HW4_hwenjun

Wenjun Han

10/11/2020

Problem 2: Using the dual nature to our advantage

Using a mixture of true matrix math plus component operations cleans up our code giving better readibility.

```
# Use and initialize the data below
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)
```

Use the data and implement the gradient descent algorithm.

```
# First we create the function
gradient <- function(X=cbind(1,rep(1:10,10)), h = c(0,1,2,3,4),tol=0.001,
                     alpha=0.1, theta=c(1,2)){
  iter <- 0
  m <- length(h)
  theta_sp <- c(1,1,1,1)
  theta_old <- theta
  theta_new <- as.matrix(c(0,0),nrow=2)
  while(abs(theta_old[1]-theta_new[1])>tol & abs(theta_old[2]-theta_new[2])>tol){
    theta_new[1] = theta_old[1] - alpha*(1/m)*sum(X%*%theta_old - X%*%theta_new)
    theta_new[2] = theta_old[2] - alpha*(1/m)*sum((X%*%theta_old - X%*%theta_new)*X[,2])
    iter <- iter + 1
  }
  return(c(theta_new, iter))
}

# Let's use the function to apply
theta_1 <- gradient(h=h,tol=0.1,alpha=0.01, theta = theta)
theta_2 <- gradient(h=h,tol=0.01,alpha=0.01, theta = theta)
theta_3 <- gradient(h=h,tol=0.001,alpha=0.001, theta = theta)
print(c(theta_1,theta_2,theta_3))
```

```
## [1] 0.9560870 1.6985938 2.0000000 0.9933961 1.9546005 4.0000000 0.9995641
## [8] 1.9970306 2.0000000
```

```
# We compare the result with lm
lm(h~0+X)
```

```
##
## Call:
## lm(formula = h ~ 0 + X)
##
## Coefficients:
##     X1      X2
## 0.9696  2.0016
```

The tolerance we use is 0.1,0.01,0.001 and the alpha we use is 0.1 and 0.01. We find that when the step size and α is smaller, the accuracy of theta is higher. And the value of the theta matrices are similar to the lm result.

Problem 3

Part a

```
# Go parallel
library(parallel)

cores <- 5
cores <- detectCores() - 1
cores <- max(1, detectCores() - 1)

# Create a cluster via makeCluster
cl <- makeCluster(cores)

# Parallel computation, make another function

n <- 10000
tol<- 1e-09
alpha <- 1e-07
beta_0 <- seq(-1, 1, by=0.0002)
beta_1 <- seq(-1, 1, by=0.0002)
iteration <- c(rep(0,10000))
theta_result <- cbind(c(rep(0,10000)),c(rep(0,10000)))

for(i in 1:n){
  theta <- as.matrix(c(beta_0[i],beta_1[i]),nrow=2)
  result <- gradient(h=h, tol=1e-09,alpha=1e-07, theta=theta)
  iteration[i] <- result[3]
  theta_result[i,1] <- result[1]
  theta_result[i,2] <- result[2]
}
result_list <- list("beta_0"=theta_result[,1],
                    "beta_1"=theta_result[,1], "iteration"=iteration)

clusterExport(cl, list("gradient"))
# Parallelize
# theta_est <- parApply(cl,h,gradient)
stopCluster(cl)
```

Part b

If we change the stopping rule and include our knowledge of the true value, it will improve the speed of computation since it has a standard result to compare. But it is not a good way to run the algorithm, since the goal of the algorithm is to help you find the best fit β for your data. In many of the cases, the true β is unknown and we need to use algorithm to estimate its value. Thus, including true value is not doable in many situations.

Part c

Gradient descent is a computational efficient algorithm and it is easy to apply. But you may need to manually choose the step size and tolerance. The algorithm can get stuck in local optima and it needs some random re-starts. And the algorithm may never converge exactly.

Problem 4: Inverting Matrices

Calculate the β throguh matrix manipulation.

```r
# Matrix manipulation
xtxi <- solve(t(X)%*%X)
beta_est <- xtxi%*%t(X)%*%h
print(beta_est)
```

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

Through matrix manipulation we can get the β value. That is because the matrix charateristics and linear model theory, we can use the invert matrix to calculate the β in a fast way.

Problem 5: Need for speed challenge

```r
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
```

Part a

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

```
## 112347224 bytes
```

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

```
## 1816357208 bytes
```

```r
# Calculate the y
# system.time({y <- p + A%*%solve(B)%*%(q-r)})
# 529 seconds
```

The size of A is 112347224 bytes and size of B is 1816357208 bytes. The calculation time is about 529 seconds in my computer.

Part b

If we break apart the computation, it should first compute the inverse matrix of B, then calculate A times B-inverse, then calculate the whole expression. We can make mathmatical simplification to reduce the time of computing inverse matrix of B, since it is one of the most time consuming step. To do that, we can observe the matrix of B is matrix C without some of the rows and columns. And matrix C is a diagonal matrix with R as diagonal block. Thus, we can compute R inverse instead and then use it to construct B-inverse matrix.

Part C

The following chuck shows how to compute the expression in a faster way.

```r
# Since B consists of id
R_inv <- solve(R)
C_new <- kronecker(R, diag(1600)) # C is a 16000 * 16000 block diagonal matrix
B_inv <- C_new[-id, -id] # matrix of dimension 15068 * 15068
C_new <- NULL
# system.time({y <- p + A%*%B_inv%*%(q-r)})
# 98 seconds
```

Thus, we can get the result that the new method spend about 98 seconds in computing y.

Problem 3

a. Create a function that computes the proportion of successes in a vector.

```r
# Create a function
success <- function(x=c(0,1,1,0)){
  n <- length(x) # The total number of trials
  suc <- sum(x) # number of successes
  proportion <- suc/n # Calculate the proportion
  return(proportion)
}
```

b. Simulate 10 flips of a coin with varying degrees of "fairness".

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

c. Apply the function in conjuction to compute the proportion of success.

```r
# First we compute the proportion of success by column
for (i in 1:10){
  pro <- success(x=P4b_data[,i])
  list_col <- c("column", i, pro)
  print(list_col)
}
```

```
## [1] "column" "1"      "0.6"
## [1] "column" "2"      "0.6"
## [1] "column" "3"      "0.6"
## [1] "column" "4"      "0.6"
## [1] "column" "5"      "0.6"
## [1] "column" "6"      "0.6"
## [1] "column" "7"      "0.6"
## [1] "column" "8"      "0.6"
## [1] "column" "9"      "0.6"
## [1] "column" "10"     "0.6"
```

```r
# Then we compute the proportion of success by row
for (i in 1:10){
  pro <- success(x=P4b_data[i,])
  list_row <- c("row", i, pro)
  print(list_row)
}
```

```
## [1] "row" "1"   "1"
## [1] "row" "2"   "1"
## [1] "row" "3"   "1"
## [1] "row" "4"   "1"
## [1] "row" "5"   "0"
## [1] "row" "6"   "0"
## [1] "row" "7"   "0"
## [1] "row" "8"   "0"
## [1] "row" "9"   "1"
## [1] "row" "10"  "1"
```

Based on the result, we can see that for columns, the success result is the same. For the rows, the success proportion is either 0 or 1. Either way is not random but seems more like a fixed number. The matrix data we created is not as we expected, one reason is because of the seed is fixed so the result is the same.

d. Fix the above matrix

```
# Create a function to meet the requirement, input is probability
create_vect <- function(p=0.5){
  data <- rbinom(10, 1, prob = p)
}

# Create a vector of the desired probabilities
probability <- c(seq(0.31,0.4,by=0.01))

# Create a matrix
P4b_data <- sapply(probability, create_vect)

# Use function in part a to compute the success
proportion_P4b_col <- apply(P4b_data, 2, success) # by column
print(proportion_P4b_col)
```

```
##  [1] 0.2 0.3 0.4 0.3 0.4 0.6 0.3 0.3 0.5 0.6
```

```
proportion_P4b_row <- apply(P4b_data, 1, success) # by column
print(proportion_P4b_row)
```

```
##  [1] 0.7 0.3 0.5 0.5 0.3 0.1 0.8 0.4 0.1 0.2
```

Thus, we can prove that the function and this method works since we get a resonable proportion vector by column and row.

