HW4_Xinyis

2020-10-14

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 2: 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 <- as.matrix(c(1,2),nrow=2)
X <- cbind(1,rep(1:10,10))
h <- as.vector(X%*%theta+rnorm(100,0,0.2))
m <- dim(X)[1]
#theta = matrix(0,2,1)
THETA = matrix(5,2,1)
alpha = 0.01
while((abs(theta[1] - THETA[1])>1e-06) || (abs(theta[2] - THETA[2])>1e-06)){
  THETA = theta
  h_0 = X%*%THETA
  h_y = sweep(as.matrix(h_0), 1, h, '-')
  theta[1] = THETA[1] - alpha*mean(h_y)
```

```
  h_yx = sweep(h_y, 1, as.matrix(X[,2]), '*')
  theta[2] = THETA[2] - alpha*mean(h_yx)
}
print(theta)
```

```
##           [,1]
## [1,] 0.9700454
## [2,] 2.0014948
```

```
# regression
dat = as.data.frame(cbind(h,X))
fit = lm(h~0+X, data = dat)
summary(fit)
```

```
##
## Call:
## lm(formula = h ~ 0 + X, data = dat)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -0.5724 -0.1342 -0.0164  0.1179  0.4807
##
## Coefficients:
##    Estimate Std. Error t value Pr(>|t|)
## X1 0.969571   0.042344    22.9   <2e-16 ***
## X2 2.001563   0.006824   293.3   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.196 on 98 degrees of freedom
## Multiple R-squared:  0.9998, Adjusted R-squared:  0.9998
## F-statistic: 2.297e+05 on 2 and 98 DF,  p-value: < 2.2e-16
```

Here, I use alpha - the step size as 0.001 and tolerance value as 0.001, the results are very closed.

Problem 3

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.

Part 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 2, 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.

```
set.seed(1256)
X <- cbind(1,rep(1:10,10))
theta <- as.matrix(c(1,2),nrow=2)
h <- as.vector(X%*%theta+rnorm(100,0,0.2))
THETAs = expand.grid(seq(0, 2, length.out = 100), seq(1, 3, length.out = 100))
```

```
# function of Gradient Descent
# Modify the function to do parallel programming
grad = function(theta_start, X, h){
  theta_old_0 = 1000
  theta_old_1 = 1000
  alpha = 1e-07
  tol = 1e-06
  theta_new_0 = theta_start[1]
  theta_new_1 = theta_start[2]
  i = 0 # i: iteration times
  while((abs(theta_new_0-theta_old_0)>tol) || (abs(theta_new_1 - theta_old_1)>tol)){
    theta_old_0 = theta_new_0
    theta_old_1 =  theta_new_1
    theta_h = rbind(theta_old_0, theta_old_1)
    h_0 = X%*%theta_h
    h_y = sweep(as.matrix(h_0), 1, h, '-')
    theta_new_0 = theta_old_0 - alpha*sum(h_y)/length(h)
    h_yx = sweep(h_y, 1, as.matrix(X[,2]), '*')
    theta_new_1 = theta_old_1 - alpha*sum(h_yx)/length(h)
    i = i + 1
    if(i>20000) break
  }
  result = c(i, theta_new_0, theta_new_1)
  return(result)
}
```

```
library(parallel)
# A good number of clusters is the numbers of available cores minus 1.
no_cores <- detectCores() - 1
cl <- parallel::makeCluster(no_cores, setup_strategy = "sequential") # work
clusterExport(cl, 'X')
clusterExport(cl, 'h')
start_time <- Sys.time()
a <- parApply(cl,THETAs, 1, grad, X, h)
stopCluster(cl)
end_time <- Sys.time()
end_time - start_time
```

```
## Time difference of 2.035696 hours
```

```
min_iteration = min(a[1,])
max_iteration = max(a[1,])
mean_theta_0 = mean(a[2,])
sd_theta_0 = sqrt(var(a[2,]))
mean_theta_1 = mean(a[3,])
sd_theta_1 = sqrt(var(a[3,]))
iteration_summary = cbind(min_iteration, max_iteration)
print(iteration_summary)
```

```
##      min_iteration max_iteration
## [1,]             1         20001
```

```
theta_summary = cbind(mean_theta_0, mean_theta_1, sd_theta_0, sd_theta_1)
print(theta_summary)
```

```
##      mean_theta_0 mean_theta_1 sd_theta_0 sd_theta_1
```

```
## [1,]     0.999961     1.999794  0.5821502  0.5408872
```

Here, I slightly change the maximum iteration times as 20000, since if max(i) = 5000000, it will cost too much time. Also, I tried the tolerance value with 1e-09, it is too strict, and it will not converge with max(i) = 20000 although the mean value of θ_0 and θ_1 is almost same as output of R function lm(y~x). I just change the parameter setting for higher efficiency and also high accuracy.

Part 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?

I do not think it is a good way. Although it could possibly reduce the running time, however, it could give us the value that is closed to the true value, but there also exists potential problem that it might not converge, just around the local value.

Part c. What are your thoughts on this algorithm?

I think considering both efficiency and accuracy, also, let my algorithms work, we should carefully choose starting values. Also, for tolerance value 1e-09 and maximum value of iteration times with 5e+06, it is too strict, for some initial value which look like not converge, it costs too much time.

Problem 4: Inverting matrices

Ok, so John Cook makes some good points, but if you want to do:

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

what are you to do?? Can you explain what is going on?

```
# Plug into the value directly
beta = solve(t(X)%*%X)%*%t(X)%*%h
print(beta)
```

```
##           [,1]
## [1,] 0.9695707
## [2,] 2.0015630
```

Here they use Ordinary Least Squares (OLS) to estimate the β, the objective function is to minimize

$$(Y - X\beta)^t(Y - X\beta)$$

By letting the derivative of this function with respect to β be zero to get the objective β value:

$$-2X^t(Y - X\beta) = 0$$
$$\hat{\beta} = (X^tX)^{-1}X^tY$$

Problem 5: 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:

Part a.

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

```
    start_time <- Sys.time()
    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
    print(object.size(A))
```

```
## 112347224 bytes
```

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

```
## 1816357208 bytes
```

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

```
##     user   system  elapsed
## 1019.865   12.727 1101.460
```

Part b.

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

I would take the calculate the inverse of matrix B in a different way.

Part 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".

Problem 3

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

```
prop_success = function(vec){
  return(mean(vec))
}
```

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. What do you observe? What is going on?

```
apply(P4b_data, 2, FUN = function(x) prop_success(x) )
```

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

```
apply(P4b_data, 1, FUN = function(x) prop_success(x) )
```

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

```
# Generate N independent samples from a binomial (n,p) distribution
#rbinom(N,n,p)
```

d. You are to fix the above matrix by creating a function whose input is a probability and output is a vector whose elements are the outcomes of 10 flips of a coin. Now create a vector of the desired probabilities. Using the appropriate apply family function, create the matrix we really wanted above. Prove this has worked by using the function created in part a to compute and tabulate the appropriate marginal successes.

```
set.seed(12345)
matrix_generate = function(probability){
  out = rbinom(10, 1, prob = probability)
  return(out)
}
prob_input = matrix((31:40)/100,10,1)
h = sapply(prob_input, function(x) matrix_generate(x))
apply(h,1,FUN = function(x) prop_success(x))
```

