HW4_xliu96

Xueying Liu

10/12/2020

Problem 1

```r
set.seed(1256)
theta <- as.matrix(c(1,2),nrow=2)
X <- cbind(1,rep(1:10,10))
m=nrow(X)
h <- X%*%theta+rnorm(100,0,0.2)
x <- X[,2]
theta <- matrix(NA,ncol = 2,nrow = 10000)

#set the starting point (0.5,0.5)
theta[1,] <- c(0.5,0.5)
# define h0(x)=theta0+theta1*x
h0 <- function(x,para){
return(para[1]+para[2]*x)
}
# set step size alpha=0.01, tolerance=0.00001
step <- 0.01
tolerance <- 0.00001
# gradient descent algorithm
for(i in 2:10000){
theta[i,1]=theta[i-1,1]-step*mean(h0(x,theta[i-1,])-h)
theta[i,2]=theta[i-1,2]-step*mean((h0(x,theta[i-1,])-h)*x)
if((theta[i,1]-theta[i-1,1])<tolerance && (theta[i,2]-theta[i-1,2])<tolerance ){
print(theta[i,])
break}
}
```

```
## [1] 0.9648193 2.0022455
```

```r
# using lm()
lm(h~X)
```

```
##
## Call:
## lm(formula = h ~ X)
##
## Coefficients:
## (Intercept)           X1           X2
##      0.9696           NA       2.0016
```

By implementing this algorithm, we can get $\theta_0 = 0.9648193, \quad \theta_1 = 2.0022455$. Compared with the result of $lm()$ function in R, we can find that their differences are smaller than 0.01.

Problem 2

```
#set the range of starting point +/- 1 from the true
theta0 <- seq(0.96-1,0.96+1,length.out = 100)
theta1 <- seq(2-1,2+1,length.out = 100)
#10000 different combinations of start values
grid <- as.matrix(expand.grid(theta0,theta1))

# set step size and tolerance
step <- 1e-5
tolerance <- 1e-9

# gradient descent algorithm
graddesc <- function(thetastart){
  set.seed(1256)
  theta <- as.matrix(c(1,2),nrow=2)
  X <- cbind(1,rep(1:10,10))
  h <- X%*%theta+rnorm(100,0,0.2)
  x <- X[,2]
  h0 <- function(x,theta0,theta1){
    return(theta0+theta1*x)
  }

  thetastart <- thetastart
  theta0.old <- thetastart[1]
  theta1.old <- thetastart[2]
  theta0.new <- theta0.old - step*mean(h0(x,theta0.old,theta1.old)-h)
  theta1.new <- theta1.old - step*mean((h0(x,theta0.old,theta1.old)-h)*x)
  iter <- 1
  while((abs(theta1.new-theta1.old)>tolerance) && (abs(theta0.new-theta0.old)>tolerance)){
    theta0.old <- theta0.new
    theta1.old <- theta1.new
    theta0.new <- theta0.old-step*mean(h0(x,theta0.old,theta1.old)-h)
    theta1.new <- theta1.old-step*mean((h0(x,theta0.old,theta1.old)-h)*x)
    iter <- iter + 1
    if(iter>50000) break
  }
  return(c(theta0.new,theta1.new,iter,thetastart))
}
```

```
# do parallel in 8 cores
cl<-makeCluster(8)
registerDoParallel(cl)
time.2 <- system.time(result.2 <-unlist(parApply(cl, grid, 1, graddesc)))
stopCluster(cl)
```

```
theta0.mean <- mean(result.2[1,])
theta1.mean <- mean(result.2[2,])
theta0.sd <- sd(result.2[1,])
theta1.sd <- sd(result.2[2,])
```

	theta0	theta1
mean	0.9606470	2.002975

	theta0	theta1
sd	0.0561582	0.092821

part b

If we change our stop rule based on our knowledge of the true parameter that we can stop if we reach the nearly 0 neighborhood of the true parameter, we may have problem that it may not converge to that true parameter. A good way to run gradient descent algorithm is to try different step size and starting value.

part c

This algorithms has advantage that it chooses a direct path towards the minimum, but it also has disadvantages that it may converge at local minima and saddle points and has slower learning since an update is performed only after we go through all observations. Therefore, we should be careful and double check our results when we are using this algorithm.