Problem 4

1. Reimport the dataset and create a dataframe and scatter plot function.

```
# Import the data from last assignment
device <- data.frame(readRDS("HW3_data.RDS"))
device_df <- data.frame(device)

# Change the label name to x and y
colnames(device_df) <- c("observer", "x", "y")

# Create a scatter plot based on x and y
scaplot <- function(X=device_df){
  x <- X[,2]
  y <- X[,3]
  return(plot(x,y))
}
```

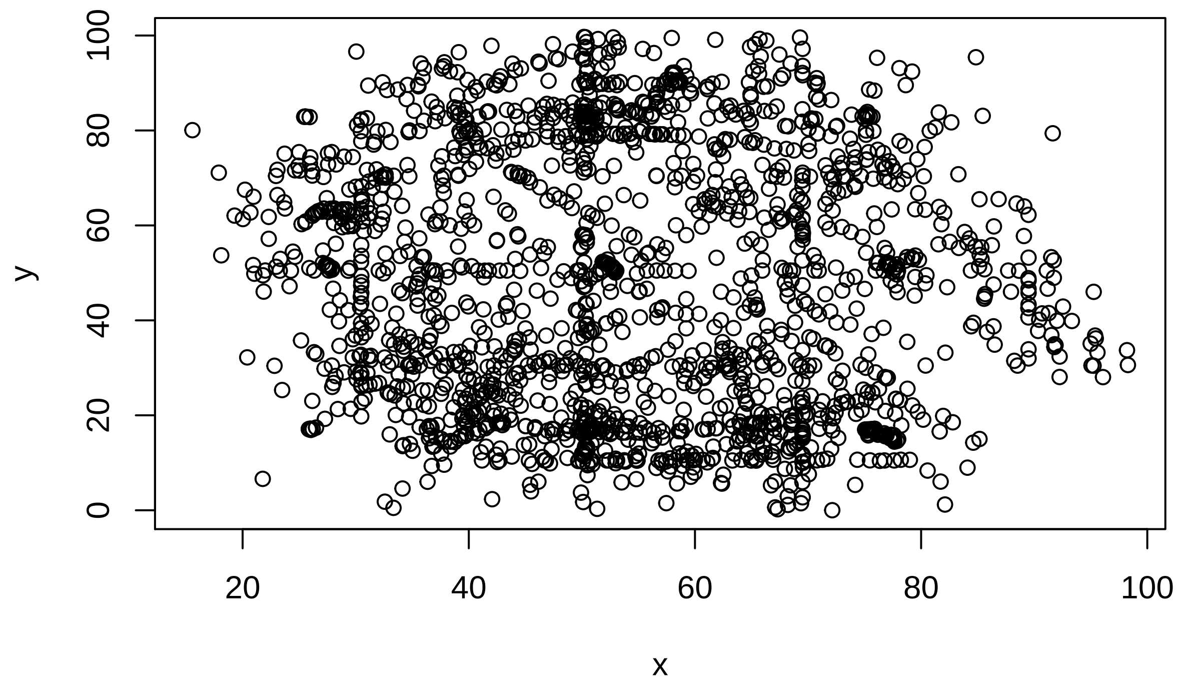
2. Use the function to create a single scatter plot of the entire dataset, and a seperate scatter plot for each observer.

```
library(dplyr)
```

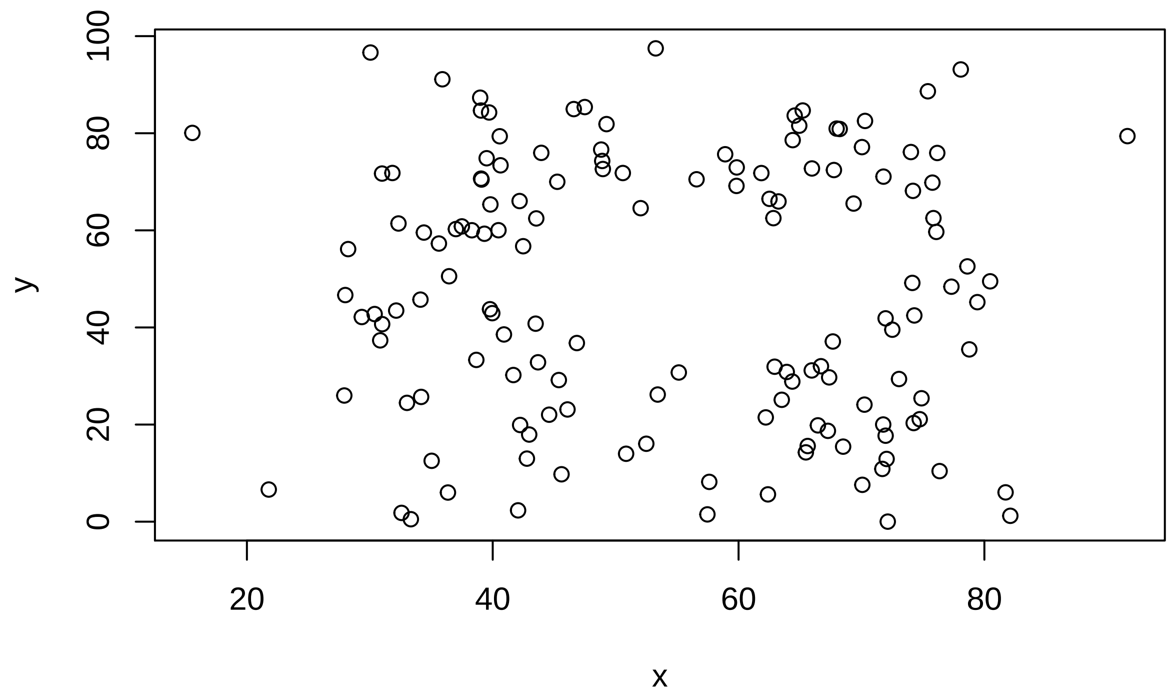
```
## Warning: package 'dplyr' was built under R version 3.6.3
```