```
##  [1] 0.8 0.3 0.5 0.6 0.3 0.0 0.7 0.3 0.2 0.3
```

```
apply(h,2,FUN = function(x) prop_success(x))
```

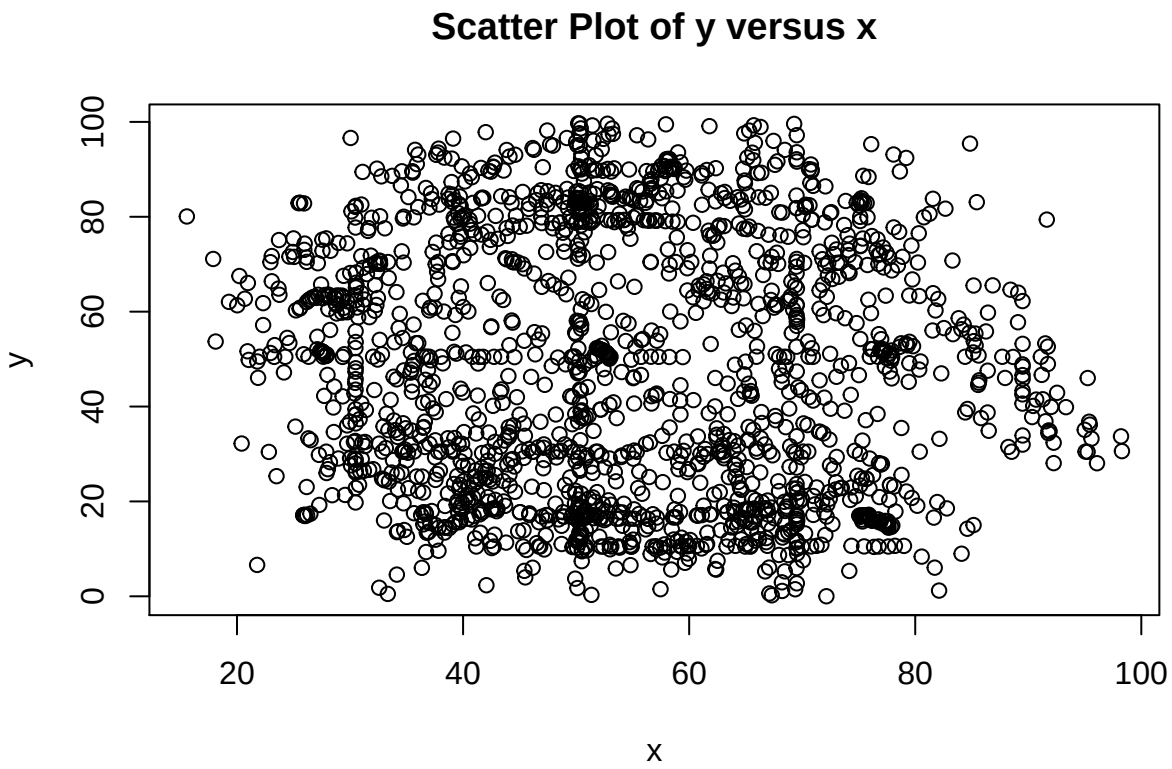
```
##  [1] 0.6 0.2 0.3 0.4 0.3 0.4 0.6 0.3 0.3 0.6
```

Problem 4

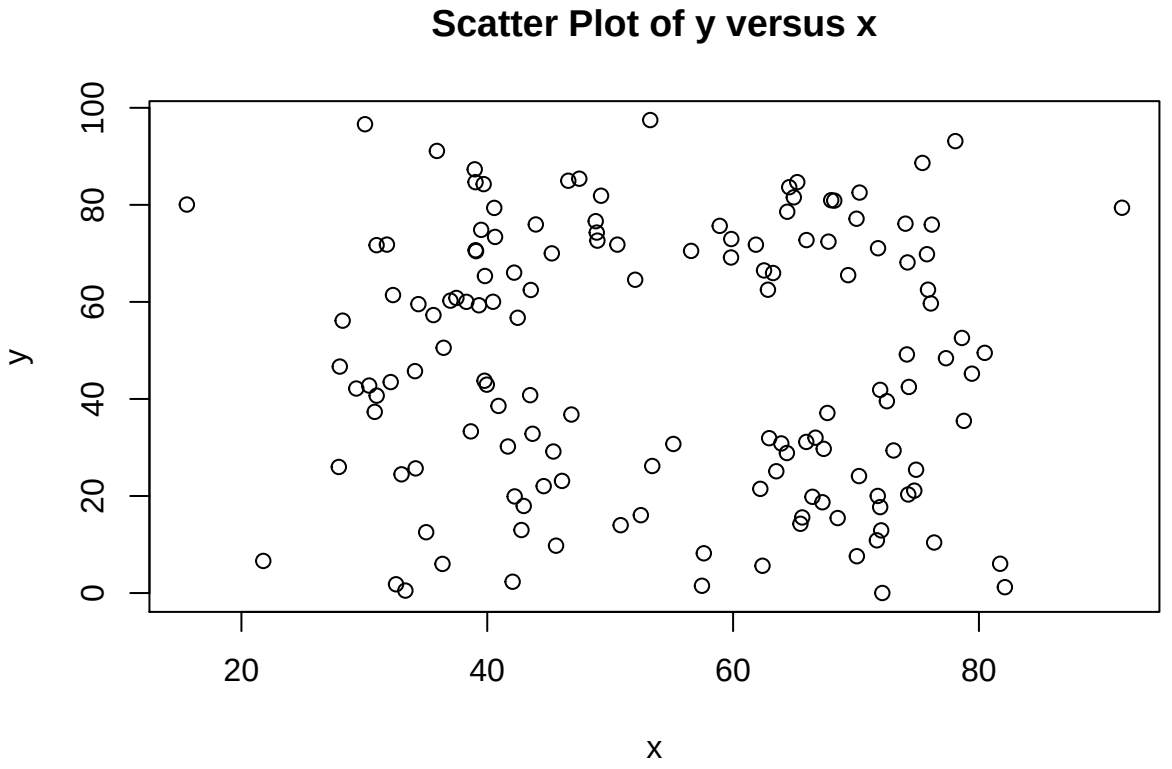
In Homework 4, we had a dataset we were to compute some summary statistics from. The description of the data was given as "a dataset which has multiple repeated measurements from two devices by thirteen Observers". Where the device measurements were in columns "dev1" and "dev2". Reimport that dataset, change the names of "dev1" and "dev2" to x and y and do the following:

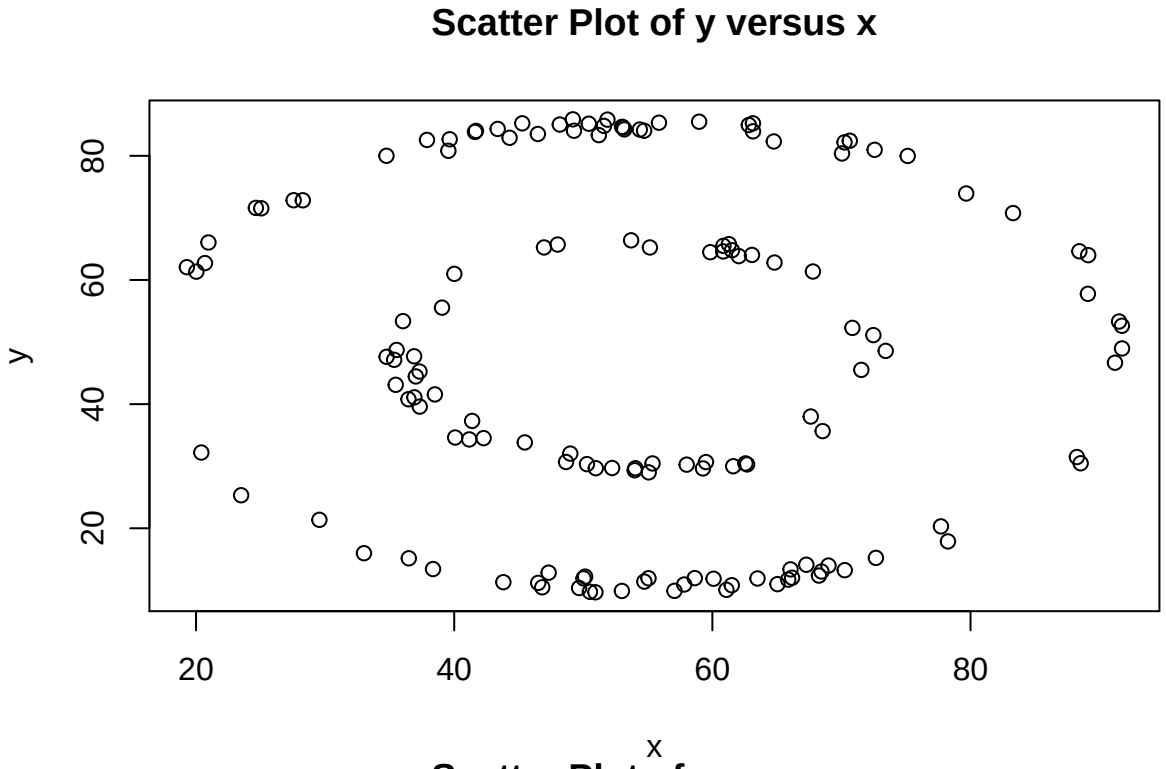