Problem 3

I will rewrite the equation as

$$(X'X)\beta = X'y$$

and then using the R code:

```
beta = solve(t(X) %*% X, t(X) %*% y)
```

The reason that why we don't solve $Ax = b$ via invert and multiply is that invert A needs $2n^3$ flops and multiply $b = A^{-1}x$ needs $2n^2$ flops, therefore, the total cost is $2n^3 + 2n^2$ flops. However, if we solve $Ax = b$ via LU factorization, it costs $\frac{2}{3}n^3$ flops to factor $A = LU$, n^2 flops tp solve $Lz = b$, and n^2 flops to solve $Ux_j = z$. The total cost is $\frac{2}{3}n^3 + 2n^2$, indicating that we should avoid using inverting and multipling a matrix.

Problem 4

```
set.seed(12456)
G <- matrix(sample(c(0,0.5,1),size=16000,replace=T),ncol=10)
R <- cor(G) # R: 10 * 10 correlation matrix of G
C <- kronecker(R, diag(1600)) # C is a 16000 * 16000 block diagonal matrix
id <- sample(1:16000,size=932,replace=F)
q <- sample(c(0,0.5,1),size=15068,replace=T) # vector of length 15068
A <- C[id, -id] # matrix of dimension 932 * 15068
B <- C[-id, -id] # matrix of dimension 15068 * 15068
p <- runif(932,0,1)
r <- runif(15068,0,1)
```

part a

```
object.size(A)
```

```
## 112347224 bytes
```

```
object.size(B)
```

```
## 1816357208 bytes
```

The size of A and B is 112347224 and 1816357208 bytes.

```r
system.time(y<-p+A%*%solve(B)%*%(q-r))
```

It takes 13 minutes to calculate y on my computer.

part b

Instead of calculating A, we can calculate $solve(B, q-r)$ first and then left multiply it by A because we should avoid inverting a matrix directly in R.

For matrix C, since it is a 16000*16000 block diagonal matrix, we can decompose it using QR decomposition or LU decomposition.

part c

The R packages *bigmemory*, and *biganalytics* provide structures for working with matrices that are too large to fit into memory. *bigalgebra* contains functions for doing linear algebra with bigmemory structures.

```r
library(bigmemory)
C <- NULL
set.seed(12456)
G <- matrix(sample(c(0,0.5,1),size=16000,replace=T),ncol=10)
R <- cor(G) # R: 10 * 10 correlation matrix of G
C <- as.big.matrix(kronecker(R, diag(1600))) # C is a 16000 * 16000 block diagonal matrix
id <- sample(1:16000,size=932,replace=F)
q <- sample(c(0,0.5,1),size=15068,replace=T) # vector of length 15068
A <- C[id, -id] # matrix of dimension 932 * 15068
B <- C[-id, -id] # matrix of dimension 15068 * 15068
p <- runif(932,0,1)
r <- runif(15068,0,1)

system.time(p+A%*%(solve(B,(q-r))))
```

Using *as.big.matrix*() function to make C a big matrix object, we can see that it takes 9 mins to get y, which uses 4 mins less than the previous operation.

Problem 5

part a

```r
# Create a function that computes the proportion of successes in a vector
successporp <- function(x){
  n <- length(x)
  success <- 0
  for (i in 1:n) {
    if(x[i]==1) success<-success+1
  }
  successporp = success/n
  return(successporp)
}

# a <- sample(c(0,1),size=100, replace=TRUE)
# successporp(a)
```

part b

```
set.seed(12345)
P4b_data <- matrix(rbinom(10, 1, prob = (31:40)/100), nrow = 10, ncol = 10, byrow = FALSE)
```

part c

```
## apply successporp function by column
apply(P4b_data,2,successporp)
```

```
##  [1] 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6
```

```
## apply successporp function by row
apply(P4b_data,1,successporp)
```

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

We found that the proportion of success in P4b_data by column is 0.6 for all 10 columns, and the proportion of success in P4b_data by row is either 1 or 0. This is because when we use *matrix*() function to create a matrix, we only set the value of the first column but ask to generate 10 columns in that matrix, therefore, it just set the rest columns same as the first column and the success proportion for each column is just that of the first one. Since the value of the same row is the same, its success proportion is either 1 or 0 depends on the value of first cloumn.

part d

```
## create a function generate outcomes of 10 flips of a coin
set.seed(123456)
flip10 <- function(p){
  rbinom(10, 1, prob = p)
}

## Create a vector of the desired probabilities
prob <- data.frame(seq(0.31,0.40,0.01))

## Create a matrix to simulate 10 flips of a coin with varying degrees of "fairness" (columns = probabi
data <- apply(prob,1,flip10)
colnames(data) <- seq(0.31,0.40,0.01)

## apply successporp function by column
columnprob <- apply(data,2,successporp)

## apply successporp function by row
rowprob <- apply(data,1,successporp)
table <- cbind(rbind(data,columnprob),rowprob)
kable(table)
```

