HW4_tdolkar

Due Wednesday Sep 30

Tsering Dolkar

10/11/2020

For each assignment, turn in by the due date/time. Late assignments must be arranged prior to submission. In every case, assignments are to be typed neatly using proper English in Markdown.

The last couple of weeks, we spoke about vector/matrix operations in R, discussed how the apply family of functions can assist in row/column operations, and how parallel computing in R is enabled. Combining this with previous topics, we can write functions using our Good Programming Practices style and adhere to Reproducible Research principles to create fully functional, redable and reproducible code based analysis in R. In this homework, we will put this all together and actually analyze some data. Remember to adhere to both Reproducible Research and Good Programming Practices, ie describe what you are doing and comment/indent code where necessary.

R Vector/matrix manipulations and math, speed considerations R's Apply family of functions Parallel computing in R, foreach and dopar

Problem 1: Using the dual nature to our advantage

Sometimes using a mixture of true matrix math plus component operations cleans up our code giving better readibility. Suppose we wanted to form the following computation:

- $while(abs(\Theta_0^i - \Theta_0^{i-1})$ AND $abs(\Theta_1^i - \Theta_1^{i-1}) > tolerance)$ {

$$\begin{aligned} \Theta_0^i &= \Theta_0^{i-1} - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_0(x_i) - y_i) \\ \Theta_1^i &= \Theta_1^{i-1} - \alpha \frac{1}{m} \sum_{i=1}^{m} ((h_0(x_i) - y_i) x_i) \end{aligned}$$

}

Where $h_0(x) = \Theta_0 + \Theta_1 x$.

Given $\mathbf{X}$ and $\vec{h}$ below, implement the above algorithm and compare the results with lm(h~0+$\mathbf{X}$). State the tolerance used and the step size, α.

```
set.seed(1256)

theta_curr <- double()
theta_prev <- double()

theta_curr <- as.matrix(c(1,2),nrow=2)
```

```
theta_prev <- as.matrix(c(0,0),nrow=2)

tolerance = 0.00000001
alpha = 0.00001
m = 100

X <- cbind(1,rep(1:10,10))
h <- X%*%theta_curr+rnorm(100,0,0.2)
```

```
i = 1
while(abs(theta_curr[1] - theta_prev[1]) > tolerance && abs(theta_curr[2] - theta_prev[2]) > tolerance)
  theta_prev <- theta_curr
  theta_curr <- theta_curr - alpha * ((1/m) * t(X) %*% (X %*% theta_curr - h))
}
theta_curr

#true parameter
lm(h~0+X)
```

Problem 2

The above algorithm is called Gradient Descent. This algorithm, like Newton's method, has "hyperparameters" that are determined outside the algorithm and there are no set rules for determing what settings to use. For gradient descent, you need to set a start value, a step size and tolerance.

a. Using a step size of $1e^{-7}$ and tolerance of $1e^{-9}$, try 10000 different combinations of start values for β_0 and β_1 across the range of possible β's +/-1 from true determined in Problem 1, making sure to take advantages of parallel computing opportunities. In my try at this, I found starting close to true took 1.1M iterations, so set a stopping rule for 5M. Report the min and max number of iterations along with the starting values for those cases. Also report the average and stdev obtained across all 10000 β's.

```
g_descent <- function(theta, h, X){
  theta_curr <- double()
  theta_prev <- double()

  theta_curr <- theta
  theta_prev <- as.matrix(c(theta[1] + 1, theta[2] + 1), nrow = 2)
  m <- length(X[,1])

  count <- 0
  step_size <- 1e-07
  tolerance <- 1e-09

  while(abs(theta_curr[1] - theta_prev[1]) > tolerance &&
        abs(theta_curr[2] - theta_prev[2]) > tolerance){
    theta_prev <- theta_curr
    theta_curr <- theta_curr - step_size * ((1/m) * t(X) %*% (X %*% theta_curr - h))
    count <- count + 1
    if(count > 5000000){
```

```
      break
    }
  }
  return(c(theta_curr, count))
}


theta <- as.matrix(c(1,2),nrow=2)
X <- cbind(1,rep(1:10,10))
h <- X%*%theta+rnorm(100,0,0.2)
p_startvalue <- cbind(runif(10000, min = 0, max = 2), runif(10000, min = 1, max = 3))

cores <- max(1, detectCores() - 1)
cl <- makeCluster(cores)
makeForkCluster(cores)
```

```
## socket cluster with 7 nodes on host 'localhost'
```

```
registerDoParallel(cl)

results <- foreach(i = 1:2, .combine = rbind) %dopar%{
  g_descent(p_startvalue[i,],h,X)
}
colnames(results) <- c("hbeta0","hbeta1","num_iter")
stopCluster(cl)
```

b. What if you were to change the stopping rule to include our knowledge of the true value? Is this a good way to run this algorithm? What is a potential problem?

c. What are your thoughts on this algorithm?

I think this is a pretty good algorithm to predict the parameters of a linear model. Although it is not as efficient and fast as the lm() function, I imagine this was one of the main go to method for estimating parameter of linear model in earlier times.

Problem 3: Inverting matrices

According to John Cook, if you want to solve: $(X'X)\hat{\beta} = X'\mathbf{y}$, even though books might write the problem as,

$$\hat{\beta} = (X'X)^{-1}X'\underline{y}$$

but that doesn't mean they expect you to calculate it that way. The first time you solve $(X'X)\hat{\beta} = X'\mathbf{y}$, you factor $(X'X)$ and save that factorization. Then when you solve for the next $X'\mathbf{y}$, the answer comes much faster. (Factorization takes $O(n^3)$ operations. But once the matrix is factored, solving $(X'X)\hat{\beta} = X'\mathbf{y}$ takes only $O(n^2)$ operations. Suppose $n = 1,000$. This says that once you've solved $(X'X)\hat{\beta} = X'\mathbf{y}$ for a new $X'\mathbf{y}$ 1,000 times faster than the first one.

```
#invert that matrix
time1 <- system.time(print(solve(t(X)%*%X)%*%t(X)%*%h))
```

```
##           [,1]
## [1,] 1.010763
## [2,] 2.000110
```

```
time1
```

```
##    user  system elapsed
##   0.001   0.000   0.001
```

```
#don't invert that matrix
time2 <- system.time(print(solve(t(X)%*%X, t(X)%*%h)))
```

```
##           [,1]
## [1,] 1.010763
## [2,] 2.000110
```

```
time2
```

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

Problem 4: Need for speed challenge

In this problem, we are looking to compute the following:

$$y = p + AB^{-1}(q - r) \tag{1}$$

Where A, B, p, q and r are formed by:

```
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)
C<-NULL #save some memory space
```

a.

How large (bytes) are A and B? Without any optimization tricks, how long does the it take to calculate y?

```
size_A <- object.size(A)
print(size_A)
```

```
## 112347224 bytes
```

```r
size_B <- object.size(B)
print(size_B)
```

```
## 1816357208 bytes
```

```r
#set up the variables for matrix operation
q <- as.vector(q)
r <- as.vector(r)
p <- as.vector(p)

naive_ytime <- system.time({p + A %*% solve(B) %*% (q - r)})

print(naive_ytime)
```

```
##    user  system elapsed
## 891.448   5.416 903.296
```

b.

How would you break apart this compute, i.e., what order of operations would make sense? Are there any mathematical simplifications you can make? Is there anything about the vectors or matrices we might take advantage of?

```r
library(Matrix)
```

```
##
## Attaching package: 'Matrix'

## The following objects are masked from 'package:tidyr':
##
##     expand, pack, unpack
```

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

print(y_time)
```

```
##    user  system elapsed
## 724.102   4.703 733.944
```

c.

Use ANY means (ANY package, ANY trick, etc) necessary to compute the above, fast. Wrap your code in "system.time({})", everything you do past assignment "C <- NULL".