1. create a function that accepts a dataframe of values, title, and x/y labels and creates a scatter plot
2. use this function to create:
   (a) a single scatter plot of the entire dataset
   (b) a seperate scatter plot for each observer (using the apply function)

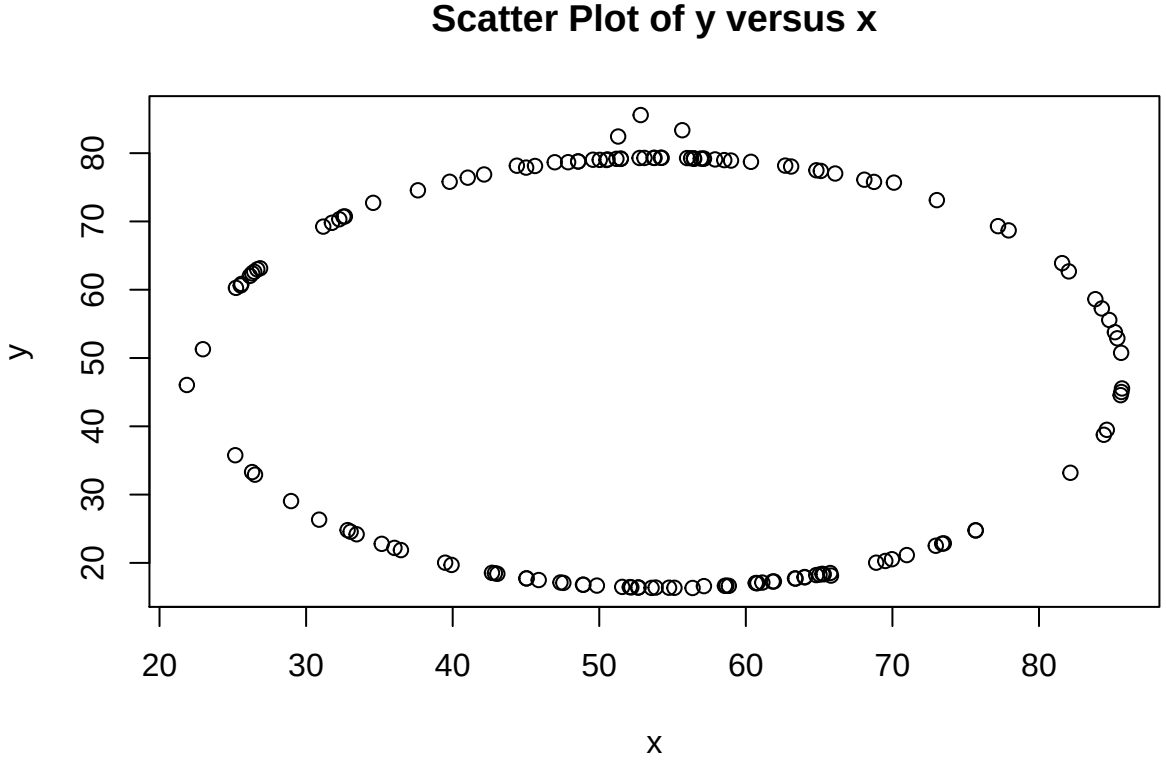
```
setwd("~/Desktop/VT Course/STAT 5014")
dat <- readRDS("HW3_data.rds")
dat_p4 = as.data.frame(dat)
colnames(dat_p4) = c('Observer', 'x','y')
scat_creation = function(values, title, label){
  plot(values$y~values$x, xlab = label[1], ylab = label[2], main = title)
}
label_input = c('x', 'y')
scat_creation(dat_p4, title = 'Scatter Plot of y versus x', label_input)
```

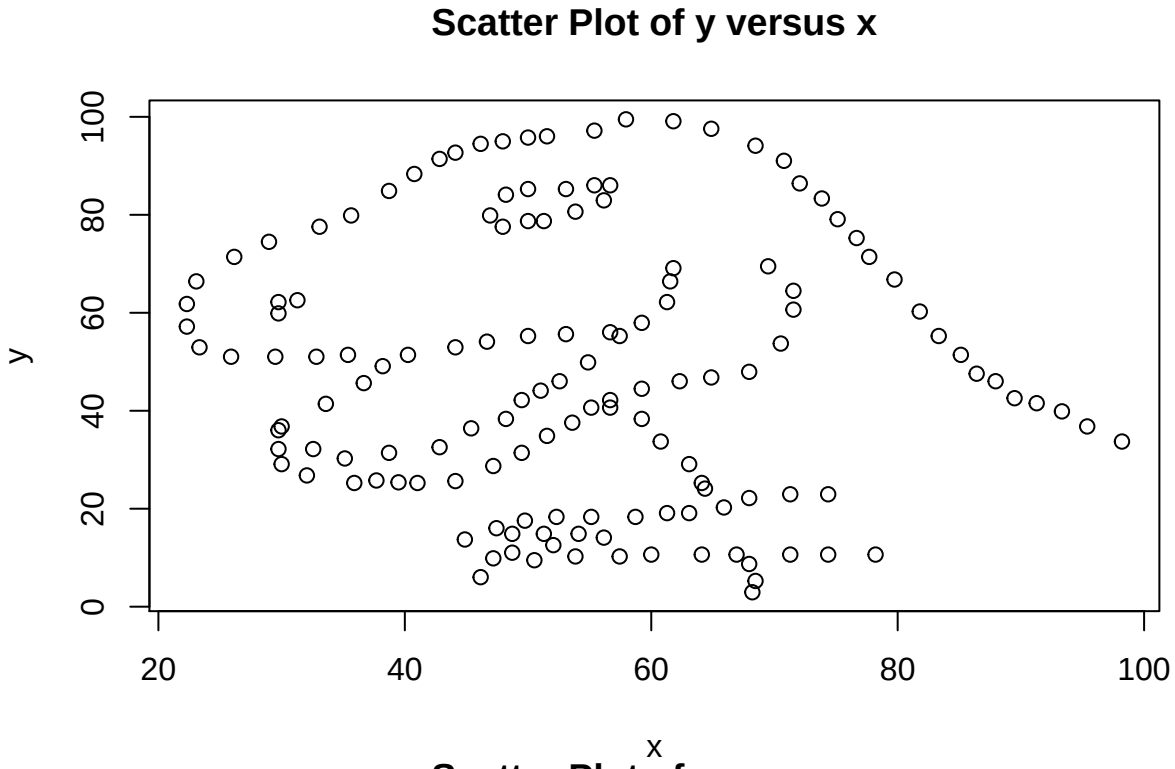


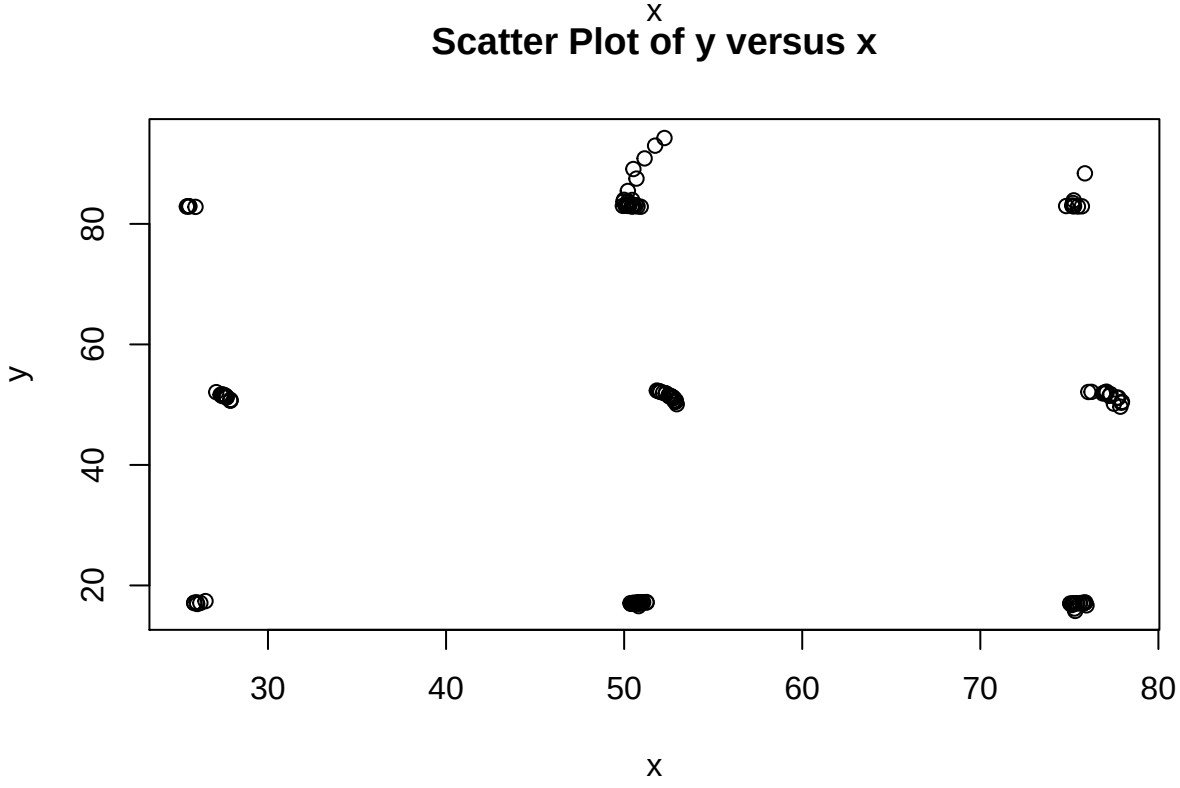