	0.31	0.32	0.33	0.34	0.35	0.36	0.37	0.38	0.39	0.4	rowprob
	1.0	1.0	0.0	0.0	0.0	0.0	0	0.0	0.0	0.0	0.2
	1.0	0.0	0.0	1.0	1.0	1.0	0	1.0	0.0	0.0	0.5
	0.0	1.0	0.0	0.0	1.0	0.0	0	1.0	1.0	0.0	0.4
	0.0	1.0	0.0	0.0	0.0	1.0	0	0.0	1.0	0.0	0.3
	0.0	1.0	0.0	1.0	0.0	1.0	0	1.0	0.0	0.0	0.4
	0.0	1.0	1.0	0.0	0.0	1.0	0	0.0	0.0	0.0	0.3

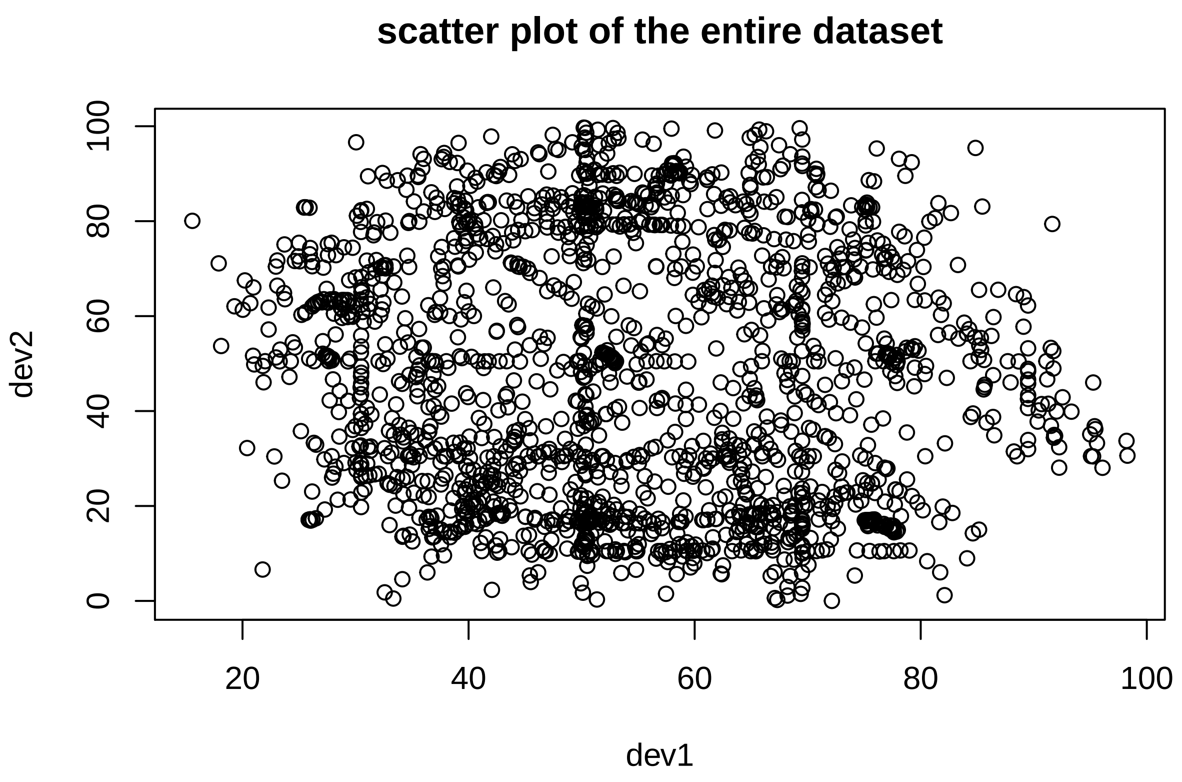
	0.31	0.32	0.33	0.34	0.35	0.36	0.37	0.38	0.39	0.4	rowprob
	0.0	1.0	1.0	1.0	0.0	1.0	0	1.0	0.0	1.0	0.6
	0.0	0.0	1.0	1.0	1.0	1.0	0	0.0	0.0	1.0	0.5
	1.0	0.0	1.0	1.0	1.0	0.0	0	0.0	0.0	0.0	0.4
	0.0	1.0	0.0	0.0	0.0	0.0	0	1.0	0.0	0.0	0.2
columnprob	0.3	0.7	0.4	0.5	0.4	0.6	0	0.5	0.2	0.2	0.2

