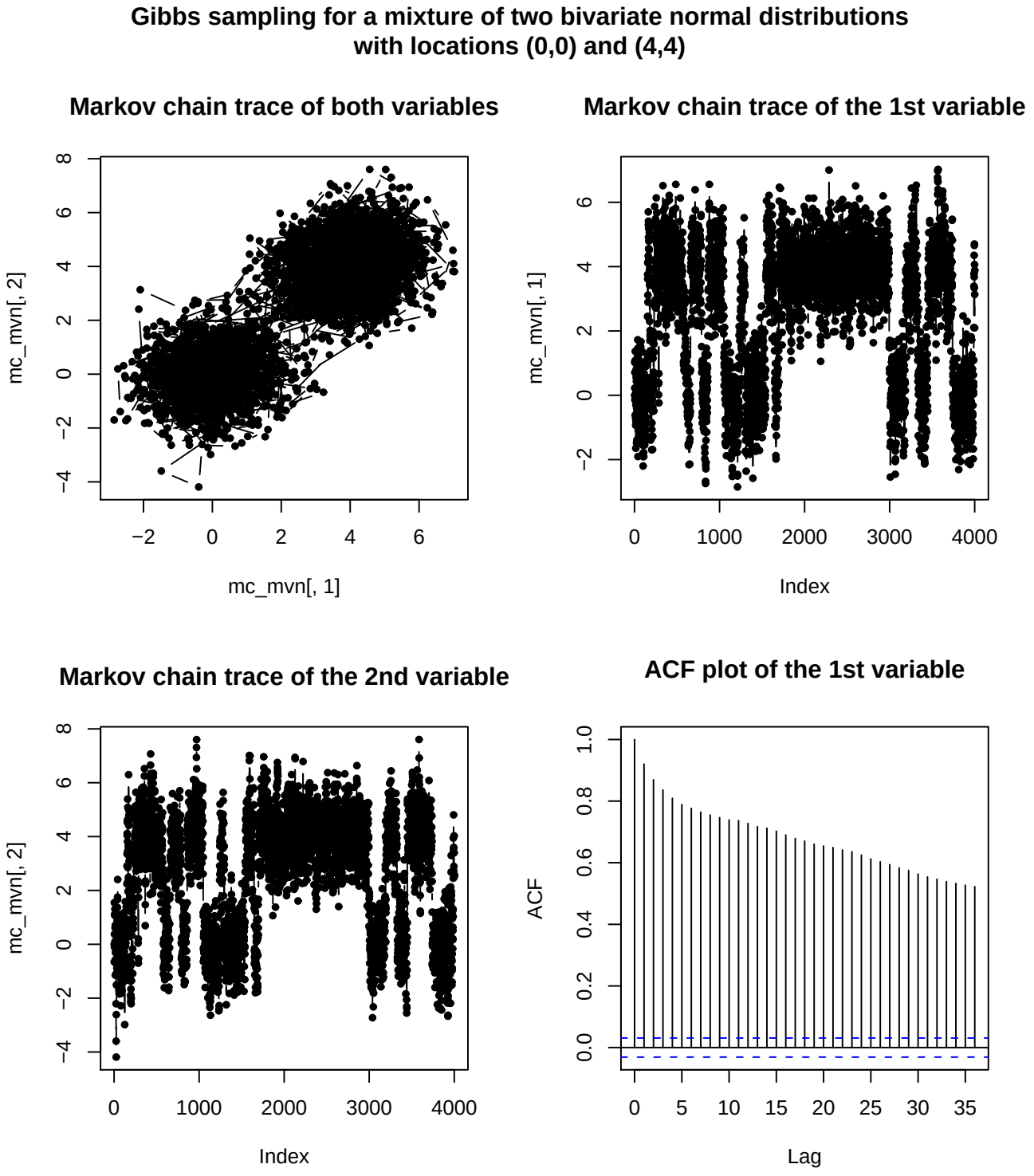


Figure 8.7: The plots displaying the samples drawn from a mixture of two bivariate normal distributions, using Gibbs sampling with each variable sampled by Metropolis method with a Gaussian proposal distribution.