```
t = split(dat_p4, dat$Observer)
lapply(t, scat_creation,'Scatter Plot of y versus x',c('x', 'y') )
```

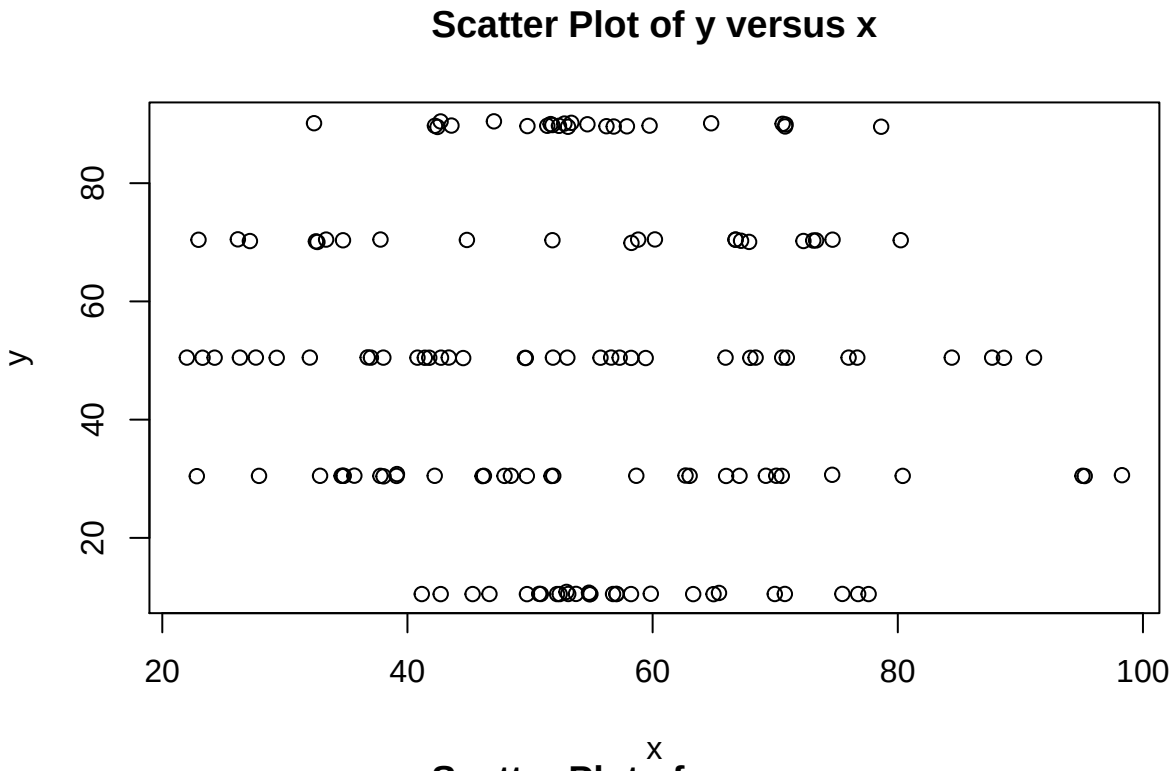


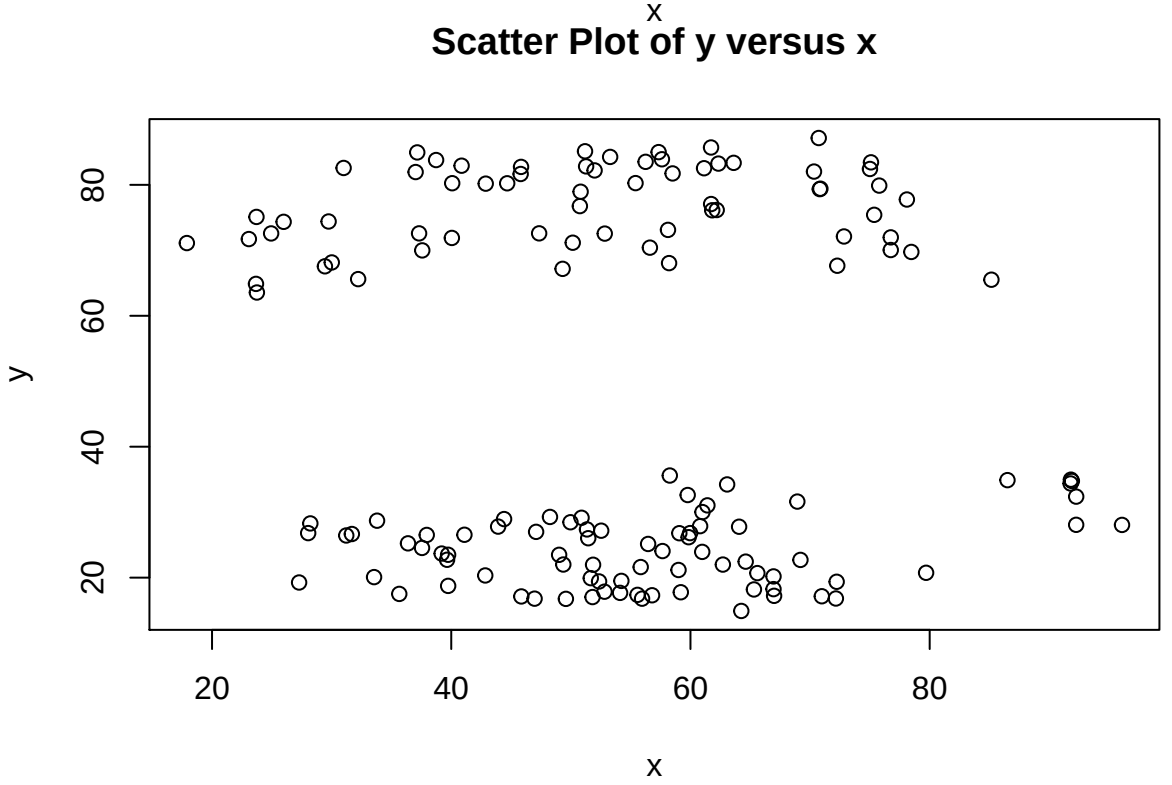

Scatter Plot of y versus x
y
x
20
40
60
80

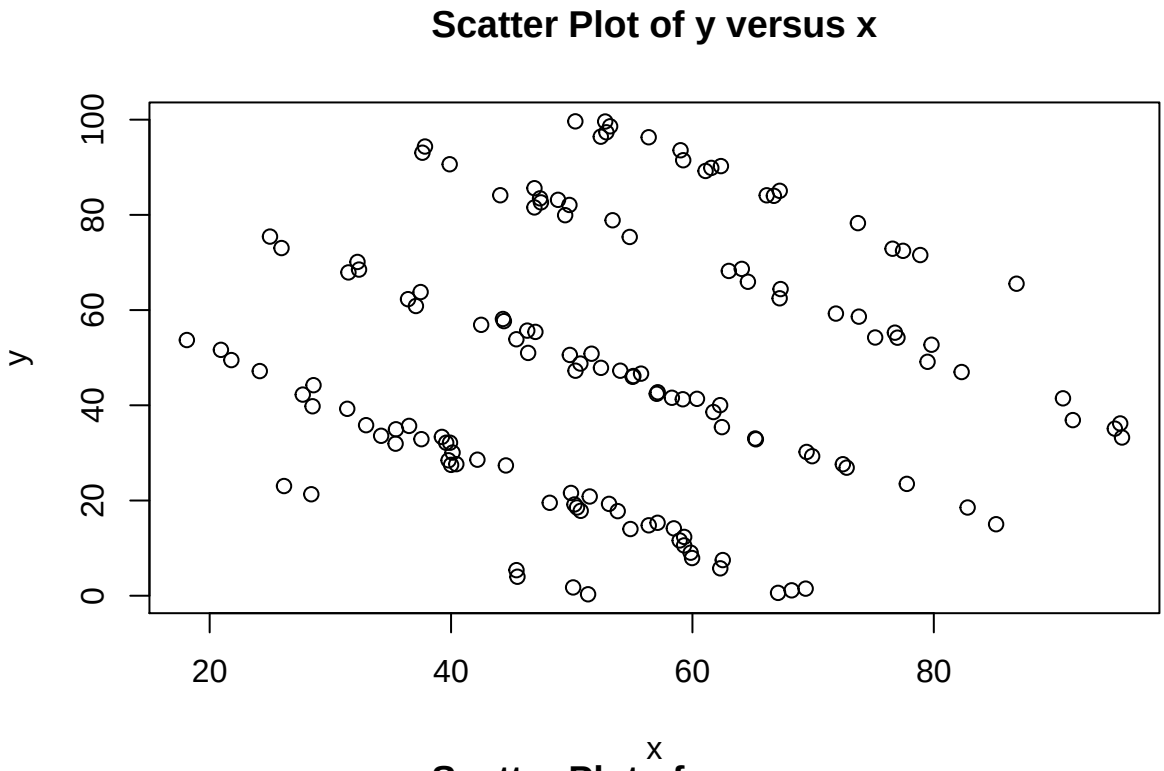

Scatter Plot of y versus x
y
x
20
30
40
50
60
70
80


Scatter Plot of y versus x
y
x
0
20
40
60
80
100
20
40
60
80
100

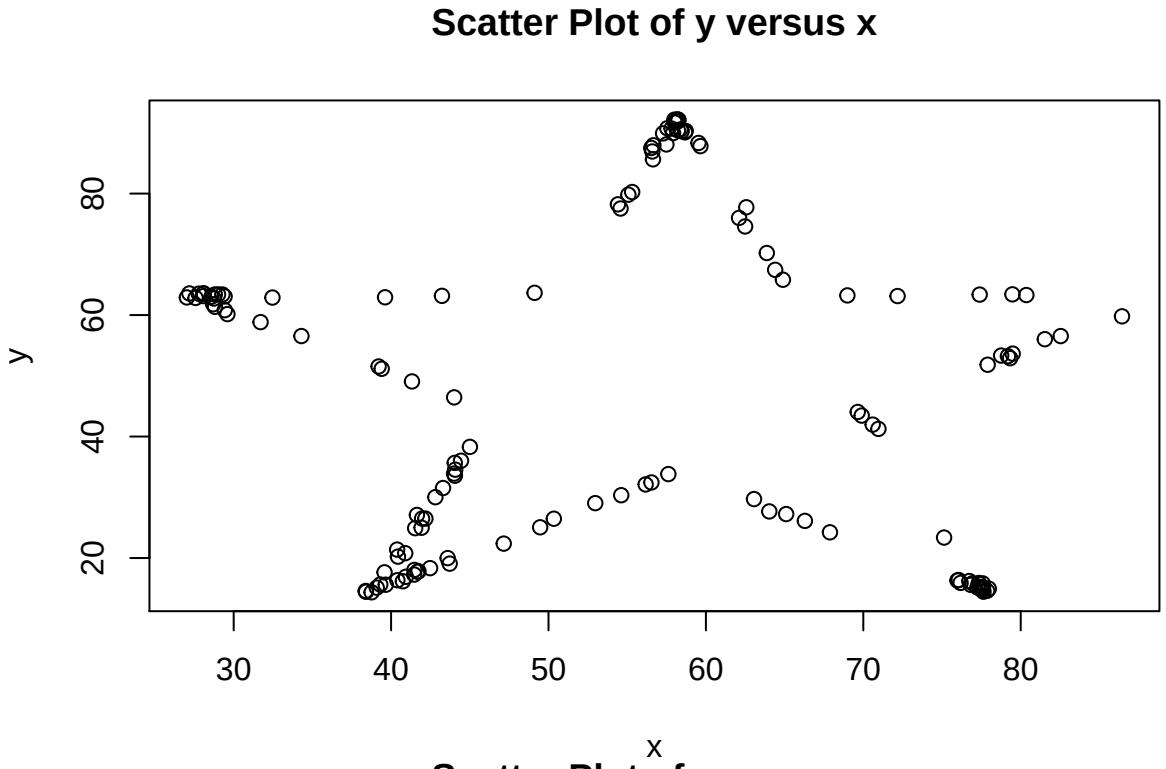

Scatter Plot of y versus x
y
x
20
40
60
80
30
40
50
60
70
80

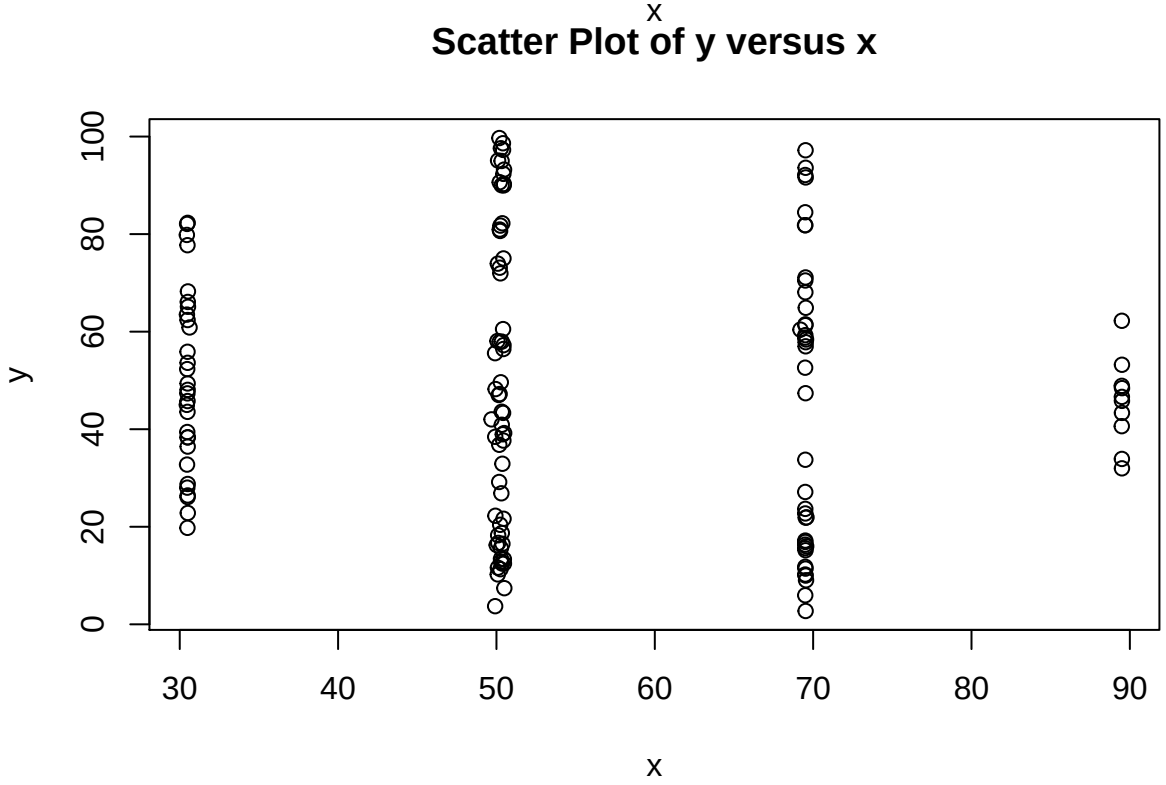