```r
# library(bigmemory)
#
# fast_ytime <- system.time({p+as.big.matrix(A)%*%(as.big.matrix(solve(B)) %*% (q-r))})
#
# fast_ytime
```

Problem 5

a. Create a function that computes the proportion of successes in a vector. Use good programming practices.

```
proportion_of_success <- function(data){
  count = 0
  for(i in 1:length(data)){
    if(data[i] == 1){
      count = count + 1
    }
  }
  return(count/length(data))
}
```

b. Create a matrix to simulate 10 flips of a coin with varying degrees of "fairness" (columns = probability) as follows:

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

c. Use your function in conjunction with apply to compute the proportion of success in P4b_data by column and then by row.

```
p_col <- apply(P4b_data, 2, proportion_of_success)
p_col
```

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

```
p1 <- sum(p_col)/length(p_col)
p1
```

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

```
p_row <- apply(P4b_data, 1, proportion_of_success)
p_row
```

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

```
p2 <- sum(p_row)/length(p_row)
p2
```

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

The apply function takes the data and goes through every column and finds proportion of success of each column. Since the column of the matrix has varying occurrence of success and failure, we get a value < 1. When the apply function takes the data and goes through every row and finds proportion of success of each row, since we have rows with just 1's and 0's in the matrix, the proportion of success of individual rows are either 1 or 0.

d.

```
#a function whose input is a probability and output is a vector whose elements are
#the outcomes of 10 flips of a coin.
outcome_of_flips <- function(p){
  outcomes <- qbinom(p, 1, prob = (46:55)/100)
  return(outcomes)
}

#a vector of the desired probabilities.
desired_probabilities <- c(0.2, 0.5, 0.8, 0.5, 0.1, 0.7, 0.5, 0.3, 0.9, 0.8)

#create the matrix we really wanted in part b.
desired_matrix <- sapply(desired_probabilities, outcome_of_flips)

#Prove this has worked by using the function created in part a to compute and
#tabulate the appropriate marginal successes.
mS_col <- apply(desired_matrix, 2, proportion_of_success)
mS_col
```

```
##  [1] 0.0 0.5 1.0 0.5 0.0 1.0 0.5 0.0 1.0 1.0
```

```
mS_row <- apply(desired_matrix, 1, proportion_of_success)
mS_row
```

```
##  [1] 0.4 0.4 0.4 0.4 0.4 0.7 0.7 0.7 0.7 0.7
```

Problem 6

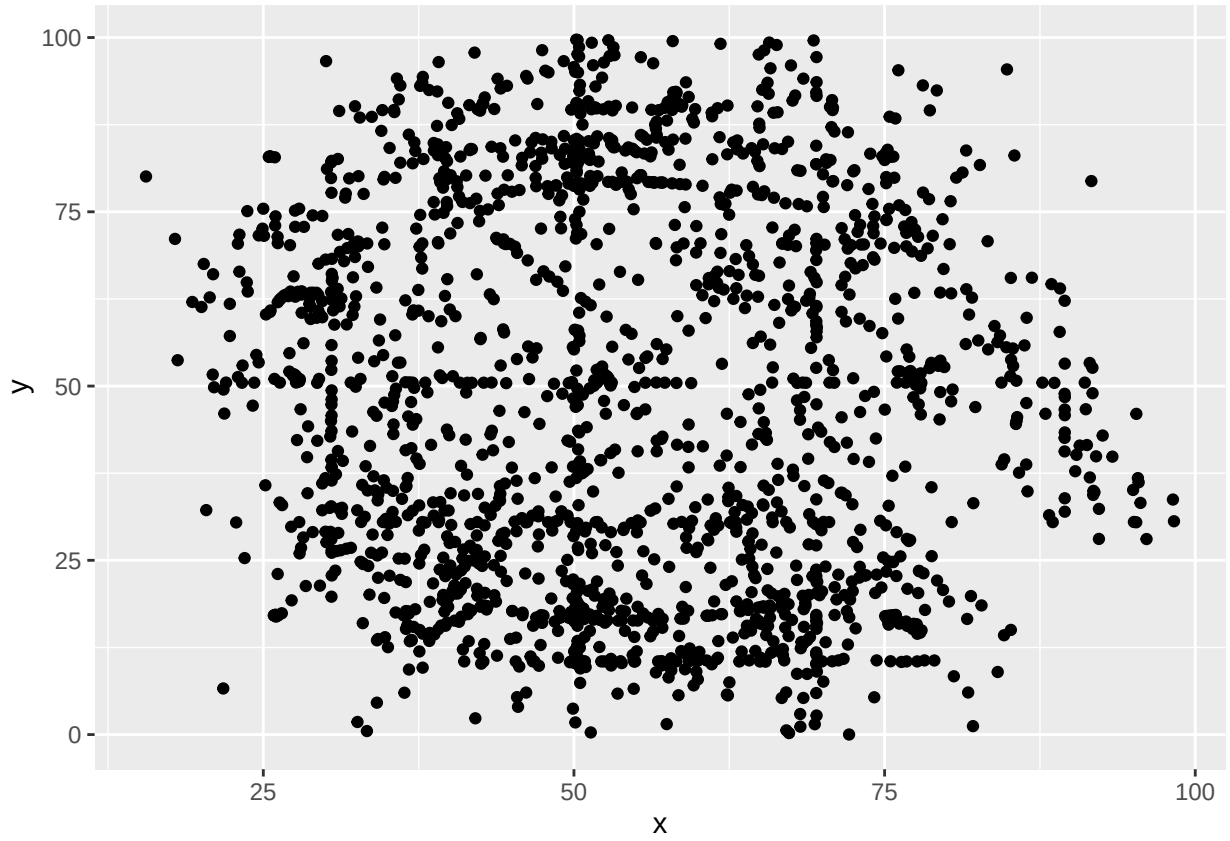
The description of the data was given as "a dataset which has multiple repeated measurements from two devices by thirteen Observers". Where the device measurements were in columns "dev1" and "dev2". Reimport that dataset, change the names of "dev1" and "dev2" to x and y and do the following:

```
#A dataset which has multiple repeated measurements from two devices
#(dev1 and dev2) by thirteen Observers.
observations <- readRDS("~/Downloads/Stat Progr Packages/STAT5014_tdolkar/HW3_data.rds")
observations <- observations %>%
                  arrange(Observer)
colnames(observations) <- c("observers", "x", "y")
observations$observers <- as.factor(observations$observers)
```