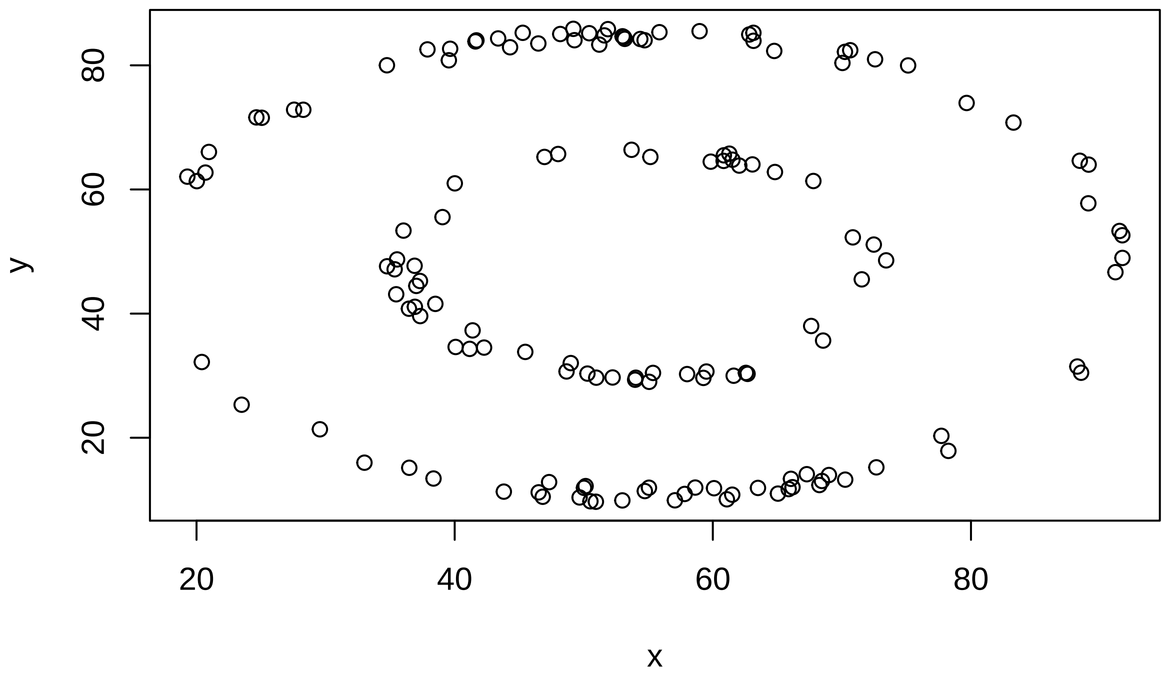
```
##
## Attaching package: 'dplyr'

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

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

```
# Create a single scatter plot of the entire dataset
scaplot()
```

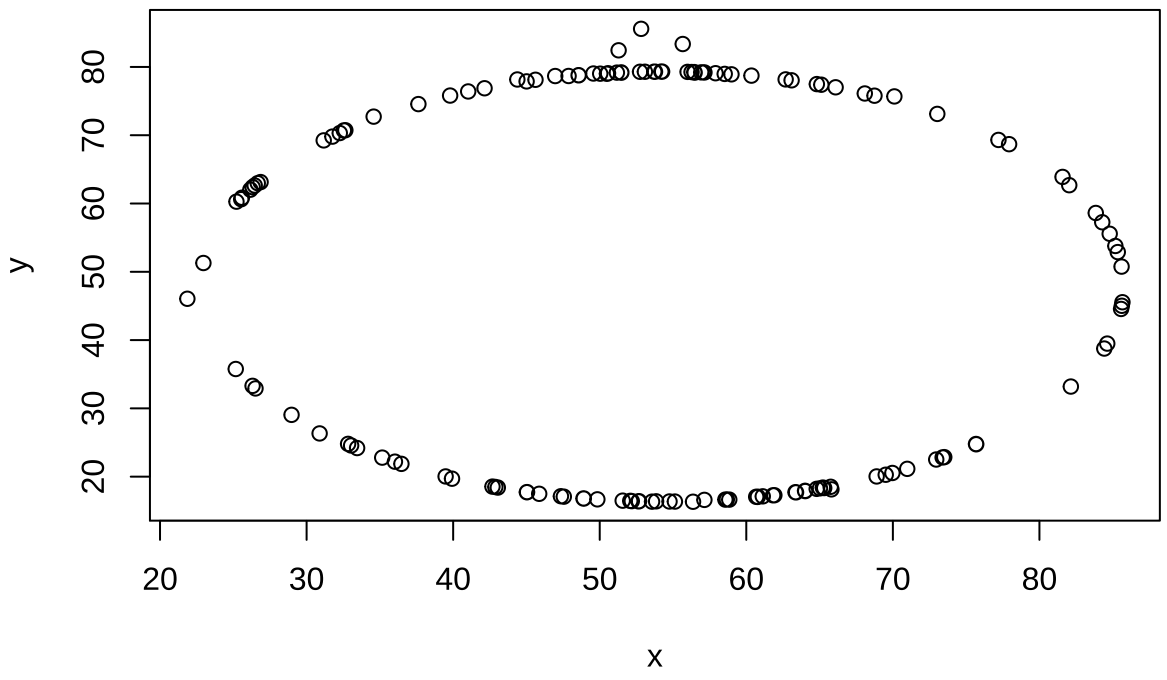