Scatter Plot of y versus x
y
x
20
40
60
80
100

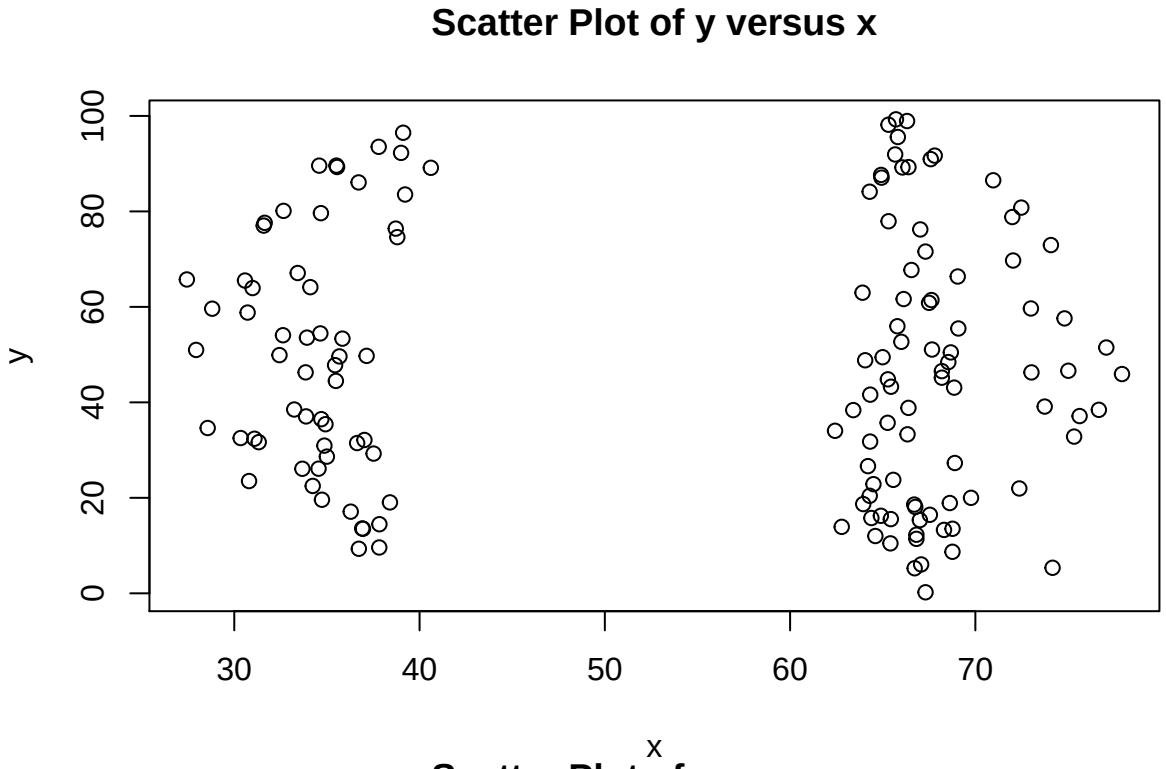

Scatter Plot of y versus x
y
x
20
40
60
80

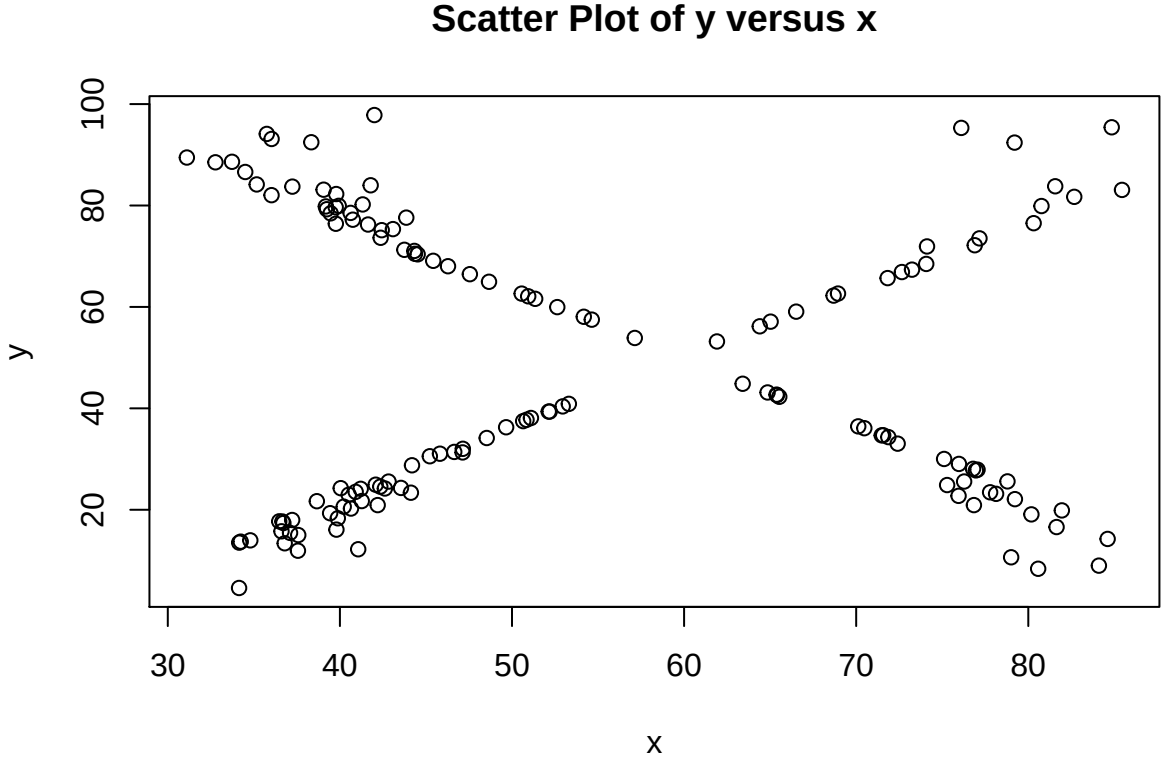

Scatter Plot of y versus x
y
x
0
20
40
60
80
100
20
40
60
80


Scatter Plot of y versus x
y
x
20
40
60
80
100
20
40
60
80


Scatter Plot of y versus x
y
x
20
40
60
80
30
40
50
60
70
80


Scatter Plot of y versus x
y
x
0
20
40
60
80
100
30
40
50
60
70
80
90


Scatter Plot of y versus x
y
x
0
20
40
60
80
100
30
40
50
60
70


Scatter Plot of y versus x
y
x
20
40
60
80
100
30
40
50
60
70
80