Problem 6

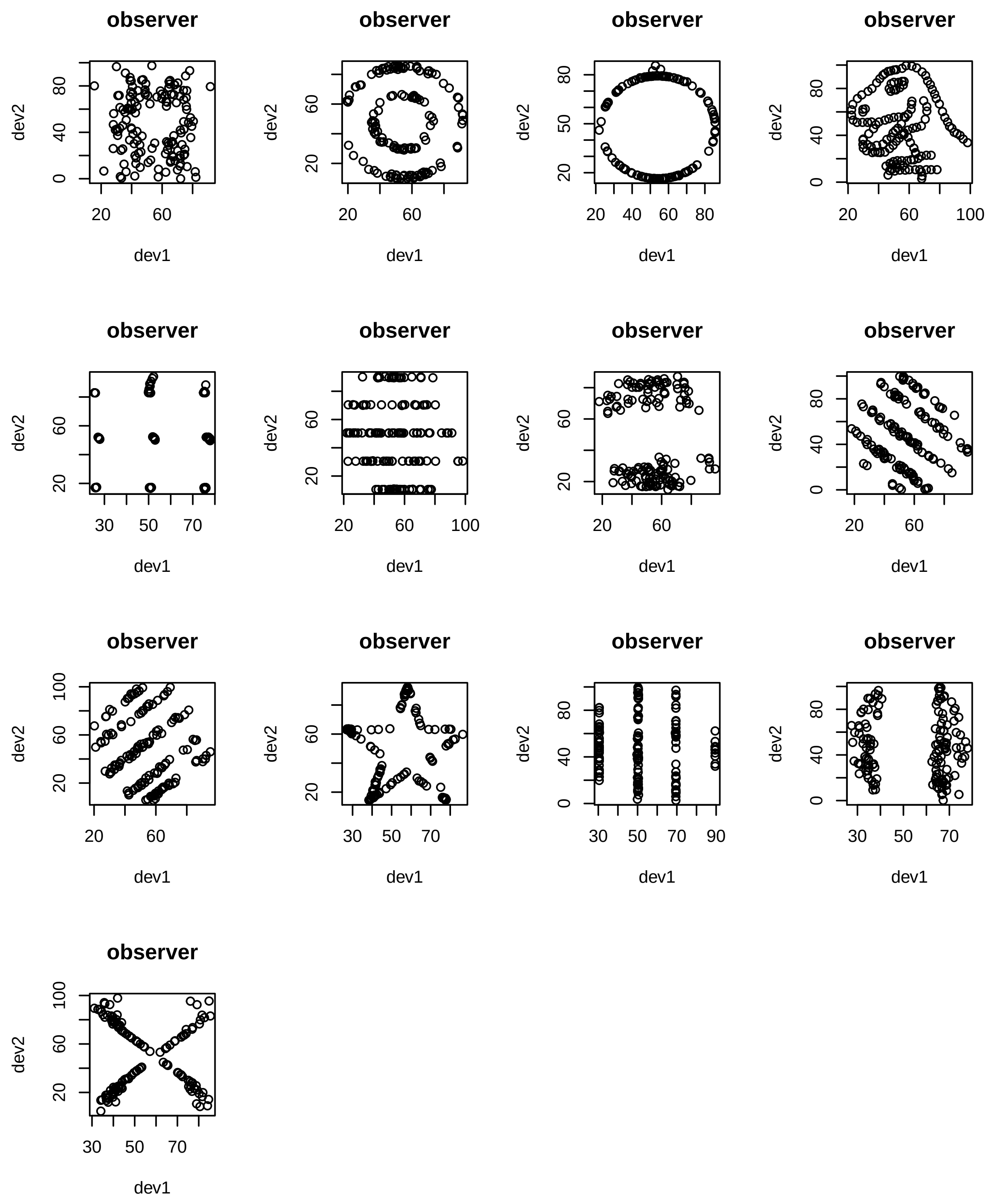
```r
observer <- readRDS("HW3_data.rds")
colnames(observer)[2:3] <- c("x","y")

observerlist <- list()
# create a function to plot scatter plot
myscatter <- function(data,xlab,ylab,title){
  plot(data$x,data$y,xlab = xlab,ylab = ylab,main = title)
}
```

2

```r
# a single scatter plot of the entire dataset
myscatter(observer,"dev1","dev2","scatter plot of the entire dataset")
```



```r
# a seperate scatter plot for each observer
par(mfrow=c(4,4))
uniqobserver <- factor(observer$Observer)
sapply(split(observer,uniqobserver),FUN=myscatter,xlab="dev1",ylab="dev2",title="observer")
```