```
# Create a seperate scatter plot for each observer
for (i in 1:13){
  data_tep <- filter(device_df, observer == i)
  scaplot(data_tep)
}
```

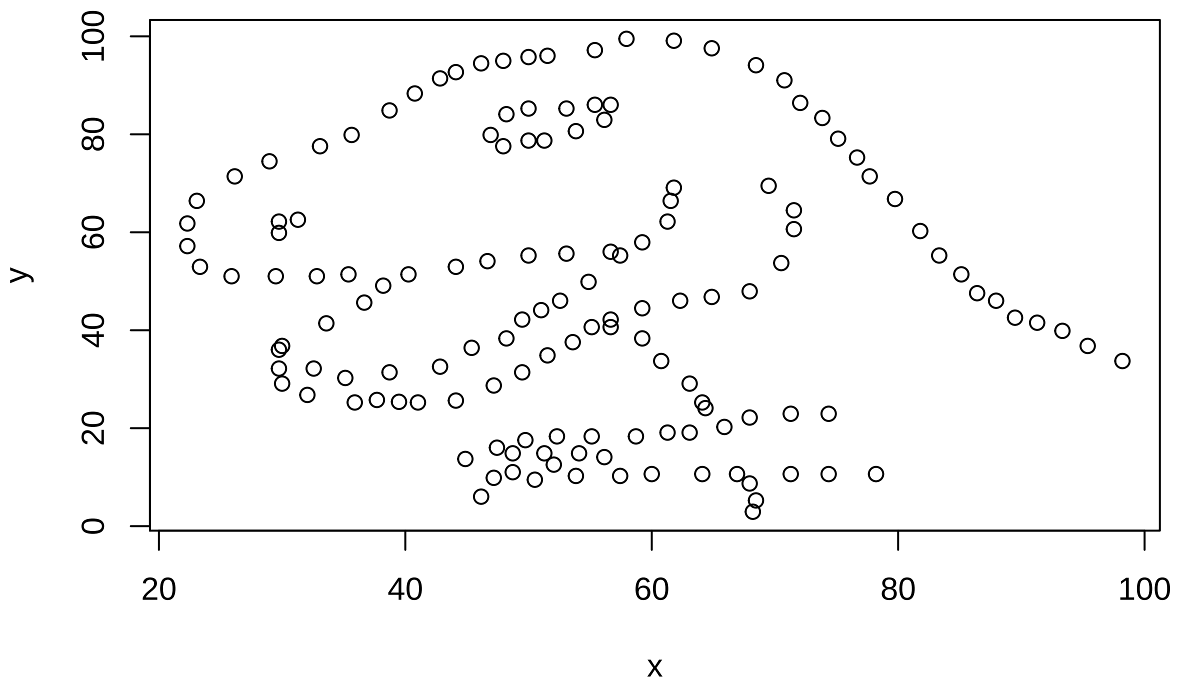

y
x
0
20
40
60
80
100
20
40
60
80

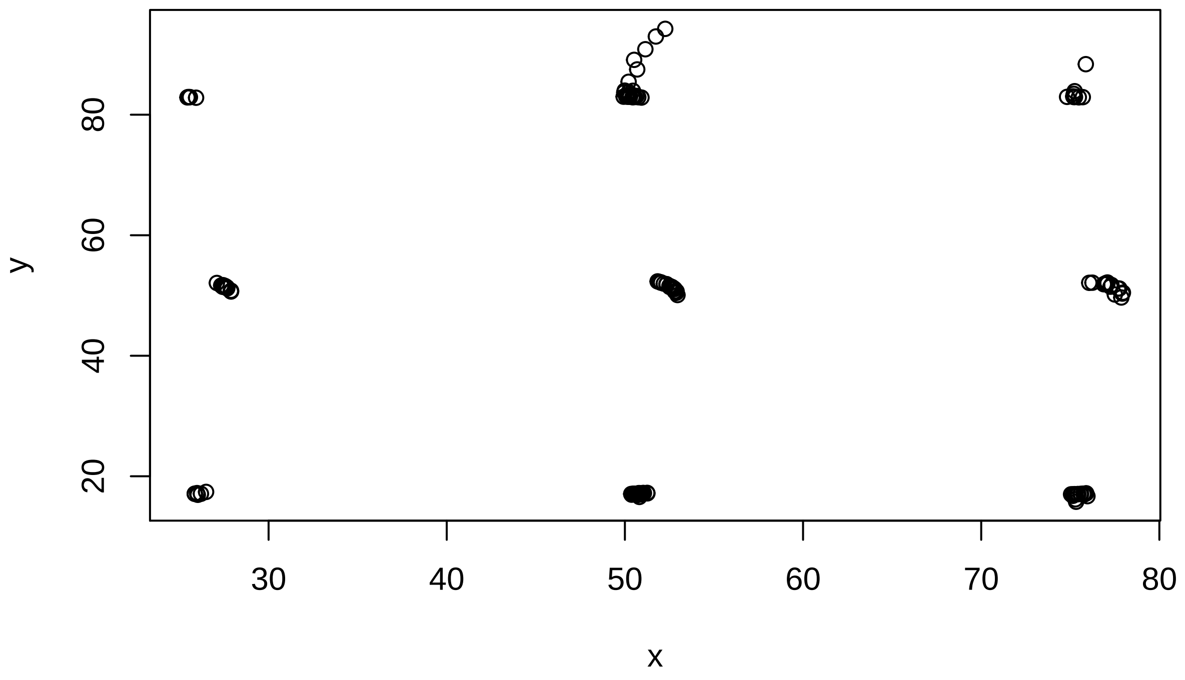


y
80
60
40
20
20
40
60
80
x

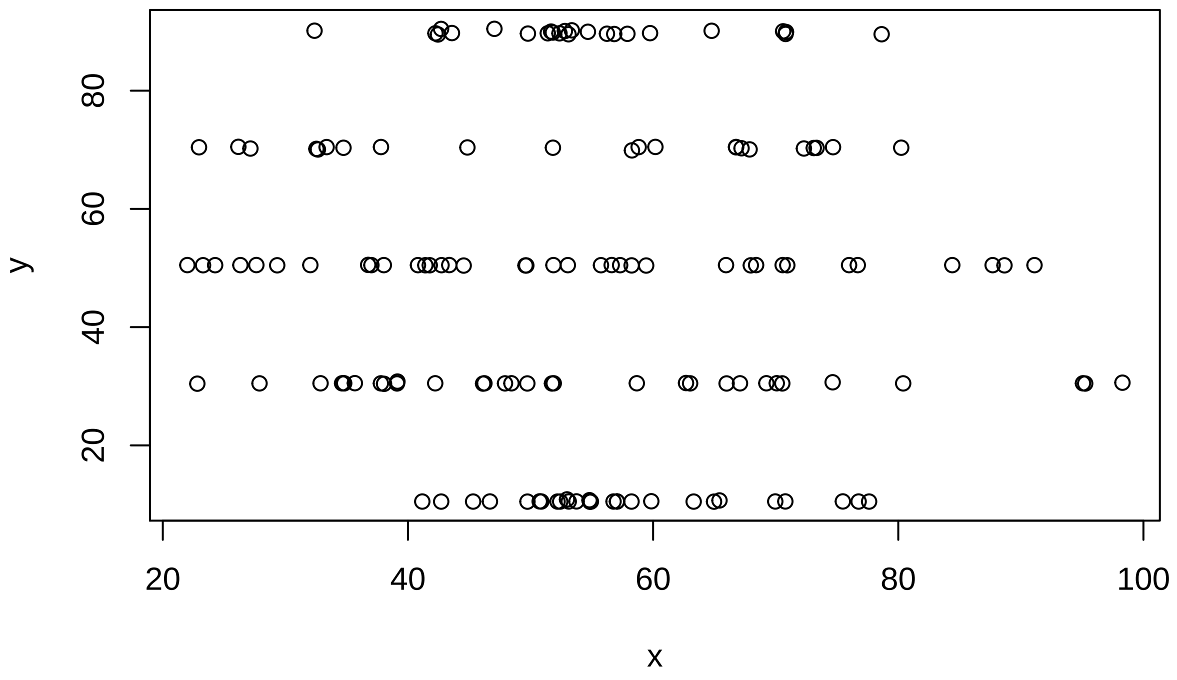