```
## $`1`
## NULL
##
## $`2`
## NULL
##
```

```
## $`3`
## NULL
##
## $`4`
## NULL
##
## $`5`
## NULL
##
## $`6`
## NULL
##
## $`7`
## NULL
##
## $`8`
## NULL
##
## $`9`
## NULL
##
## $`10`
## NULL
##
## $`11`
## NULL
##
## $`12`
## NULL
##
## $`13`
## NULL
```

Problem 5

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.

Part 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
    library(downloader)
    download("http://www.farinspace.com/wp-content/uploads/us_cities_and_states.zip",dest="us_cities_sta
    unzip("us_cities_states.zip", exdir="./")
    #read in data, looks like sql dump, blah
    library(data.table)
    states <- fread(input = "./us_cities_and_states/states.sql",skip = 23,sep = "'", sep2 = ",", header
    states = states[-8,]
    ### YOU do the CITIES
     cities<- fread(input = "./us_cities_and_states/cities.sql",sep = "'", sep2 = ",", header = F, sele
    ### I suggest the cities_extended.sql may have everything you need
    ### can you figure out how to limit this to the 50?
```

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

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:data.table':
##
##     between, first, last
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
library(fiftystater)
colnames(cities) = c('city', 'state')
dat = group_by(cities,state)
sum_table = summarise(dat, city_count = n())
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
sum_table = sum_table[-40,]
sum_table = sum_table[-8,]
table_final = as.data.frame(sum_table)
table_final = cbind(states[,1], table_final)
table_final = table_final[,-2]
colnames(table_final) = c('State', 'city_count')
knitr::kable(table_final)
```

State	city_count
Alaska	229
Alabama	579
Arkansas	605
Arizona	264
California	1239
Colorado	400
Connecticut	269
Delaware	57
Florida	524
Georgia	629
Hawaii	92
Iowa	937
Idaho	266
Illinois	1287
Indiana	738
Kansas	634
Kentucky	803
Louisiana	479
Massachusetts	511
Maryland	430
Maine	461
Michigan	885

State	city_count
Minnesota	810
Missouri	942
Mississippi	440
Montana	360
North Carolina	762
North Dakota	373
Nebraska	528
New Hampshire	255
New Jersey	579
New Mexico	346
Nevada	99
New York	1612
Ohio	1069
Oklahoma	585
Oregon	379
Pennsylvania	1802
Rhode Island	70
South Carolina	377
South Dakota	364
Tennessee	548
Texas	1466
Utah	250
Virginia	839
Vermont	288
Washington	493
Wisconsin	753
West Virginia	753
Wyoming	176

Part c. Create a function that counts the number of occurances 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.