```
s_plot <- function(data,x,y,obs=NULL){
  par(mfrow = c(1,1))
  if(is.null(obs)){
    data %>%
      ggplot(aes(x=x,y=y)) + geom_point()
  }
  else{
    data %>%
      ggplot(aes(x=x,y=y)) + geom_point() + facet_wrap(observers~.)
  }
}
```

a. a single scatter plot of the entire dataset:

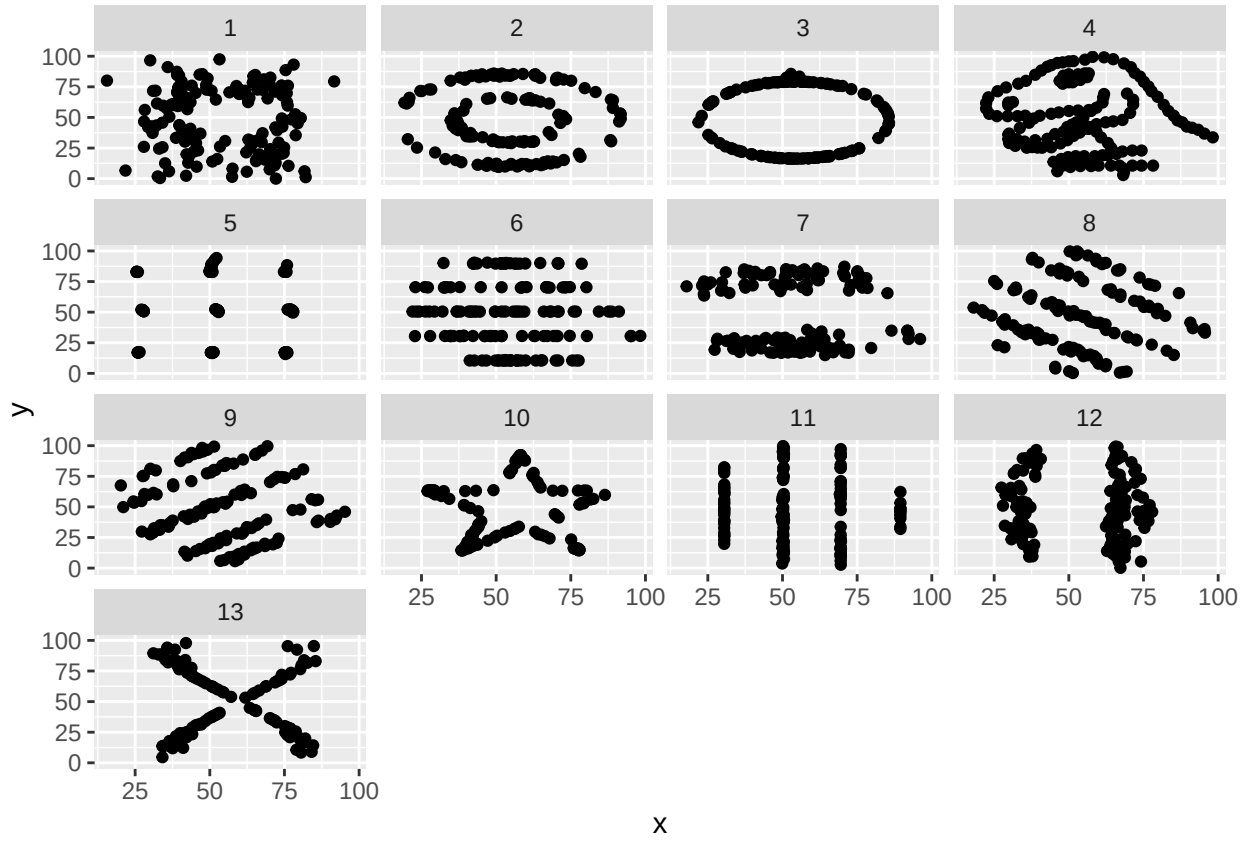
```
s_plot(observations,observations$x,observations$y)
```



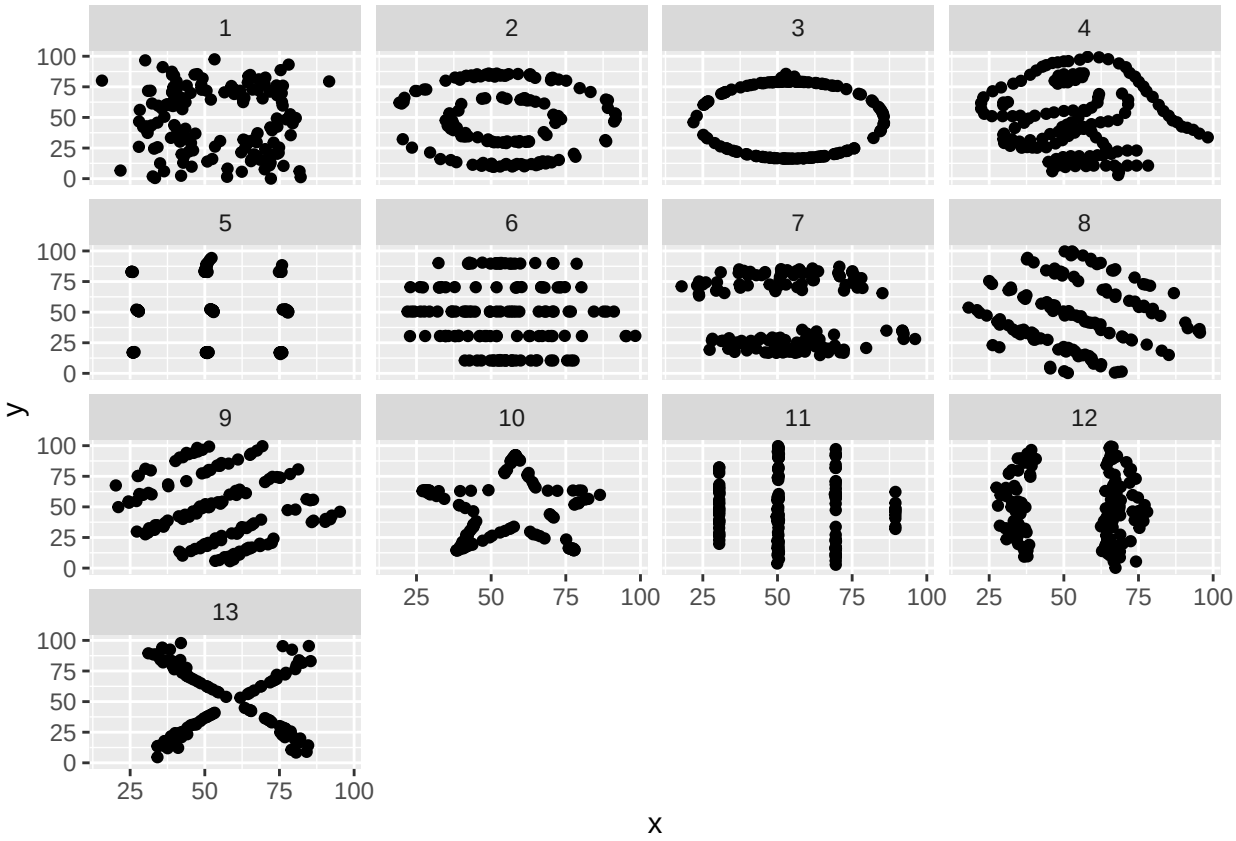
b. separate scatter plot for each observer:

```
lapply(names(observations[,2:3]),function(x)s_plot(observations,observations$x,observations$y,observati
```

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




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



Problem 7

Our ultimate goal in this problem is to create an annotated map of the US. I am giving you the code to create said map, you will need to customize it to include the annotations.

a. Get and import a database of US cities and states. Here is some R code to help:

```
#we are grabbing a SQL set from here
# http://www.farinspace.com/wp-content/uploads/us_cities_and_states.zip
#download the files, looks like it is a .zip
download("http://www.farinspace.com/wp-content/uploads/us_cities_and_states.zip",dest="us_cities_states
unzip("us_cities_states.zip", exdir="./")

#read in data, looks like sql dump, blah
states <- fread(input = "./us_cities_and_states/states.sql",skip = 23,sep = "'", sep2 = ",", header = F
### YOU do the CITIES
### I suggest the cities_extended.sql may have everything you need
cities <- fread(input = "./us_cities_and_states/cities_extended.sql",skip = 26,sep = "'", sep2 = ",", he
### can you figure out how to limit this to the 50?
# for(i in 1:nrow(states)){
#   US_cities.2 <- cities %>%
#                   filter(V4 == states$V4[i])
# }
```

```
US_cities <- data.frame()

for(i in 1:nrow(states)){
  for(j in 1:nrow(cities)){
    if(cities$V4[j] == states$V4[i]){
      US_cities <- rbind(US_cities, cities[j])
    }
  }
}
```

b. Create a summary table of the number of cities included by state.