Problem 7

part a

```
library(downloader)
download("http://www.farinspace.com/wp-content/uploads/us_cities_and_states.zip",
         dest="us_cities_states.zip")
unzip("us_cities_states.zip")

library(data.table)
states <- data.frame(fread(input = "us_cities_and_states/states.sql",skip = 23,
                           sep = "'", sep2 = ",", header = F, select = c(2,4)))
#limit to 50 states
states <- states[-c(which(states$V2=="District of Columbia" )),]

cities_extended <- fread(input = "us_cities_and_states/cities_extended.sql",skip = 23,
                         sep = "'", sep2 = ",", header = F, select = c(2,4))
#limit to 50 states
cities_extended <- cities_extended[-c(which(cities_extended$V4=="DC" ),which(cities_extended$V4=="PR" )
```

part b

```
cities_extended$V4 <- as.factor(cities_extended$V4)
countcities <- aggregate(cities_extended$V2,by=list(cities_extended$V4),FUN=length)
countcities <- cbind(countcities,tolower(states$V2))
colnames(countcities) <- c("Abbreviation","citycounts","state")
head(countcities)
```

```
##   Abbreviation citycounts      state
## 1           AK        273     alaska
## 2           AL        838    alabama
## 3           AR        709   arkansas
## 4           AZ        532    arizona
## 5           CA       2651 california
## 6           CO        659   colorado
```

part c

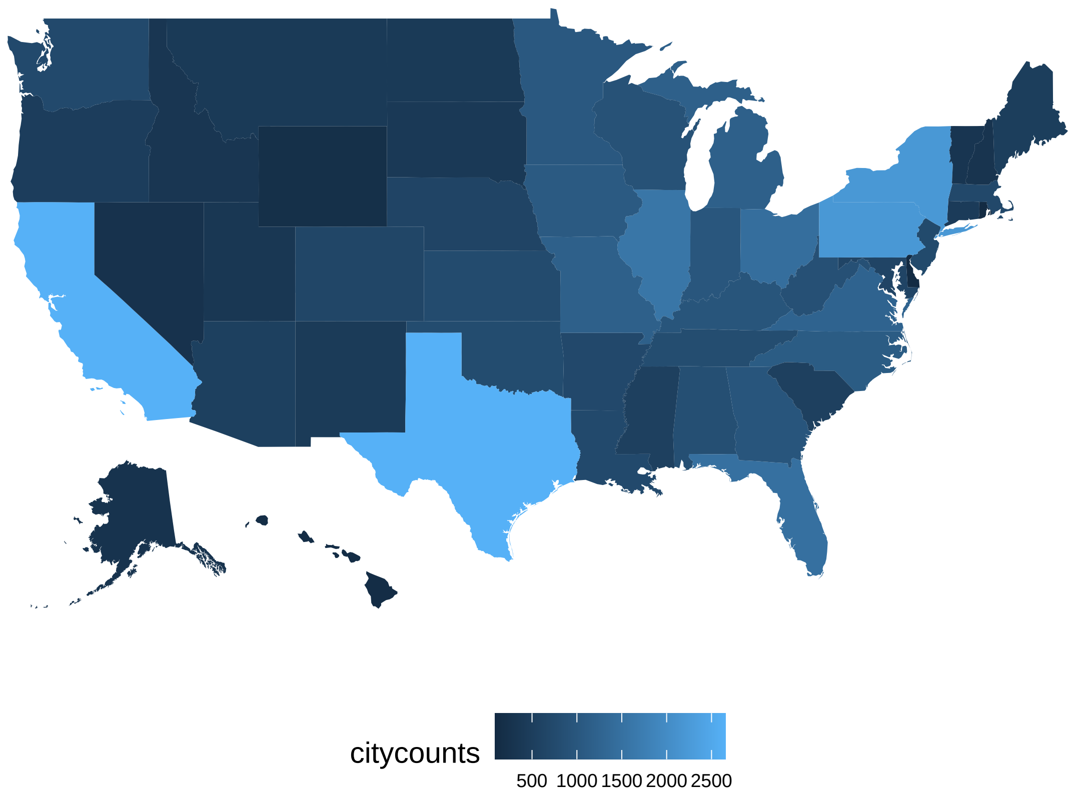
```
## counts the number of occurances of a letter in a string
letter_count <- data.frame(matrix(NA,nrow=50, ncol=26))

getCount <- function(x,y){
  temp <- strsplit(x,"")[[1]]
  count <- 0
  for(i in 1:length(temp)){
    if(identical(temp[i],y)) count<-count +1
  }
  return(count)
}

for(i in 1:26){
  letter_count[,i] <- apply(as.matrix(states$V2),1,getCount,y=letters[i])
}
```

```
colnames(letter_count) <- letters
row.names(letter_count) <- states$V2
head(letter_count,3)
```