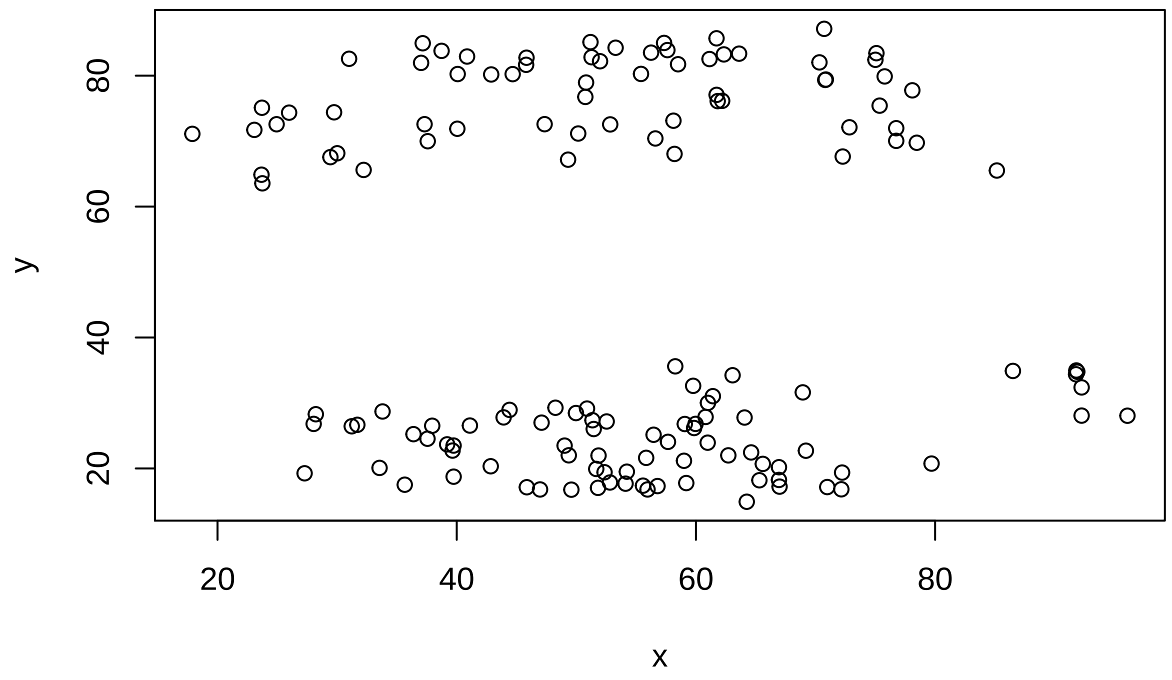
y
x
20
30
40
50
60
70
80

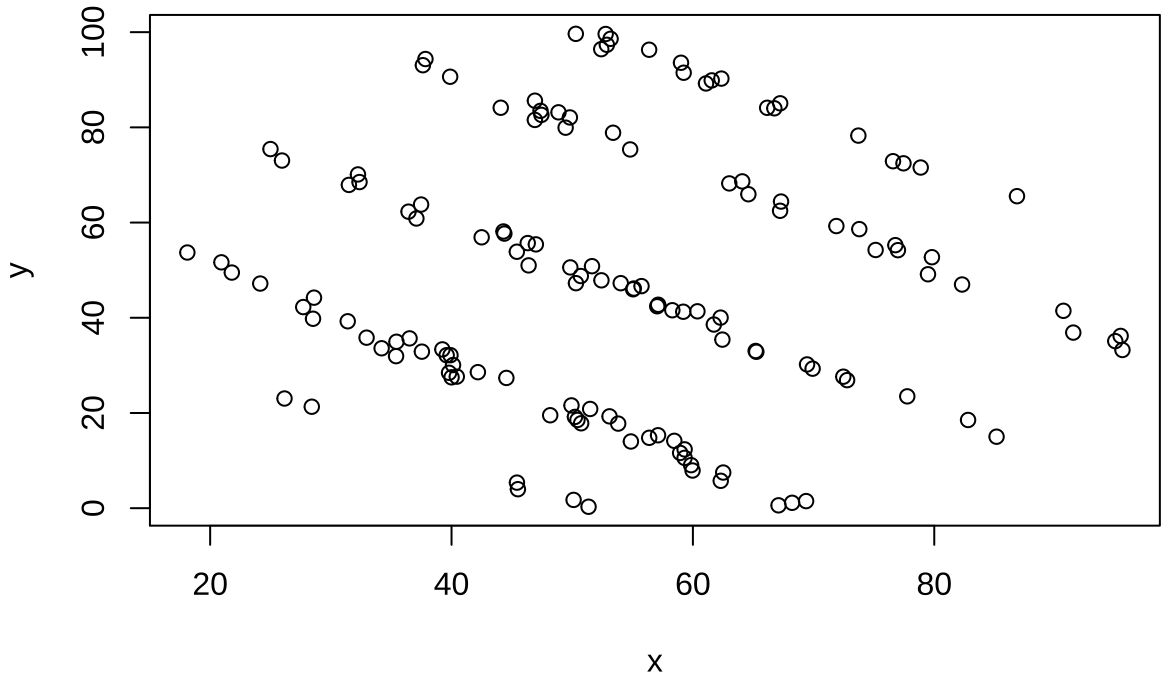

y
x
0
20
40
60
80
100
20
40
60
80
100

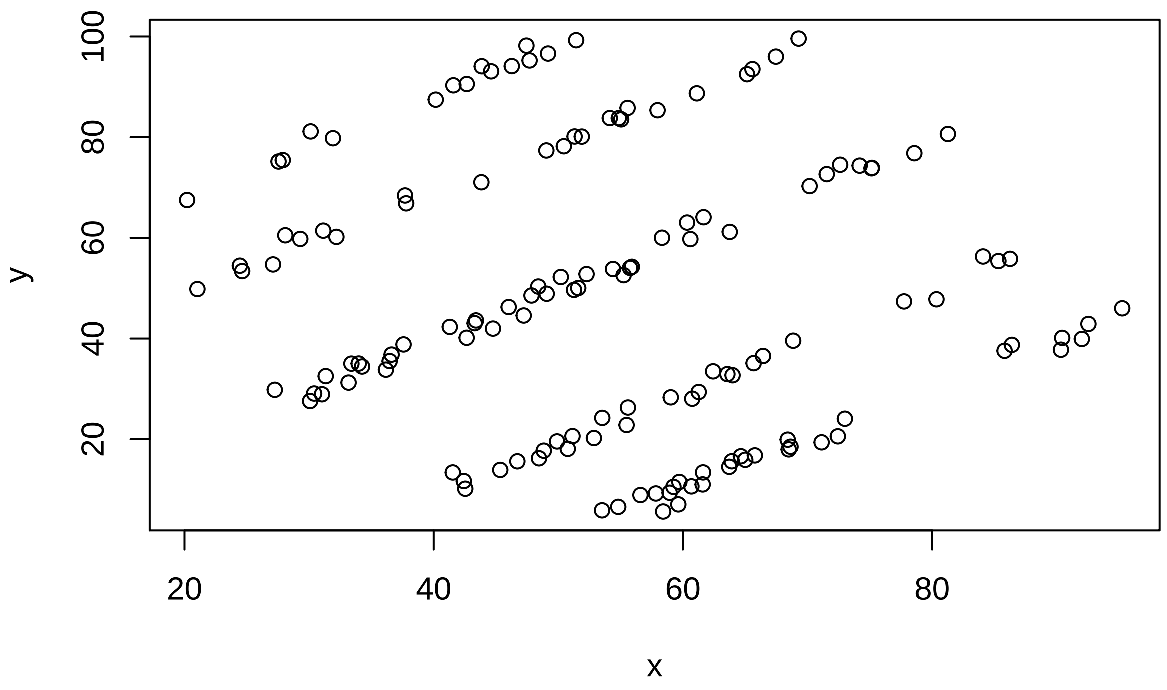


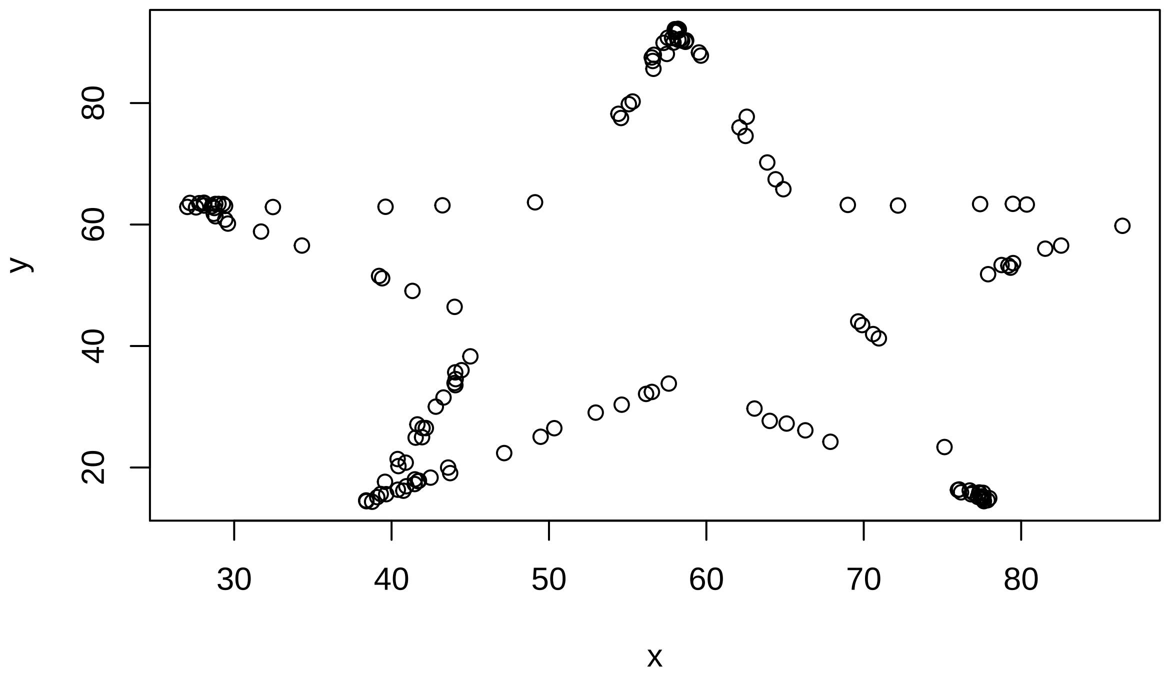
y
20
40
60
80
30
40
50
60
70
80
x


y
20
40
60
80
20
40
60
80
100
x

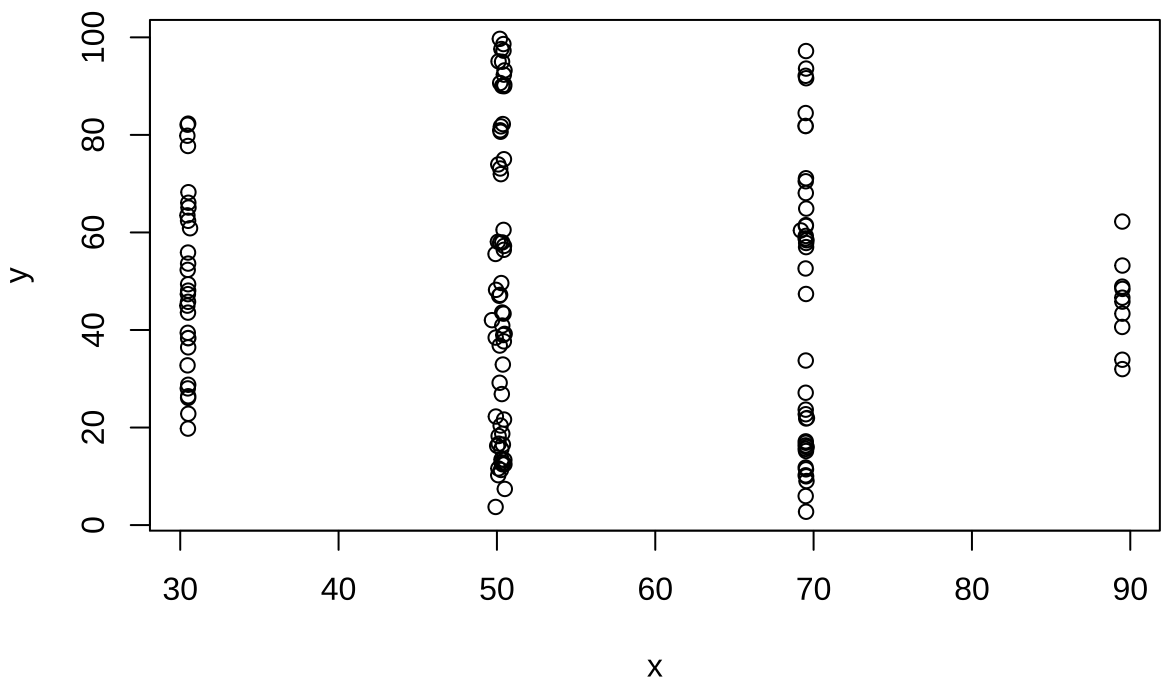

y
x
20
40
60