```
#count the number of cities in each state
freq <- US_cities %>%
                count(V4)
colnames(states) <- c('state', 'state_code')

#add a column with count of cities for each state
states <- states %>% add_column(city_in_each_state_freq = freq$n)

#make all state in lower case writing for easier comparison
states <- states %>% mutate(state = tolower(state))

#take out DC and add it last since it is not really a state.
capital_city <- states %>% filter(state == "district of columbia")
states <- subset(states, state != "district of columbia")
states <- rbind(states, capital_city)
```

c. Create a function that counts the number of occurences of a letter in a string. The input to the function should be "letter" and "state_name". The output should be a scalar with the count for that letter.

Create a for loop to loop through the state names imported in part a. Inside the for loop, use an apply family function to iterate across a vector of letters and collect the occurrence count as a vector.

```
##pseudo code
letter_count <- data.frame(matrix(NA,nrow=50, ncol=26))
getCount <- function(state_name){
  count = data.frame(matrix(0,nrow=1, ncol=26))
  letter_in_state <- unlist(strsplit(state_name,""))
  for(i in letter_in_state){
    if(i == 'A'||i == 'a'){
      count[1,1] = sum(str_count(state_name, c("a",'A')))
    }
    else if(i == 'B'||i == 'b'){
      count[1,2] = sum(str_count(state_name, c("b",'B')))
    }
    else if(i == 'C'||i == 'c'){
      count[1,3] = sum(str_count(state_name, c("c",'C')))
    }
    else if(i == 'D'||i == 'd'){
```

```
    count[1,4] = sum(str_count(state_name, c("d",'D')))
  }
  else if(i == 'E'||i == 'e'){
    count[1,5] = sum(str_count(state_name, c("e",'E')))
  }
  else if(i == 'F'||i == 'f'){
    count[1,6] = sum(str_count(state_name, c("f",'F')))
  }
  else if(i == 'G'||i == 'g'){
    count[1,7] = sum(str_count(state_name, c("g",'G')))
  }
  else if(i == 'H'||i == 'h'){
    count[1,8] = sum(str_count(state_name, c("h",'H')))
  }
  else if(i == 'I'||i == 'i'){
    count[1,9] = sum(str_count(state_name, c("i",'I')))
  }
  else if(i == 'J'||i == 'j'){
    count[1,10] = sum(str_count(state_name, c("j",'J')))
  }
  else if(i == 'K'||i == 'k'){
    count[1,11] = sum(str_count(state_name, c("k",'K')))
  }
  else if(i == 'L'||i == 'l'){
    count[1,12] = sum(str_count(state_name, c("l",'L')))
  }
  else if(i == 'M'||i == 'm'){
    count[1,13] = sum(str_count(state_name, c("m",'M')))
  }
  else if(i == 'N'||i == 'n'){
    count[1,14] = sum(str_count(state_name, c("n",'N')))
  }
  else if(i == 'O'||i == 'o'){
    count[1,15] = sum(str_count(state_name, c("o",'O')))
  }
  else if(i == 'P'||i == 'p'){
    count[1,16] = sum(str_count(state_name, c("p",'P')))
  }
  else if(i == 'Q'||i == 'q'){
    count[1,17] = sum(str_count(state_name, c("q",'Q')))
  }
  else if(i == 'R'||i == 'r'){
    count[1,18] = sum(str_count(state_name, c("r",'R')))
  }
  else if(i == 'S'||i == 's'){
    count[1,19] = sum(str_count(state_name, c("s",'S')))
  }
  else if(i == 'T'||i == 't'){
    count[1,20] = sum(str_count(state_name, c("t",'T')))
  }
  else if(i == 'U'||i == 'u'){
    count[1,21] = sum(str_count(state_name, c("u",'U')))
  }
```

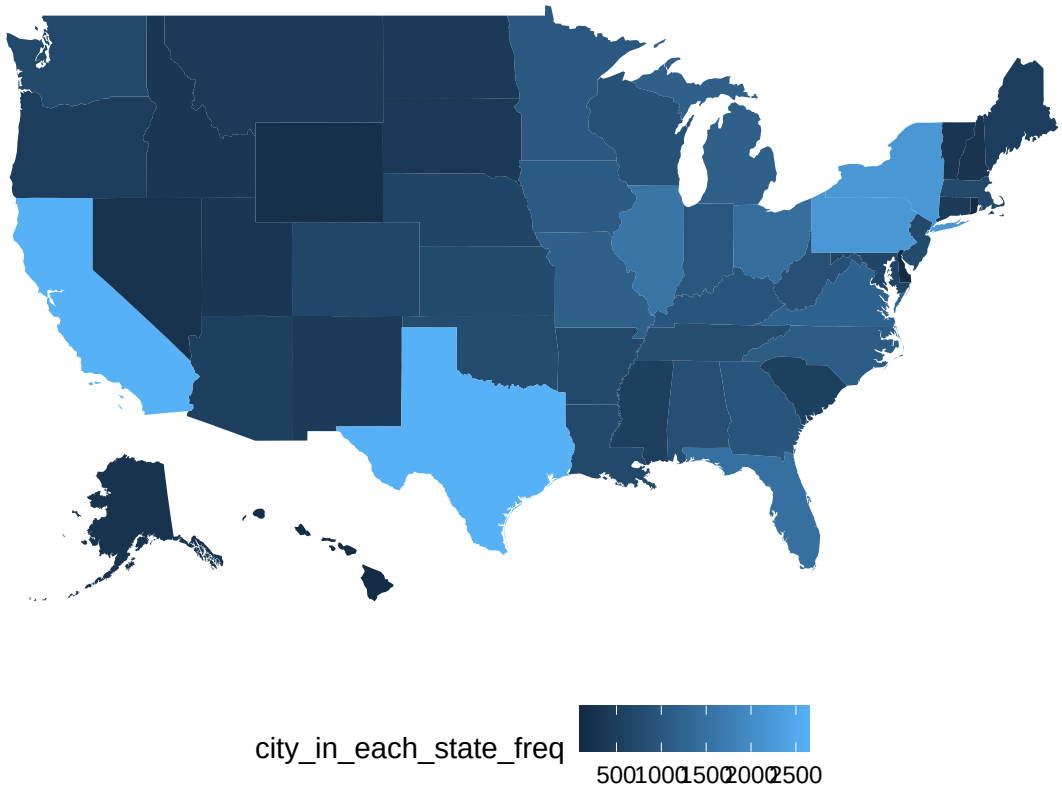
```
    else if(i == 'V'||i == 'v'){
      count[1,22] = sum(str_count(state_name, c("v",'V')))
    }
    else if(i == 'W'||i == 'w'){
      count[1,23] = sum(str_count(state_name, c("w",'W')))
    }
    else if(i == 'X'||i == 'x'){
      count[1,24] = sum(str_count(state_name, c("x",'X')))
    }
    else if(i == 'Y'||i == 'y'){
      count[1,25] = sum(str_count(state_name, c("y",'Y')))
    }
    else if(i == 'Z'||i == 'z'){
      count[1,26] = sum(str_count(state_name, c("z",'Z')))
    }
  }
  return(count)
}
for(i in 1:51){
  letter_count[i,] <- sapply(states$state[i],getCount)
}
colnames(letter_count) <- c('A',"B","C","D","E","F","G","H","I","J","K","L","M","N","O","P","Q","R","S"
```

d.

Map 1: should be colored by count of cities on our list within the state.

Quick and not so dirty map:

```
#https://cran.r-project.org/web/packages/fiftystater/vignettes/fiftystater.html

library(fiftystater)

data("fifty_states") # this line is optional due to lazy data loading
#crimes <- data.frame(state = tolower(rownames(USArrests)), USArrests)
# map_id creates the aesthetic mapping to the state name column in your data
p <- ggplot(states, aes(map_id = state)) +
  # map points to the fifty_states shape data
  geom_map(aes(fill = city_in_each_state_freq), 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
```



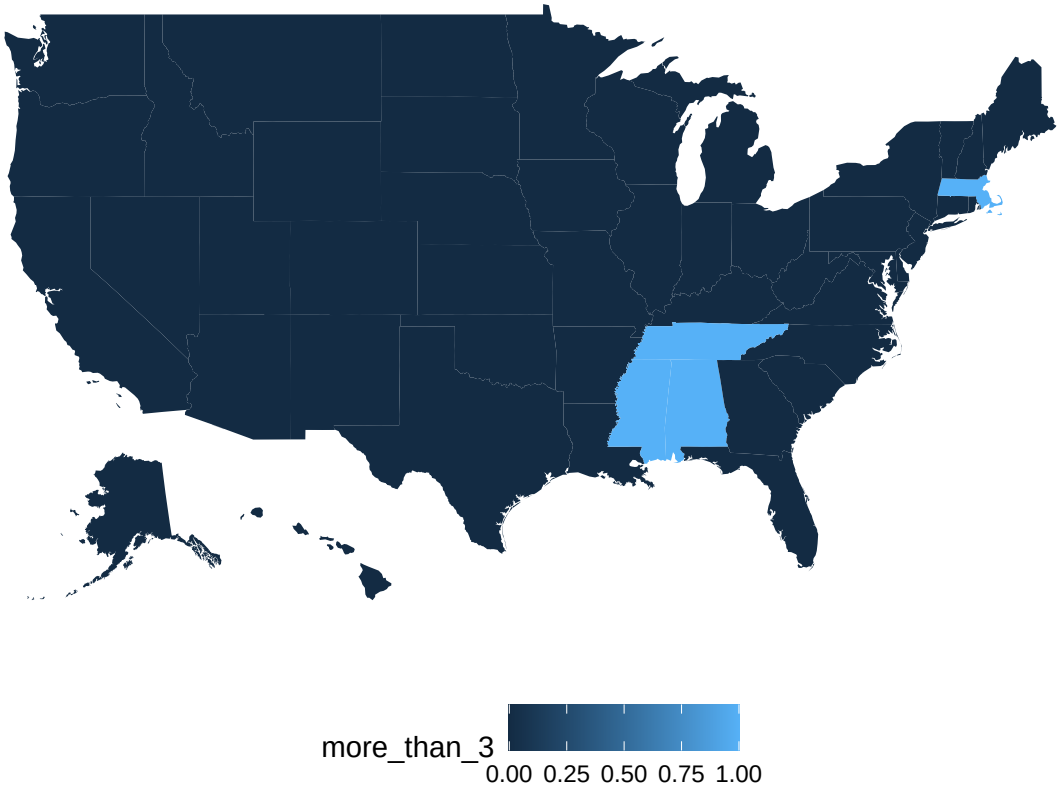

```
#ggsave(plot = p, file = "HW6_Problem6_Plot_Settlage.pdf")
```

As the state goes from darker shade of blue to lighter, the count of cities in each state increases.

Map 2: should highlight only those states that have more than 3 occurrences of ANY letter in their name.

```
more_than_3 <- apply(letter_count, 1, function(x) if(any(x > 3)) { return(1)} else {return(0)})

states <- cbind(states, more_than_3)

#https://cran.r-project.org/web/packages/fiftystater/vignettes/fiftystater.html

library(fiftystater)

data("fifty_states") # this line is optional due to lazy data loading
crimes <- data.frame(state = tolower(rownames(USArrests)), USArrests)
# map_id creates the aesthetic mapping to the state name column in your data
p <- ggplot(states, aes(map_id = state)) +
  # map points to the fifty_states shape data
  geom_map(aes(fill = more_than_3), 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
```




```
#ggsave(plot = p, file = "HW6_Problem6_Plot_Settlage.pdf")
```

The light blue states have more than 3 occurrences of any letter in their name.

Problem 8

Bootstrapping

Sometimes, you really want more data to do the desired analysis, but going back to the "field" is often not an option. An often used method is bootstrapping. Check out the second answer here for a really nice and detailed description of bootstrapping: https://stats.stackexchange.com/questions/316483/manually-bootstrapping-linear-regression-in-r.

What we want to do is bootstrap the Sensory data to get non-parametric estimates of the parameters. Assume that we can neglect item in the analysis such that we are really only interested in a linear model lm(y~operator).

```
## getting http://www2.isye.gatech.edu/~jeffwu/wuhamadabook/data/Sensory.dat

url <- "http://www2.isye.gatech.edu/~jeffwu/wuhamadabook/data/Sensory.dat"
```

```
sensorydata_raw<- fread(url, data.table = FALSE, fill = TRUE, skip = 2, header = FALSE)
saveRDS(sensorydata_raw, "sensorydata_raw.RDS")
#Saves the object in it's native format without the name: When importing, it is easier for us.
sensorydata_raw <- readRDS("sensorydata_raw.RDS")
```

```
#if the first condition is T, return the index value
Na <- which(is.na(sensorydata_raw$V6),arr.ind=T)
df <- cbind(rep(1:10, each = 2), sensorydata_raw[Na,])
df$V6 <- NULL
#create a clone data frame of the raw data, to get a better aligned raw data
# and not change the original raw data.
# new_sensorydata_raw <- sensorydata_raw
sensorydata_raw[Na,] <- df
colnames(sensorydata_raw) <- c("Item","1","2","3","4","5")
Operator <- stack(sensorydata_raw[,2:6])
sensorydata_tidy_br <- data.frame(Item=rep(sensorydata_raw$Item, 5),
                                  as.character(Operator[,2]), as.numeric(Operator[,1]))
colnames(sensorydata_tidy_br) <- c('Item', "Operator", "sensorydata")
sensorydata_tidy_br <- sensorydata_tidy_br[sort(sensorydata_tidy_br$Operator,decreasing=F,
                                                index.return=T)[[2]],]
```

Part a. First, the question asked in the stackexchange was why is the supplied code not working. This question was actually never answered. What is the problem with the code?If you want to duplicate the code to test it, use the quantreg package to get the data.