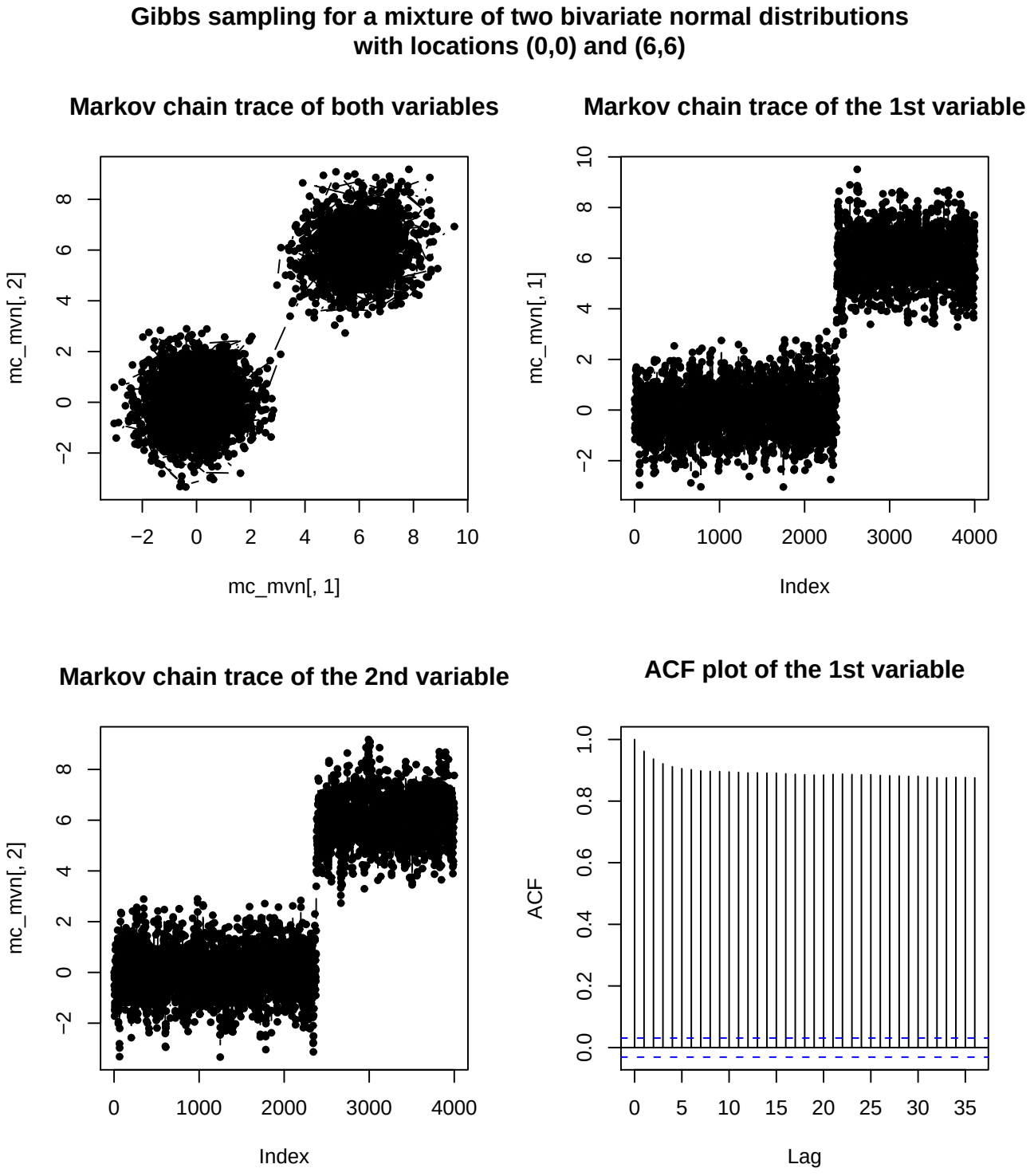


Figure 8.8: The plots displaying the samples drawn from a mixture of two bivariate normal distributions, using Gibbs sampling with each variable sampled by Metropolis method with a Gaussian proposal distribution.