80
20
40
60
80

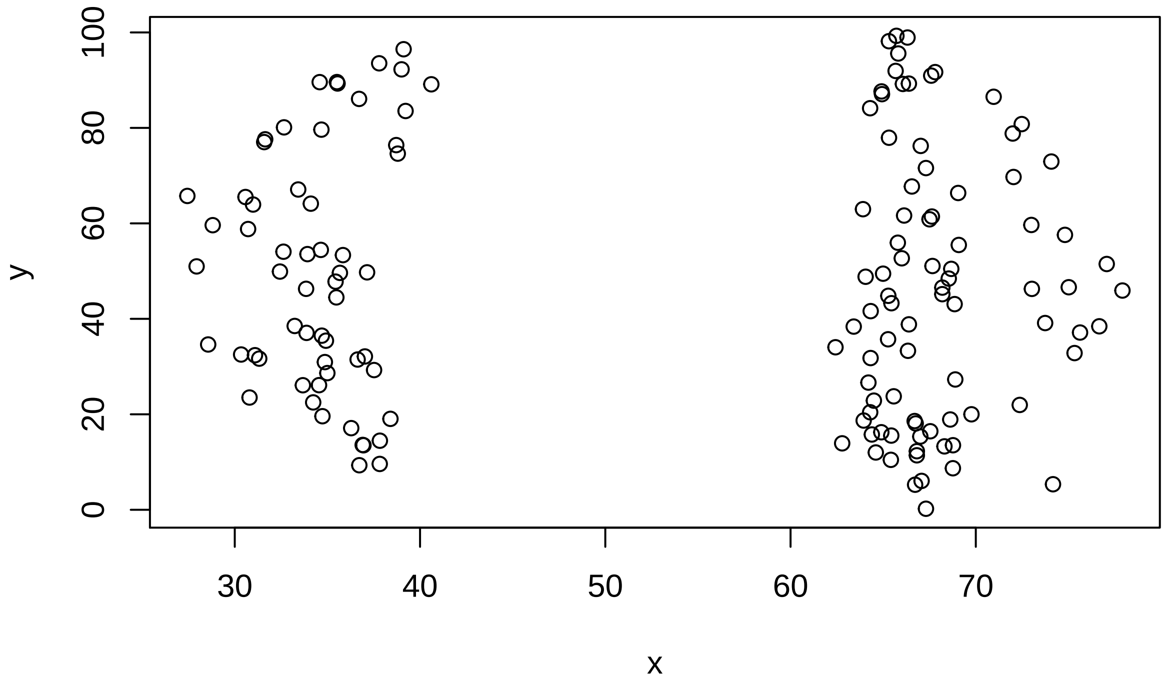

0
20
40
60
80
100
y
20
40
60
80
x

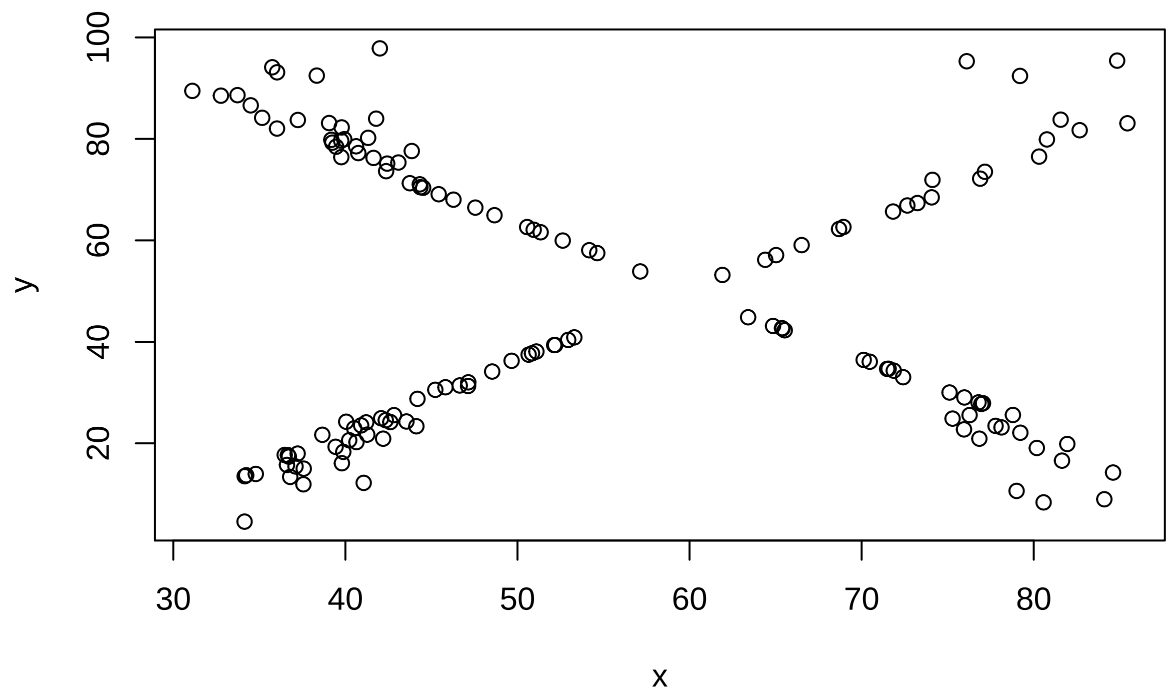

y
x
20
40
60
80
100



y
80
60
40
20
30
40
50
60
70
80
x


y
x
0
20
40
60
80
100
30
40
50
60
70
80
90


100
80
60
40
20
0
y
30
40
50
60
70
x



Problem 5

Part a

Get and import a database of US cities and states