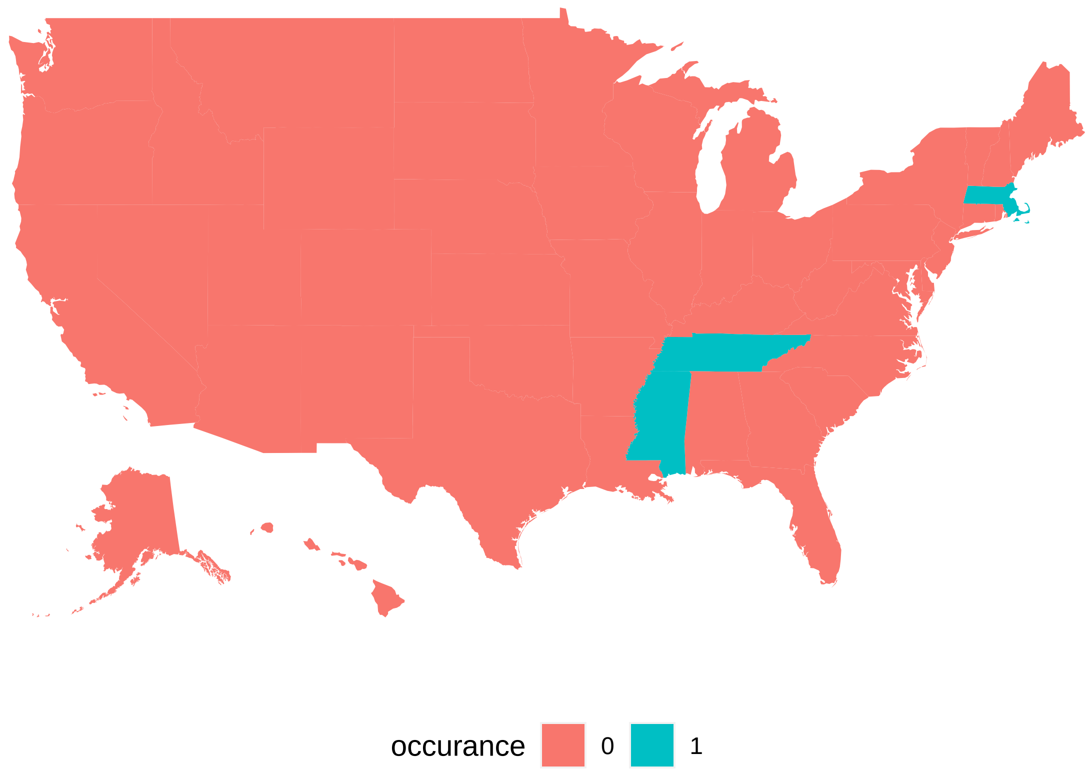
```
##          a b c d e f g h i j k l m n o p q r s t u v w x y z
## Alaska   2 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0
## Alabama  3 1 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
## Arkansas 2 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 1 2 0 0 0 0 0 0 0
```

part d

```
## Map 1 colored by count of cities within the state
data("fifty_states")

p <- ggplot(countcities, aes(map_id = state)) +
geom_map(aes(fill = citycounts), map = fifty_states) +
expand_limits(x = fifty_states$long, y = fifty_states$lat) +
coord_map() +
scale_x_continuous(breaks = NULL) +
scale_y_continuous(breaks = NULL) +
labs(x = "", y = "") +
theme(legend.position = "bottom",legend.text = element_text(size = 7),
panel.background = element_blank())
p
```



```r
## Map 2 highlight only those have more than 3 occurances of ANY letter
highlight <- matrix(NA,nrow = 50,ncol = 2)
highlight[,1] <- tolower(states$V2)
highlight[which(letter_count >3, arr.ind = T)[,1],2]=1
highlight[-which(letter_count >3, arr.ind = T)[,1],2]=0
highlight <- data.frame(highlight)
colnames(highlight) <- c("state","occurance")

p <- ggplot(highlight, aes(map_id = state)) +
geom_map(aes(fill = occurance), map = fifty_states) +
expand_limits(x = fifty_states$long, y = fifty_states$lat) +
coord_map() +
scale_x_continuous(breaks = NULL) +
scale_y_continuous(breaks = NULL) +
labs(x = "", y = "") +
theme(legend.position = "bottom",
panel.background = element_blank())
p
```



Occcurance equals 1 indicates states that have more than 3 occurances of any letter in thier name.