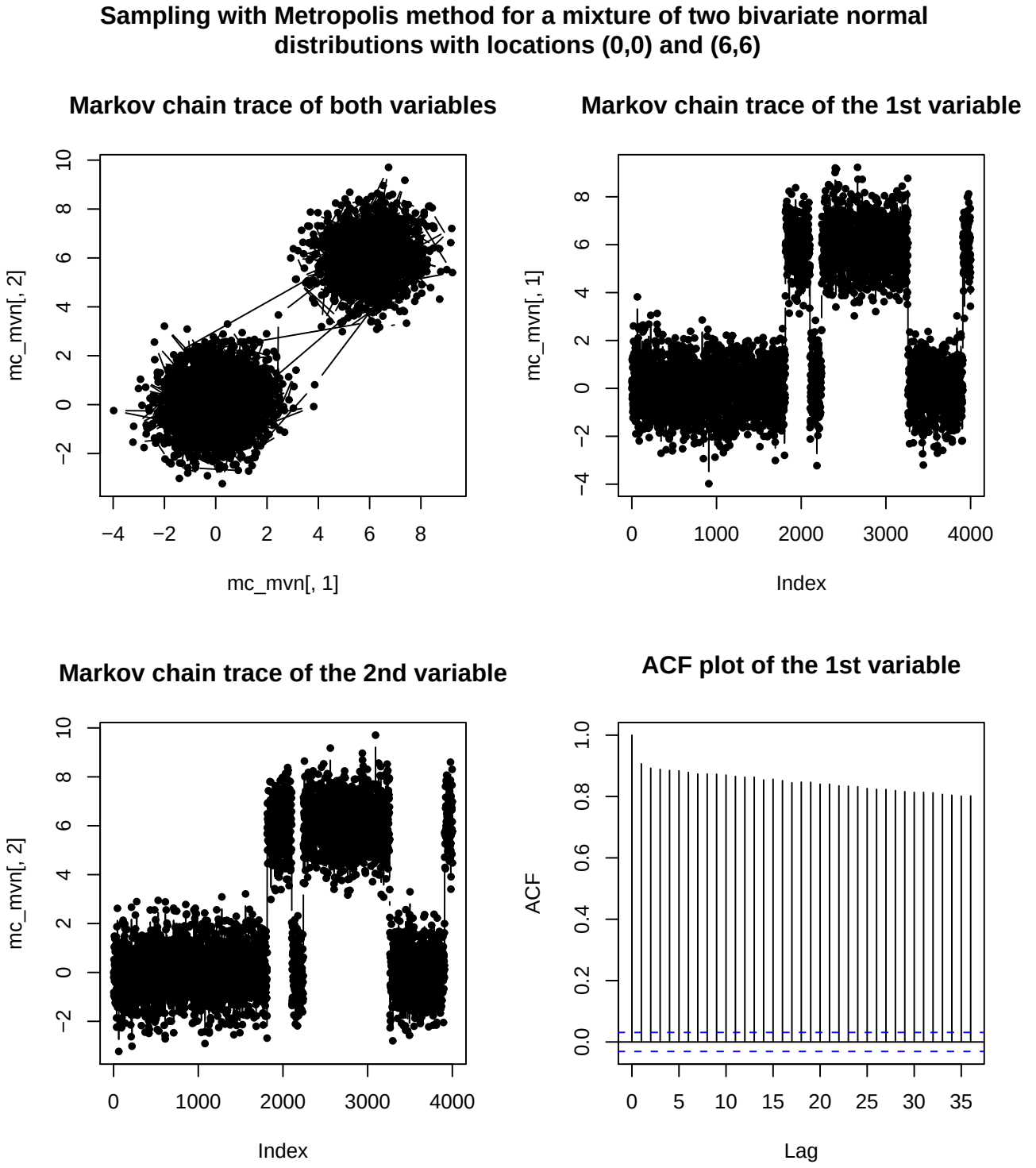


Figure 8.9: The plots displaying the samples drawn from a mixture of two bivariate normal distributions, using Metropolis method with a multivariate Gaussian proposal distribution.

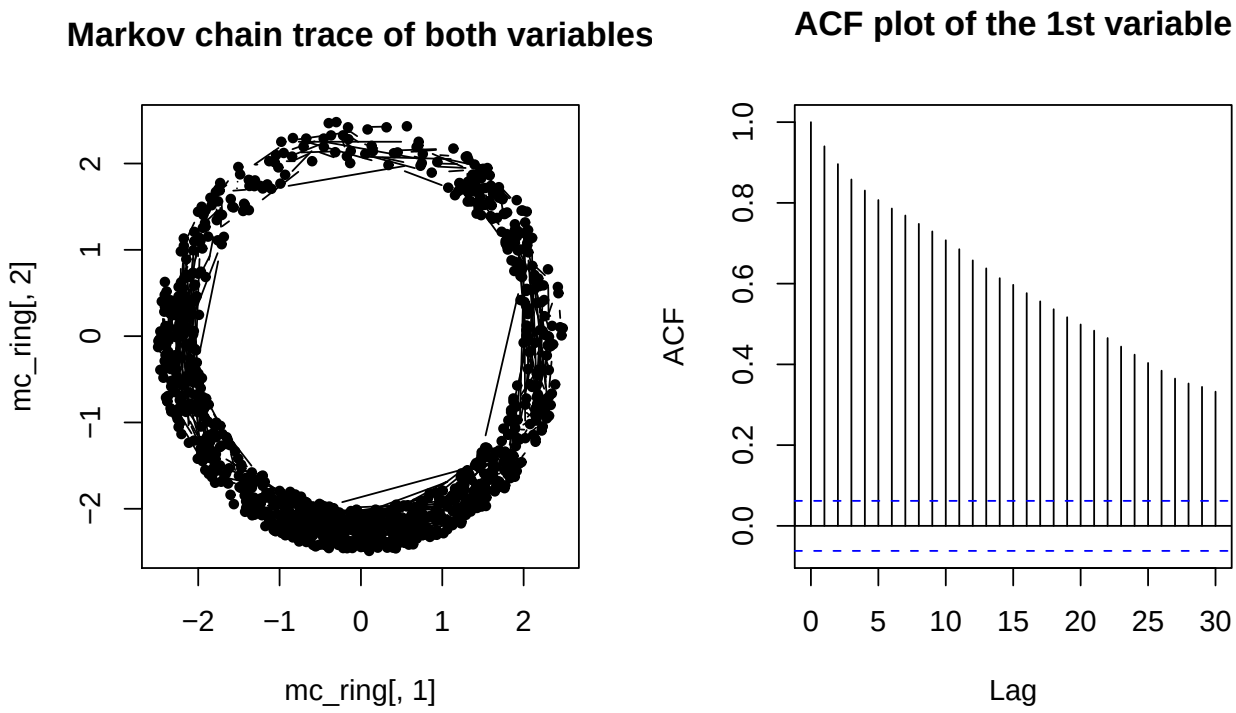


Figure 8.10: The plots displaying the samples drawn from uniform distribution over region $\{(x, y)|2 < \sqrt{x^2 + y^2} < 2.5\}$, using Gibbs sampling with each variable sampled by Metropolis method with a Gaussian proposal distribution.