```
count_letter = function(letter, state_name){
  result = lengths(regmatches(state_name, gregexpr(letter, state_name, ignore.case=TRUE)))
  return(result)
}
```

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 occurance count as a vector.

```
    ##pseudo code
letter_count <- data.frame(matrix(NA,nrow=52, ncol=26))
    #getCount <- function(what args){
       # temp <- strsplit(state_name)
        # how to count??
       # return(count)
    #}
count_letter = function(state_name, letter){
  result = lengths(regmatches(state_name, gregexpr(letter, state_name, ignore.case=TRUE)))
  return(result)
}
states_name =as.matrix(states[,1])
for(i in 1:50){
```

```
let = matrix(LETTERS, 1, 26)
let = as.data.frame(let)
colnames(let) = LETTERS
letter_count[i,] <- apply(let,2,FUN = function(x) count_letter(x,state_name=states_name[i]))
}
letter_count = as.data.frame(letter_count)
colnames(letter_count) = LETTERS
```

Part d.

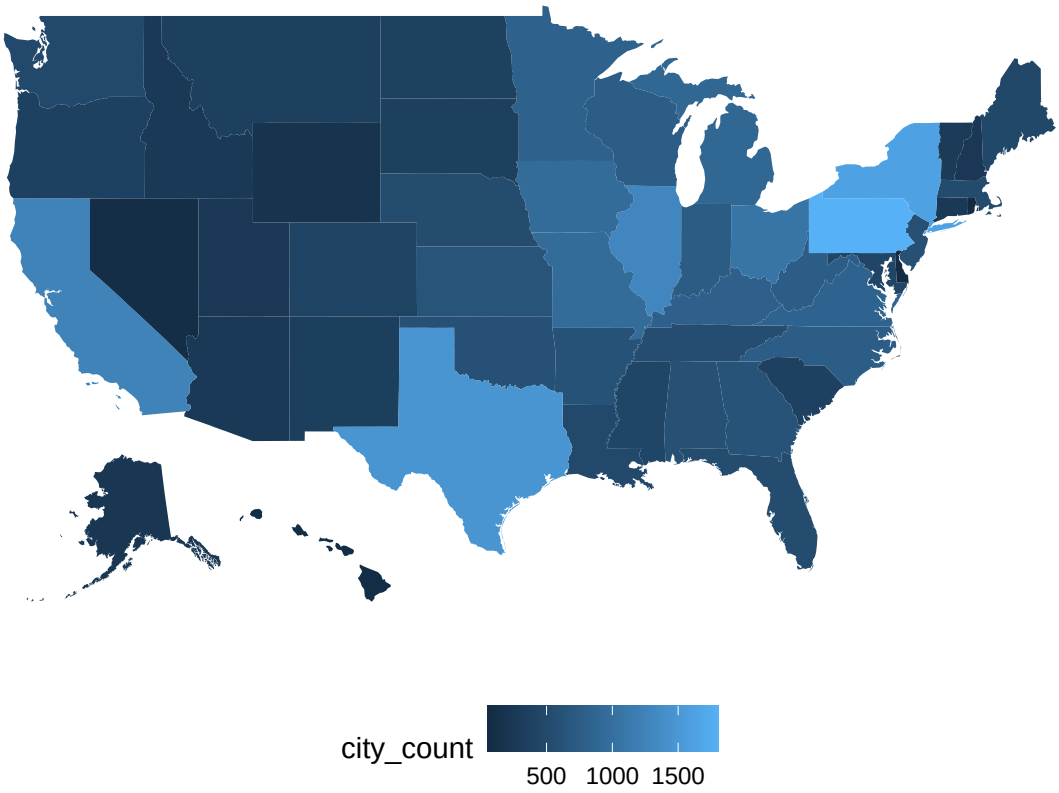
Create 2 maps to finalize this. Map 1 should be colored by count of cities on our list within the state. Map 2 should highlight only those states that have more than 3 occurances of ANY letter in their name. Ohio

Alabama Quick and not so dirty map:

```
    #https://cran.r-project.org/web/packages/fiftystater/vignettes/fiftystater.html
library(ggplot2)
library(mapproj)
```

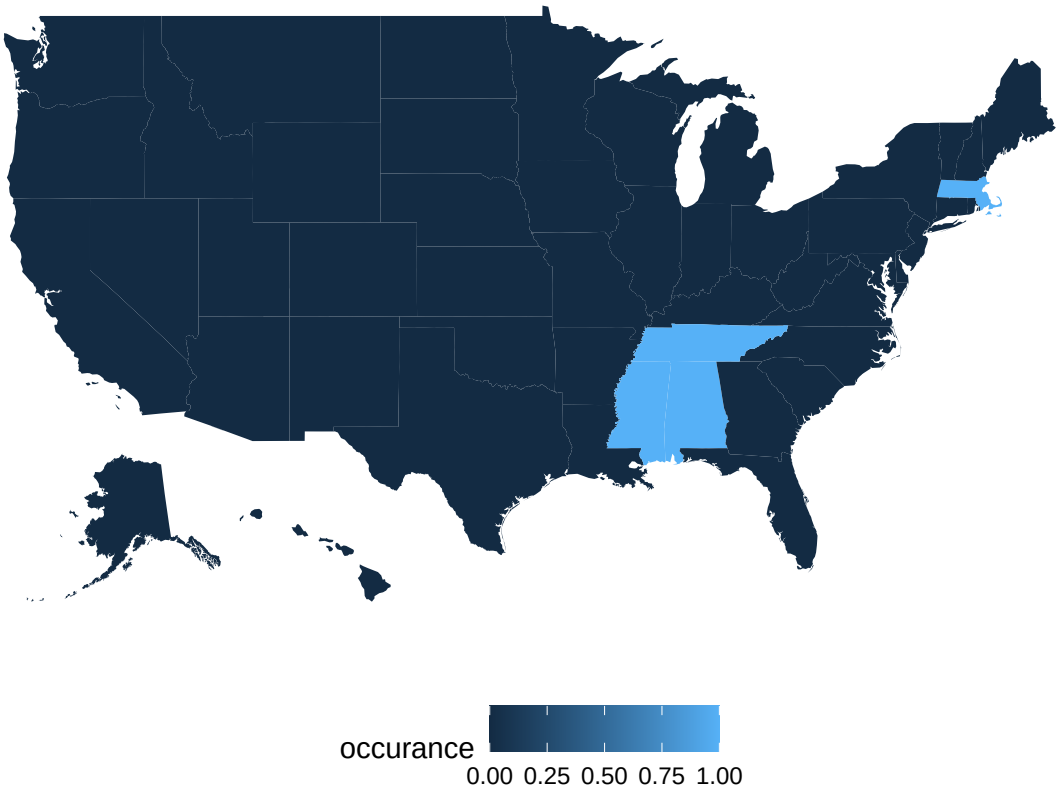
```
## Loading required package: maps
```

```
library(fiftystater)
state = as.data.frame(states)
table_gg = as.data.frame(cbind(sum_table[,2], tolower(state[,1])))
colnames(table_gg) = c('city_count', 'state')
    data("fifty_states") # this line is optional due to lazy data loading
    # map_id creates the aesthetic mapping to the state name column in your data
    p <- ggplot(table_gg, aes(map_id = state)) +
      # map points to the fifty_states shape data
      geom_map(aes(fill = city_count), 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
```




```
library(ggplot2)
library(fiftystater)
library(mapproj)
occurance = matrix(0, 50,1)
for (i in 1:50){
  trial = letter_count[i,]
  occurance[i] = length(which(trial>3))
}
```