Problem 8

part a

The reason is that when creating df08 matrix, he used $cbind(logapple08, logrm08)$. However, the colnames of df08 is not "logapple08" and "logrm08", instead they are "AAPL.Adjusted" and "IXIC.Adjusted". So we can either define the correct column names before running the bootstrap, or we can change the formula in $lm(logapple08\ logrm08, data = bootdata)$ to $lm(AAPL.Adjusted\ IXIC.Adjusted, data = bootdata)$.

```
df08<-cbind(logapple08,logrm08)
colnames(df08) <- c("logapple08","logrm08") ## define the right column names

set.seed(666)
Boot=1000
sd.boot=rep(0,Boot)
for(i in 1:Boot){
# nonparametric bootstrap
bootdata=df08[sample(nrow(df08), size = 251, replace = TRUE),]
sd.boot[i]= coef(summary(lm(logapple08~logrm08, data = bootdata)))[2,2]
}
summary(sd.boot)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## 0.04131 0.05556 0.05940 0.05968 0.06361 0.08308
```

part b

```
#bootstrap the Sensory data to get non-parametric estimates of the parameters
n <- dim(sensory_data_tidy)[1]
p <- 1/n
equalweight <- rep(p,n) #assign equal weights to each data for sampling

# generate function to get lm coef
set.seed(1234567)
lmcoef<- function(n){
  ind <- sample(1:n,size = n,replace = TRUE,prob = equalweight) # equal weight to get balanced data
  temp <- sensory_data_tidy[ind,]
  temp.model <- lm(value~Operator, data = temp)
  coeff <- matrix(coefficients(temp.model),ncol = 5)
  return(coeff)
}


# generate bootstrap function
myboot <- function(B,n){
  results <- matrix(NA, nrow = B,ncol = 5,dimnames = list(NULL,c("Intercept","operator2","operator3","op
  for(b in 1:B){
    results[b,] <- lmcoef(n)
  }
  results <- data.frame(results)
  return(apply(results,2,mean))
}

# begin bootstrap and record time
B <- 100 #number of bootstraps
```

```
result.9.b <- myboot(B=B,n=n)
time.9.b <- system.time(myboot(B,n))
result.9.b
```

```
##  Intercept  operator2  operator3  operator4  operator5
##  4.6190441  0.4074812 -0.4219788  0.5100530 -0.3246397
```

```
time.9.b
```

```
##    user  system elapsed
##    0.19    0.00    0.19
```

We can get the parameter estimator through bootstrapping by taking the average of the 100 results.

part c

```
cores <- detectCores()-1
cl <- makeCluster(cores)
registerDoParallel(cl)

n <- dim(sensory_data_tidy)[1]
B <- 100
coef <- c()
results <- foreach(b=1:B,.combine = 'rbind') %dopar%{
   coef[b] <- lmcoef(n)
}

results <- data.frame(results)
result.9.c <- apply(results,2,mean)
time.9.c <- system.time(foreach(b=1:B,.combine = 'rbind') %dopar%{coef[b] <- lmcoef(n)})
stopCluster(cl)
result.9.c
```

```
##         X1         X2         X3         X4         X5
##  4.6027688  0.4809373 -0.4271739  0.5929349 -0.4032748
```

```
time.9.c
```

```
##    user  system elapsed
##    0.11    0.02    0.22
```

	Bootstrap	Parallel
Intercept	4.6190441	4.6027688
operator2	0.4074812	0.4809373
operator3	-0.4219788	-0.4271739
operator4	0.5100530	0.5929349
operator5	-0.3246397	-0.4032748
elapsed_time	0.1900000	0.2200000