```
#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_states.zip")
unzip("us_cities_states.zip", exdir="C:/Users/mlsoc-works/Documents/Wenjun Han/Study/STAT 5014 Stat Prog

#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 = F, select = c(2,4))
#read in data of cities
#I suggest the cities_extended.sql may have everything you need
cities <- fread(input = "./us_cities_and_states/cities_extended.sql",
                skip = 23,sep = "'", sep2 = ",", header=F, select = c(2,4))

# can you figure out how to limit this to the 50? Delete DC
states_new <- states[-8,]
```

```r
# We can find that some of the cities are repeated
cities_dis <- distinct(cities,V2,.keep_all=TRUE)
```

Part b

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

```r
# Group the data by their states

cities_num <- cities_dis %>% group_by(V4) %>% count()
cities_table <- data.frame(cities_num)
colnames(cities_table)<- c("state", "Count")
```

Part c

Create a function that counts the number of occurances of a letter in a string.

```r
# Create the function
library(stringr)
count_letter <- function(letter="a", state_name="Washington"){
  m <- str_count(state_name, pattern = letter)
  return(m)
}
```

```r
# Create a for loop to loop through the state names imported in part a.
letter_count <- data.frame(matrix(NA,nrow=50, ncol=26))

getCount <- function(letter="a", state_name="Washington"){
  count<- str_count(state_name, pattern = letter)
  return(count)
}

# We need to compare 26 letters
letter <- paste(letters,sep = "")

# Set up the for loop
for(i in 1:50){
  state <- states_new[i,1]
  for (j in 1:26) {
    letter_count[i,j] <- getCount(letter=letter[j],state_name=state)
  }
}
```

Thus, we get a table letter_count which contains count for the letters in all 50 states.

Part d

Create 2 maps to finalize this.

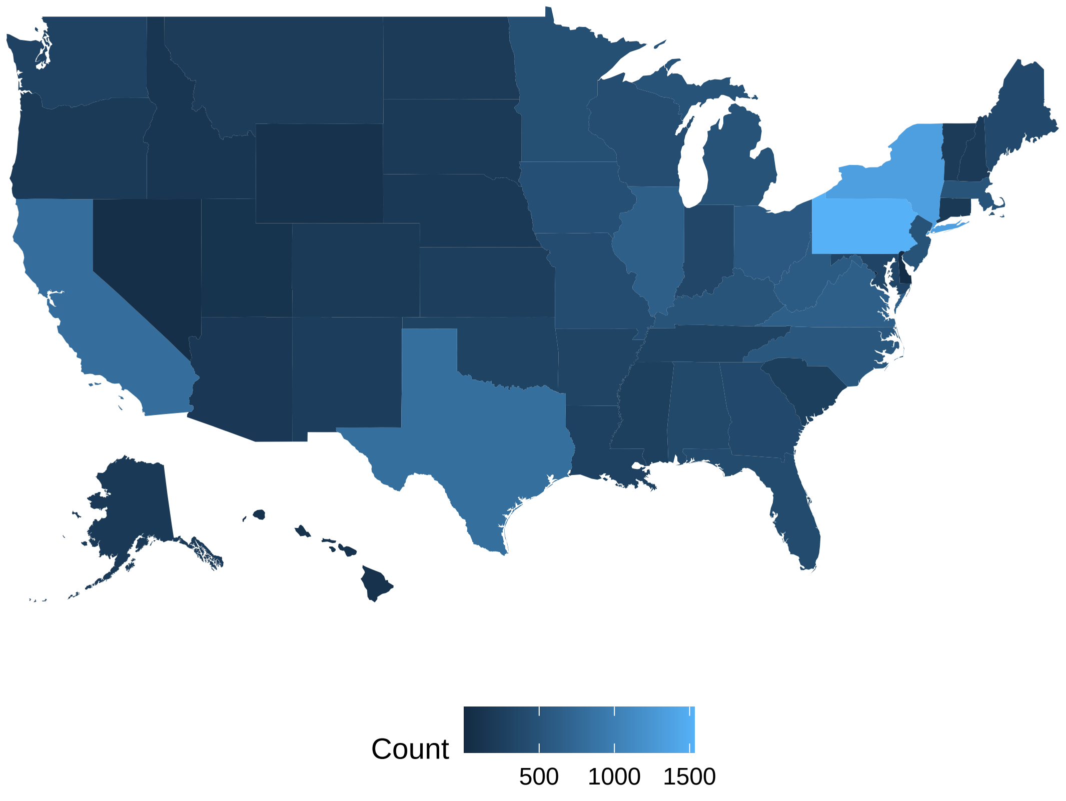
```r
#https://cran.r-project.org/web/packages/fiftystater/vignettes/fiftystater.html
library(ggplot2)
devtools::install_github("wmurphyrd/fiftystater")
```

```
## WARNING: Rtools is required to build R packages, but is not currently installed.
##
## Please download and install Rtools 4.0 from https://cran.r-project.org/bin/windows/Rtools/.

## Downloading GitHub repo wmurphyrd/fiftystater@HEAD

## WARNING: Rtools is required to build R packages, but is not currently installed.
##
## Please download and install Rtools 4.0 from https://cran.r-project.org/bin/windows/Rtools/.