```
#1)fetch data from Yahoo
#AAPL prices
apple08 <- getSymbols('AAPL', auto.assign = FALSE, from = '2008-1-1', to =
"2008-12-31")[,6]
```

```
## 'getSymbols' currently uses auto.assign=TRUE by default, but will
## use auto.assign=FALSE in 0.5-0. You will still be able to use
## 'loadSymbols' to automatically load data. getOption("getSymbols.env")
## and getOption("getSymbols.auto.assign") will still be checked for
## alternate defaults.
##
## This message is shown once per session and may be disabled by setting
## options("getSymbols.warning4.0"=FALSE). See ?getSymbols for details.
```

```
#market proxy
rm08<-getSymbols('^ixic', auto.assign = FALSE, from = '2008-1-1', to =
"2008-12-31")[,6]

#log returns of AAPL and market
logapple08<- na.omit(ROC(apple08)*100)
logrm08<-na.omit(ROC(rm08)*100)

#OLS for beta estimation
beta_AAPL_08<-summary(lm(logapple08~logrm08))$coefficients[2,1]

#create df from AAPL returns and market returns
```

```
df08<-cbind(logapple08,logrm08)

#added line to refer to variables to fit them in a linear model
colnames(df08) <- c("logapple08", "logrm08")

set.seed(666)
Boot_times=1000

#First of all variable 'Boot' is not defined in the code.
#sd.boot=rep(0,Boot)
sd.boot=data.frame(matrix(NA,nrow=2, ncol=Boot_times))

#for(i in 1:Boot){
for(i in 1:Boot_times){

# nonparametric bootstrap
bootdata=df08[sample(nrow(df08), size = 251, replace = TRUE),]

#sd.boot[i]= coef(summary(lm(logapple08~logrm08, data = bootdata)))[2,2]
#since this is just the standard error of betah1

sd.boot[,i]= coef(summary(lm(logapple08~logrm08, data = bootdata)))[,1]
#this will keep record of 1000 value of betah0, betah1 instead of std. error
}
```

The problematic parts of the original code has been commented out in between codes with explanation as to what it was doing and it follows up with what the corrections in the code will do to fix it.

Part b. Bootstrap the analysis to get the parameter estimates using 100 bootstrapped samples. Make sure to use system.time to get total time for the analysis. You should probably make sure the samples are balanced across operators, ie each sample draws for each operator.

```
library(boot)
# function to obtain parameter estimate from the data
 parameter_estimates <- function(formula, data, indices) {
  d <- data[indices,] # allows boot to select sample
  fit <- lm(formula, data=d)
  return(summary(fit)$coefficients)
}
# bootstrapping with 100 replications
naive_time <- system.time({results <- boot(data=sensorydata_tidy_br, statistic=parameter_estimates,
   R=100, formula=sensorydata~Operator)})

# view results
results
```

```
##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
## Call:
```

```
## boot(data = sensorydata_tidy_br, statistic = parameter_estimates,
##     R = 100, formula = sensorydata ~ Operator)
##
##
## Bootstrap Statistics :
##           original        bias     std. error
## t1*   4.593333e+00 -4.153346e-02 4.192676e-01
## t2*   4.700000e-01  1.872374e-02 5.972773e-01
## t3*  -4.266667e-01  2.894372e-02 5.703188e-01
## t4*   6.000000e-01  5.798524e-02 6.041086e-01
## t5*  -3.266667e-01  7.206454e-02 6.256709e-01
## t6*   3.893704e-01 -2.751938e-03 3.515148e-02
## t7*   5.506528e-01 -4.736482e-04 3.674111e-02
## t8*   5.506528e-01 -4.352006e-03 3.627690e-02
## t9*   5.506528e-01  1.580016e-04 3.551572e-02
## t10*  5.506528e-01 -1.565527e-03 4.129958e-02
## t11*  1.179682e+01  5.775886e-02 1.397885e+00
## t12*  8.535323e-01  5.003462e-02 1.103364e+00
## t13* -7.748379e-01  5.069030e-02 1.038134e+00
## t14*  1.089616e+00  1.243159e-01 1.128640e+00
## t15* -5.932352e-01  1.338019e-01 1.148781e+00
## t16*  6.134921e-23  1.046282e-13 1.044615e-12
## t17*  3.947720e-01 -1.480397e-02 3.130053e-01
## t18*  4.396972e-01 -3.637813e-02 2.925567e-01
## t19*  2.776900e-01  2.717602e-02 2.787673e-01
## t20*  5.539479e-01 -1.075682e-01 3.092917e-01
```

```
naive_time
```

```
##    user  system elapsed
##   0.100   0.001   0.104
```

Part c. Redo the last problem but run the bootstraps in parallel (`cl <- makeCluster(8)`), don't forget to `stopCluster(cl)`). Why can you do this? Make sure to use system.time to get total time for the analysis.

```
# function to obtain parameter estimate from the data
 parameter_estimates <- function(formula, data, indices) {
  d <- data[indices,] # allows boot to select sample
  fit <- lm(formula, data=d)
  return(summary(fit)$coefficients)
}
# bootstrapping with 100 replications
parallel_time <- system.time({results <- boot(data=sensorydata_tidy_br, statistic=parameter_estimates,
   R=100, formula=sensorydata~Operator, parallel = "multicore")})

# view results
results
```

```
##
## ORDINARY NONPARAMETRIC BOOTSTRAP
```

```
##
##
## Call:
## boot(data = sensorydata_tidy_br, statistic = parameter_estimates,
##     R = 100, formula = sensorydata ~ Operator, parallel = "multicore")
##
##
## Bootstrap Statistics :
##           original        bias     std. error
## t1*   4.593333e+00  2.263746e-02 4.392201e-01
## t2*   4.700000e-01 -3.608691e-02 6.353829e-01
## t3*  -4.266667e-01 -4.455368e-03 5.675266e-01
## t4*   6.000000e-01 -4.018023e-02 5.741259e-01
## t5*  -3.266667e-01 -7.989124e-02 6.244180e-01
## t6*   3.893704e-01 -1.669007e-03 3.600822e-02
## t7*   5.506528e-01  9.619547e-05 3.695039e-02
## t8*   5.506528e-01 -1.188962e-03 3.891400e-02
## t9*   5.506528e-01 -7.406663e-03 4.000254e-02
## t10*  5.506528e-01 -3.862868e-03 3.787157e-02
## t11*  1.179682e+01  1.868556e-01 1.386951e+00
## t12*  8.535323e-01 -5.254071e-02 1.170990e+00
## t13* -7.748379e-01  1.838379e-03 1.033622e+00
## t14*  1.089616e+00 -5.791285e-02 1.052022e+00
## t15* -5.932352e-01 -1.521192e-01 1.134573e+00
## t16*  6.134921e-23  7.577783e-17 5.634371e-16
## t17*  3.947720e-01 -1.240551e-03 3.108667e-01
## t18*  4.396972e-01 -3.955506e-02 2.980106e-01
## t19*  2.776900e-01  1.050021e-01 3.196386e-01
## t20*  5.539479e-01 -1.408474e-01 3.057243e-01
```