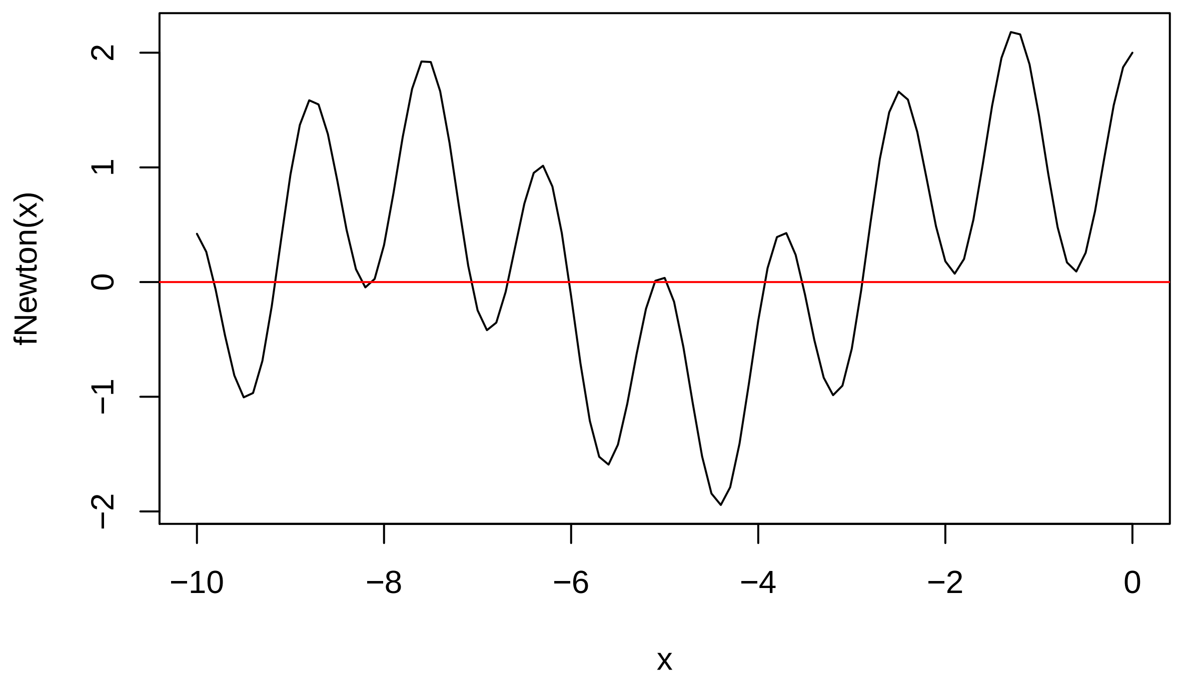
It is obvious that run the bootstrap in parallel taks less time than just run it directly.

Problem 9

part a

The function is approximately periodic when x<0, therefore, we could only consider the solution between x=-10 and x=0. From the plot, we can see that there are 12 roots.

```r
# plot of the function
fNewton <- function(x) 3^x - sin(x) + cos(5*x)
curve(fNewton,from = -10, to = 0)
abline(h=0,col="red")
```



```r
# Create a vector as a "grid" covering all the roots
grid <- as.matrix(seq(-10,0,length.out = 100))
findroot <- function(x,n,tol){
  iter <- 1
  itervalue <- c()
  while(iter <= n){
    x = x - fNewton(x)/Deriv(fNewton)(x)
    iter <- iter + 1
    itervalue <- c(itervalue,x)
  }

  if(abs(itervalue[n]-itervalue[n-1])<tol) return(itervalue[n])
}


time.10.a <- system.time(roots <- unlist(sapply(grid,findroot,n=50,tol=1e-5)))
```

```r
result.10.a <-unique(round(roots[which(-10<roots & roots<0)],3))
```

```r
time.10.a
```

```
##    user  system elapsed 
##  223.59    0.39  234.74
```

```r
result.10.a
```

```
##  [1] -9.163 -9.817 -8.116 -8.247 -6.676 -7.068 -4.972 -6.021 -5.107 -3.930
## [11] -3.529 -2.887
```

part b

```r
# using the parApply with 8 workers
cl<-makeCluster(8)
registerDoParallel(cl)

grid <- as.matrix(seq(-10,0,length.out = 100))
findroot <- function(x,n,tol){
  library(Deriv)
  fNewton <- function(x) 3^x - sin(x) + cos(5*x)
  iter <- 1
  itervalue <- c()
  while(iter <= n){
    x = x - fNewton(x)/Deriv(fNewton)(x)
    iter <- iter + 1
    itervalue <- c(itervalue,x)
  }

  if(abs(itervalue[n]-itervalue[n-1])<tol) return(itervalue[n])
}
time.10.b <- system.time(roots <- unlist(parApply(cl,grid,1,findroot,n=50, tol=1e-5)))
stopCluster(cl)
result.10.b <- unique(round(roots[which(-10<roots & roots<0)],3))
```

	Direct	Parallel
root1	-9.163	-9.163
root2	-9.817	-9.817
root3	-8.116	-8.116
root4	-8.247	-8.247
root5	-6.676	-6.676
root6	-7.068	-7.068
root7	-4.972	-4.972
root8	-6.021	-6.021
root9	-5.107	-5.107
root10	-3.930	-3.930
root11	-3.529	-3.529
root12	-2.887	-2.887
elapsed_time	234.740	53.890

We can see that the roots from two parts are the same, and using parallel computing do save times.