```
occ = ifelse(occurance>=1, 1,0)
data = data.frame(table_gg$state, occ)
colnames(data) = c('state', 'occurance')
 p <- ggplot(data, aes(map_id = state)) +
      # map points to the fifty_states shape data
      geom_map(aes(fill = occurance), map = fifty_states) +
      expand_limits(x = fifty_states$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
```



Problem 2

Bootstrapping

Recall the sensory data from five operators:
http://www2.isye.gatech.edu/~jeffwu/wuhamadabook/data/Sensory.dat

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).

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.

Solution: Because in his code: 'sd.boot[i]= coef(summary(lm(logapple08~logrm08, data = bootdata)))[2,2]', each time he loops the boot, he does not use the boot variable, instead, he repeats using the previous matrix (vector) 'logapple08' and 'logrm08', thus he would get exactly same results each time.

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(data.table)
library(tidyverse)
```

```
## -- Attaching packages ------------------------------------------------ tidyverse 1.3.0 --
## v tibble  3.0.3     v purrr   0.3.4
## v tidyr   1.1.2     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.5.0
## -- Conflicts --------------------------------------------------- tidyverse_conflicts() --
## x dplyr::between()   masks data.table::between()
## x dplyr::filter()    masks stats::filter()
## x dplyr::first()     masks data.table::first()
## x dplyr::lag()       masks stats::lag()
## x dplyr::last()      masks data.table::last()
## x purrr::map()       masks maps::map()
## x purrr::transpose() masks data.table::transpose()
```

```
# Sensor Data
# Stack Method
url_sensor <- 'http://www2.isye.gatech.edu/~jeffwu/wuhamadabook/data/Sensory.dat'
lines<- fread(url_sensor,fill = TRUE, skip =1, header = TRUE)
# decide num of items
item_kind = NULL
dat = matrix(0, nrow(lines), ncol(lines)-1)
for (i in 1:nrow(lines)){
  if (sum(is.na(lines[i,]))>0) {
    dat[i,] = unlist(lines[i,1:5])
  } else {
    dat[i,] = unlist((lines[i,2:6]))
    item_kind = c(item_kind,lines[i,1])
    }
}
dat = as.data.frame(dat)
item = rep(seq_len(length(item_kind)), each = nrow(lines)/length(item_kind))
dat$item = paste(item, 'i',sep='')
colnames(dat) = c('1','2','3','4','5','item')
dat$item = as.factor(dat$item)
sensor_tidy_tv <- dat %>% gather(key = 'method', value = 'value', 1:5)
```

```
colnames(sensor_tidy_tv) = c('item', 'operator', 'value')
repeat_time = 1000 # 1000 bootstrap times
boot_coef_normal = matrix(0, repeat_time, 1)
pool = matrix(1:dim(sensor_tidy_tv)[1], ncol = length(unique(sensor_tidy_tv$operator)))
sample_function = function(x){
  result=sample(x, 30, replace = TRUE)
  return(result)
}
set.seed(0128)
i_normal = system.time(for (i in 1:repeat_time){
  ind = apply(pool, 2, sample_function)
```

```
  boots = sensor_tidy_tv[ind,]
  boot_coef_normal[i] =coef(lm(as.numeric(value) ~ as.numeric(operator), data = boots))[2]
})
```

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.

```
library(parallel)
library(iterators)
library(doParallel)
```

```
## Loading required package: foreach
```

```
##
## Attaching package: 'foreach'
```

```
## The following objects are masked from 'package:purrr':
##
##     accumulate, when
```

```
library(foreach)
no_cores = detectCores() - 1
cl <- parallel::makeCluster(no_cores, setup_strategy = "sequential")
if (Sys.getenv("RSTUDIO") == "1" && !nzchar(Sys.getenv("RSTUDIO_TERM")) &&
    Sys.info()["sysname"] == "Darwin" && getRversion() >= "4.0.0") {
  parallel:::setDefaultClusterOptions(setup_strategy = "sequential")
}
set.seed(0128)
i_parallel = system.time(boot_parallel<- foreach(i = 1:repeat_time, .combine = rbind) %dopar%{
  ind = apply(pool, 2, sample_function)
  boots = sensor_tidy_tv[ind,]
  coef(lm(as.numeric(value) ~ as.numeric(operator), data = boots))[2]
})
```

```
## Warning: executing %dopar% sequentially: no parallel backend registered
```

```
stopCluster(cl)
```

We can do the parallel programming because in bootstrap process, each boot sample is generated independently, they have no relationship with each other.

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

```
boot_normal = as.table(summary(boot_coef_normal))
colnames(boot_normal) = 'boot_normal'
boot_para = summary(boot_parallel)
colnames(boot_para) = 'boot_parallel'
sum = cbind(boot_normal, boot_para)
summary(boot_parallel)
```

```
##  as.numeric(operator)
##  Min.   :-0.41800
##  1st Qu.:-0.12900
##  Median :-0.04867
##  Mean   :-0.04934
```

```
##  3rd Qu.: 0.03333
##  Max.   : 0.38333
```

```
time_summary = cbind(i_normal[3], i_parallel[3])
sum = as.table(rbind(sum, time_summary))
rownames(sum) = c('Min', '1st Qu', 'Median', 'Mean', '3rd Qu', 'Max', 'Running Time')
knitr::kable(sum)
```

	boot_normal	boot_parallel
Min	Min. :-0.41800	Min. :-0.41800
1st Qu	1st Qu.:-0.12900	1st Qu.:-0.12900
Median	Median :-0.04867	Median :-0.04867
Mean	Mean :-0.04934	Mean :-0.04934
3rd Qu	3rd Qu.: 0.03333	3rd Qu.: 0.03333
Max	Max. : 0.38333	Max. : 0.38333
Running Time	0.886000000000422	1.1239999999998

Here, there are not too much differences regarding the running time between the normal process of bootsrap and parallel programming of bootstrap.

Problem 3

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.

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

Here, I choose the range of starting value is [-30, -10], by using (`length.out=1000`), I tried 1000 values.

```
grid = as.matrix(seq(-30,-10, length.out=1000))
# f(x)
f = function(x){
  f = 3^x - sin(x) + cos(5*x)
  return(f)
}

f_derivative <- function(x) {
    fp = 3^x*log(3) - cos(x) - 5*sin(5*x)
    return(fp)
}

Root_Newton = function(f,f_derivative, start_val, tol){
  tmp = start_val
  x = tmp
  err = 1
  i = 0
  while(err > tol){
    x = x - f(x)/(f_derivative(x))
```

```
    tmp = c(tmp,x)
    err = abs(f(x) - 0)
    i  = i+1
    if(i>10000)
    break
  }
  return(x)
}
set.seed(0128)
start_time <- Sys.time()
root_apply = apply(as.matrix(grid), 1,FUN = function(x) Root_Newton(f(x),f_derivative(x), x, tol = 1e-5)
# function not have all of data
# at least one data point not converge
# solution: add fail or ; iteration time
end_time <- Sys.time()
time_dif_apply = end_time - start_time
summary(root_apply)
```

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -498.204  -25.525  -19.635  -21.404  -14.530   -2.887
```

Part b. Repeat the apply command using the equivelant parApply command. Use 8 workers.

cl <- makeCluster(8).

```
Root_Newton = function(f,f_derivative, start_val, tol){
  tmp = start_val
  x = tmp
  err = 1
  i  = 0
  while(err > tol){
    x = x - f(x)/(f_derivative(x))
    tmp = c(tmp,x)
    err = abs(f(x) - 0)
    i = i +1
    if (i>10000) break
  }
  return(x)
}
library(parallel)
no_cores <- detectCores() - 1
cl = parallel::makeCluster(no_cores, setup_strategy = "sequential")
clusterExport(cl, 'f')
clusterExport(cl, 'f_derivative')
clusterExport(cl, 'Root_Newton')
set.seed(0128)
start_time <- Sys.time()
parallel_root = parApply(cl, as.matrix(grid), 1,FUN = function(x) Root_Newton(f(x), f_derivative(x), x,
end_time <- Sys.time()
time_dif_parallel = end_time - start_time
stopCluster(cl)
summary(parallel_root)
```

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
```

```
## -498.204  -25.525  -19.635  -21.404  -14.530   -2.887
```

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

```
# Summary Table
parallel_way = cbind(t(as.matrix(summary(parallel_root))), time_dif_apply)
apply_way = cbind(t(as.matrix(summary(root_apply))), time_dif_parallel)
summary_table = as.data.frame(rbind(apply_way,parallel_way))
rownames(summary_table) = c('Normal Apply', 'Parallel Program')
knitr::kable(summary_table)
```

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.	time_dif_parallel
Normal Apply	-498.2042	-25.52544	-19.63496	-21.40408	-14.52987	-2.887058	0.0547879
Parallel Program	-498.2042	-25.52544	-19.63496	-21.40408	-14.52987	-2.887058	0.0371401

Based on the summary table above, we can see that for part a and part b, by using apply function and parallel programming, we could get exactly same results. And the running time of parallel programming is much less than that of using apply function to get the root since to get root each time, it has to go throuhg many loops each time.

Problem 9

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