```
parallel_time
```

```
##    user  system elapsed
##   0.093   0.001   0.094
```

Create a single table summarizing the results and timing from part a and b. What are your thoughts?

Problem 9

Newton's method gives an answer for a root. To find multiple roots, you need to try different starting values. There is no guarantee for what start will give a specific root, so you simply need to try multiple. From the plot of the function in HW4, problem 8, how many roots are there?

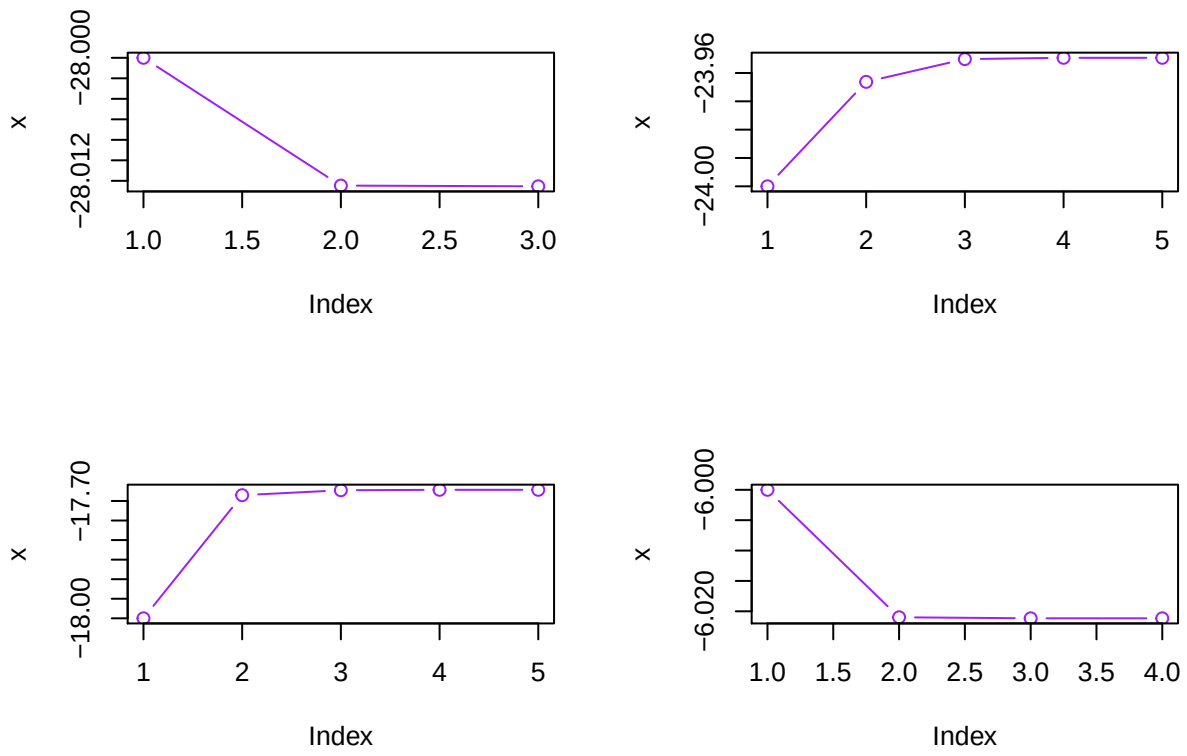
Create a vector (`length.out=1000`) as a "grid" covering all the roots and extending +/-1 to either end.

```
N_method <- function(x1 = -5){
  #function takes in a first approximation of the root x0 of the function 3^x-sin(x)+cos(5*x)
  #function uses Newton's method of approximate the root of the function
  #function terminates when successive estimates are within 1e-04 of each other
  seq <- 100
  x <- double(0)
  x[1] <-  x1
  # f <- 3^x- sin(x) + cos(5*x)
```

```
  # f_diff <- 3^x * log(3) - cos(x) - sin(5 * x) * 5
  for(i in 2:seq){
    if(3^x[i-1] * log(3) - cos(x[i-1]) - sin(5 * x[i-1]) * 5 == 0){
      stop("choose a new starting place")
    }
    x[i] <- x[i-1] - ((3^x[i-1]- sin(x[i-1]) + cos(5*x[i-1]))/
                        (3^x[i-1] * log(3) - cos(x[i-1]) - sin(5 * x[i-1]) * 5))
    if (abs(x[i]-x[i-1])<1e-04){
      break
    }
  }
  #For the initial estimate of:
  print(x[1])
  #Xn approaches 0 at:
  print(tail(x,1))
  return(plot(x, type = 'b', col = 'purple'))
}
```

```
# we have:
#f <- expression(3^x- sin(x) + cos(5*x))
init_x <- c(-28, -24, -18, -6)
par(mfrow = c(2,2))
for(i in 1:length(init_x)){
  p = N_method(init_x[i])
}
```

```
## [1] -28
## [1] -28.01253

## [1] -24
## [1] -23.95464

## [1] -18
## [1] -17.67146

## [1] -6
## [1] -6.021155
```




```
p
```

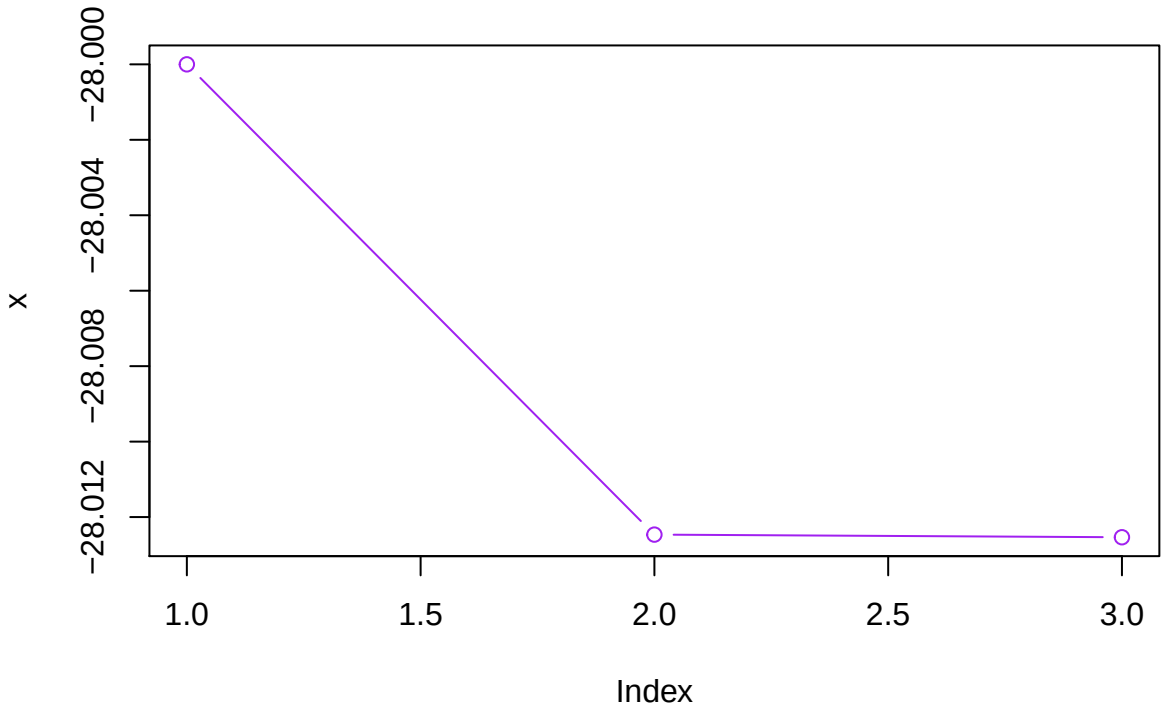
```
## NULL
```

This is the result from HW4, problem 8.

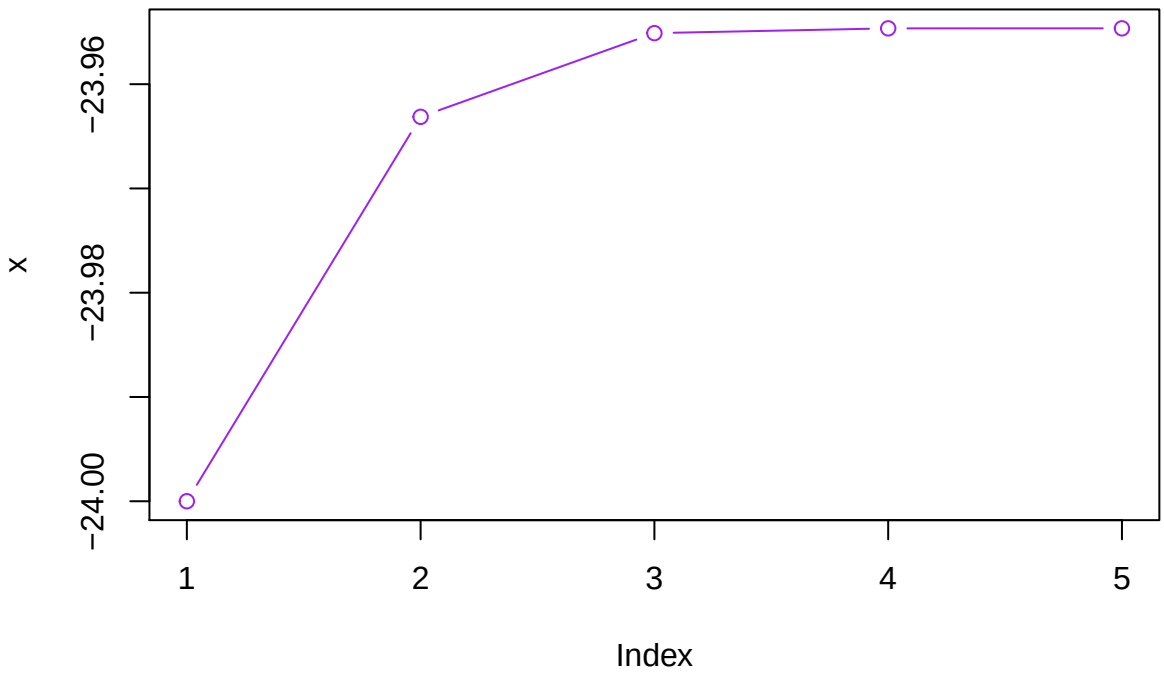
Part a. Using one of the apply functions, find the roots noting the time it takes to run the apply function.

```
N_time <- system.time(lapply(init_x,N_method))
```

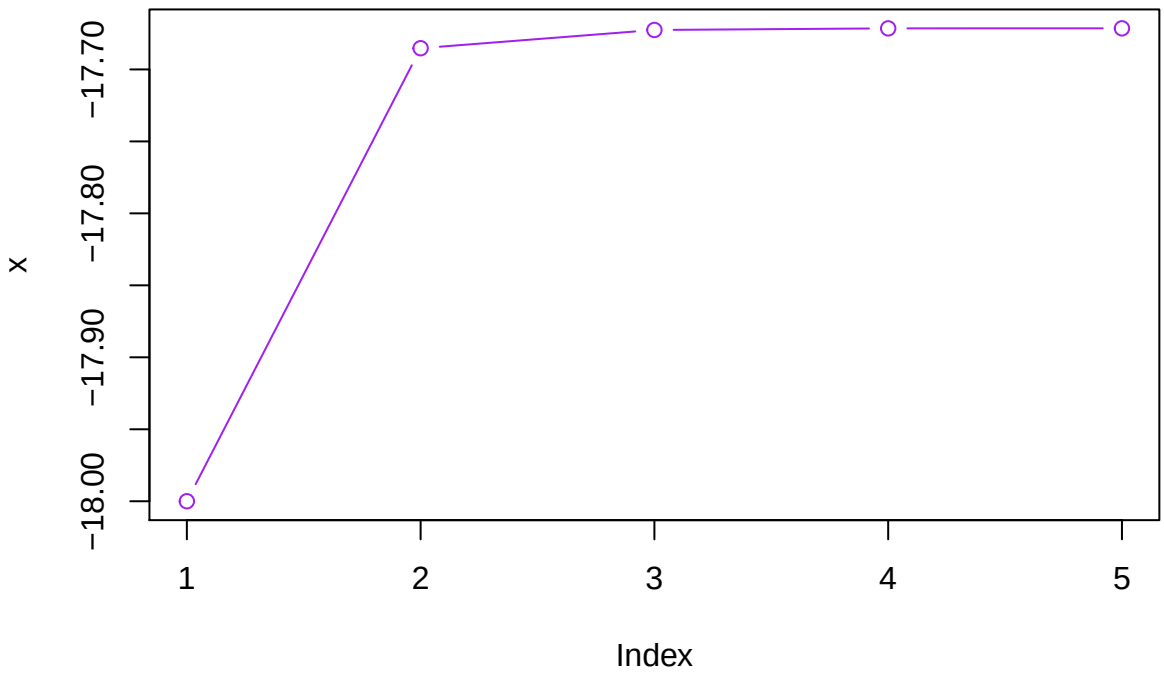
```
## [1] -28
## [1] -28.01253
```



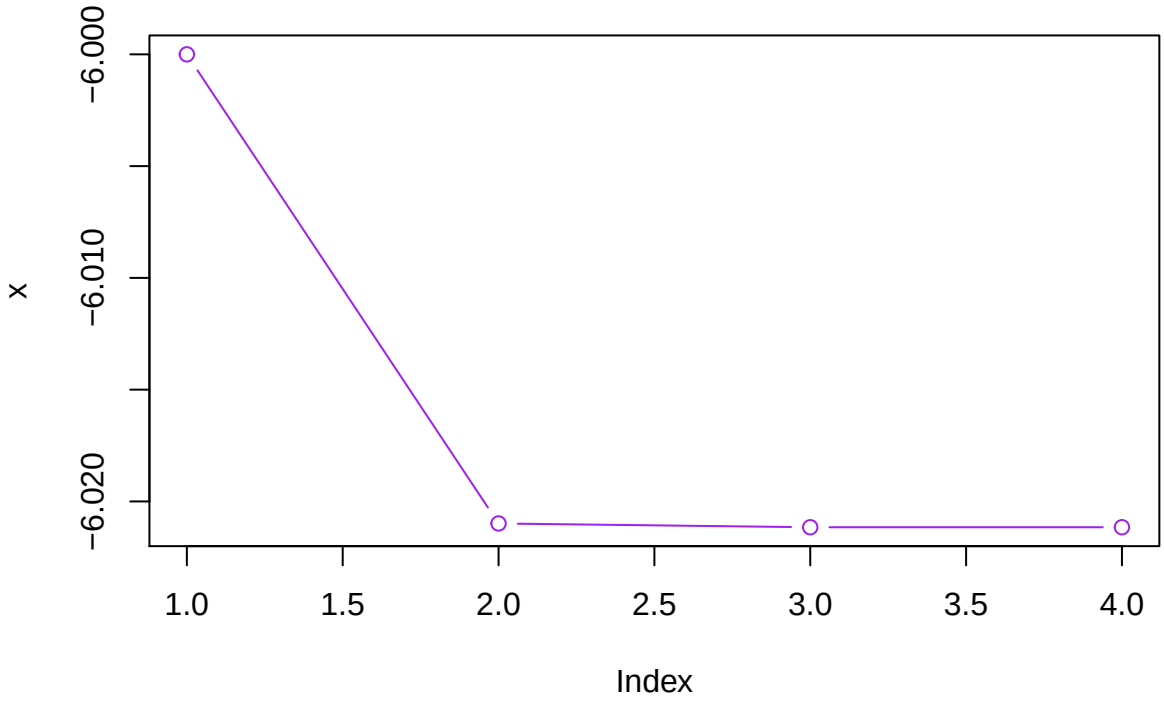

```
## [1] -24
## [1] -23.95464
```




```
## [1] -18
## [1] -17.67146
```




```
## [1] -6
## [1] -6.021155
```




```
N_time
```

```
##    user  system elapsed
##   0.003   0.000   0.003
```

I tried it with a vector(length.out = 1000) as a "grid" but to save compile time, I reproduced the same result with apply function.

Part b. Repeat the apply command using the equivelant parApply command. Use 8 workers. `cl <- makeCluster(8)`.

```
init_x <- c(-28, -24, -18, -6)

system.time({
  # Create a cluster via makeCluster
  cl <- makeCluster(8)
  # Parallelize
  parLapply(cl, init_x, N_method)})
```

```
##    user  system elapsed
##   0.010   0.005   0.481
```

```
# Stop the cluster
stopCluster(cl)
```

Create a table summarizing the roots and timing from both parts a and b. What are your thoughts?

Problem 10

Finish this homework by pushing your changes to your repo.

Only submit the .Rmd and .pdf solution files. Names should be formatted HW4_pid.Rmd and HW4_pid.pdf