##          checking for file 'C:\Users\mlsoc-works\AppData\Local\Temp\RtmpUJNFg8\remotes9ec233736a0\wmu
##       -  preparing 'fiftystater':
##    checking DESCRIPTION meta-information ...     checking DESCRIPTION meta-information ...   v  checl
##       -  checking for LF line-endings in source and make files and shell scripts
##       -  checking for empty or unneeded directories
##      Removed empty directory 'fiftystater/data-raw'
##       -  building 'fiftystater_1.0.1.tar.gz'
##
##

## Installing package into 'C:/Users/mlsoc-works/Documents/R/win-library/4.0'
## (as 'lib' is unspecified)
```

```r
library(fiftystater)
library(mapproj)
```

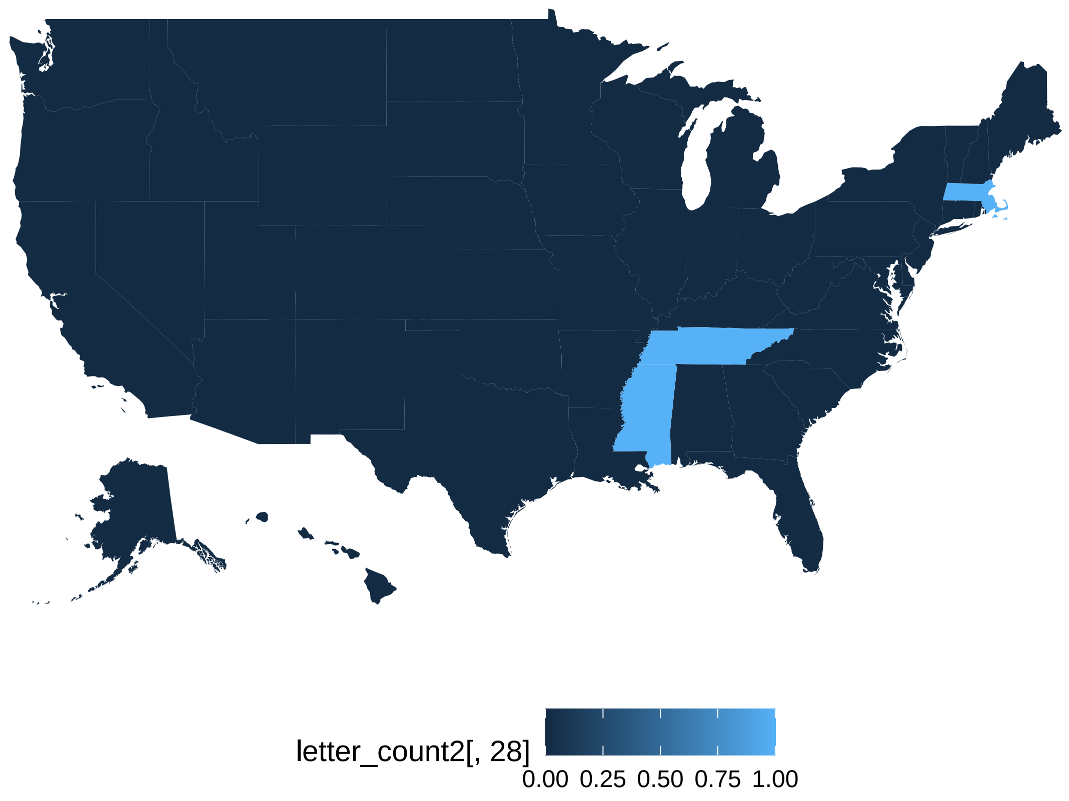
```
## Loading required package: maps
```

```r
# The first map is colored by count of cities on our list within the state
data("fifty_states") # this line is optional due to lazy data loading

# Change the state name in the city data set since it only has state code
cities_map <- cities_table
for (i in 1:52){
  if (cities_map[i,1] %in% states$V4){
    cities_map [i,1] <- states$V2[states$V4==cities_map [i,1]]
  }
}

cities_map <- cities_map[-41,]
cities_map <- cities_map[-40,]

# Lower case of the states names
cities_map$statelower <- tolower(cities_map$state)

# map_id creates the aesthetic mapping to the state name column in your data
p <- ggplot(cities_map, aes(map_id = statelower)) +
# map points to the fifty_states shape data
    geom_map(aes(fill = 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
```



```
# The second map should highlight states that have more than 3 occurances.

states_map <- states
states_map <- states[-8,] # Delete DC to fit 50 states

# Lower case of the states names
states_map$statelower <- tolower(states_map$V2)

# Calculate which states meet the requirements
letter_count2 <- letter_count
letter_count2 <- cbind(letter_count2, states_map$statelower)
letter_count2 <- cbind(letter_count2, rep(0,50))
```

```r
for (i in 1:50){
  for (j in 1:26){
    if (letter_count2[i,j] > 3){
      letter_count2[i,28] <- 1
    }
  }
}

# Draw the map
g <- ggplot(letter_count2, aes(map_id = letter_count2[,27])) +
# map points to the fifty_states shape data
    geom_map(aes(fill = letter_count2[,28]), 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())

g
```



Problem 2

Part a

The supplied code is not working because it has fixed the value of logapple08 and logrm08,so the resampling result of each iteration is exactly the same.

Let's correct the code and redo the question.

```
# install and call the library
library(quantreg)
```

```
## Loading required package: SparseM

##
## Attaching package: 'SparseM'

## The following object is masked from 'package:base':
##
##     backsolve
```

```
library(quantmod)
```

```
## Loading required package: xts

## Loading required package: zoo

##
## Attaching package: 'zoo'

## The following objects are masked from 'package:base':
##
##     as.Date, as.Date.numeric

##
## Attaching package: 'xts'

## The following objects are masked from 'package:dplyr':
##
##     first, last

## Loading required package: TTR

## Registered S3 method overwritten by 'quantmod':
##   method            from
##   as.zoo.data.frame zoo

## Version 0.4-0 included new data defaults. See ?getSymbols.
```

```r
#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.
```

```r
#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)
colnames(df08) <- c("logapple08", "logrm08") #Add one line here
set.seed(666)
Boot_times<-1000
sd.boot<-rep(0,Boot_times)

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]
}
```

Part b

Get the parameter estimates using 100 bootstrapped samples. Use sensory data.

```r
# import the sensory data
library(data.table)
```

```
##
## Attaching package: 'data.table'

## The following objects are masked from 'package:xts':
##
##     first, last
## The following objects are masked from 'package:dplyr':
##
##     between, first, last
```

```
## http://www2.isye.gatech.edu/~jeffwu/wuhamadabook/data/Sensory.dat"
url1 <- "http://www2.isye.gatech.edu/~jeffwu/wuhamadabook/data/Sensory.dat"
sensory_data_raw <- fread(url1, fill=TRUE, data.table = FALSE)
saveRDS(sensory_data_raw, "sensory_data_raw.RDS")
sensory_data_raw <- readRDS("sensory_data_raw.RDS")

# Redo the data cleaning
sensory_data_raw_dl <- sensory_data_raw[-1,-2]

# Then we convert the value from string to numeric
library(stringr)
sensory_data_raw_dl_nu <- as.numeric(unlist(str_extract_all
                                            (sensory_data_raw_dl, "[.-9]+")))

# We know that 1-10 are number of item, not true value in our table, so we delete them
sensory_data_raw_value <- sensory_data_raw_dl_nu[-c(1,17,33,49,65,81,97,113,129,145)]

## Then we reconstruct data
sensory_data_tidy_br <- data.frame(item=sort(rep(1:10,15)),
                                   operator=rep(c(1,1,1,2,2,2,3,3,3,4,4,4,5,5,5),10),
                               values=sensory_data_raw_value)
sensory <- sensory_data_tidy_br[,-1]
colnames(sensory) <- c("operator","y")

sd.boot<-rep(0,Boot_times)
set.seed(777)
# Bootstrap using the sensory data
time1<-system.time(for(i in 1:Boot_times){
# nonparametric bootstrap
  bootdata<-sensory[sample(nrow(sensory), size = 100, replace = TRUE),]
  sd.boot[i]<- coef(summary(lm(y~operator, data = bootdata)))[2,2]
})
time1
```

```
##    user  system elapsed
##    0.71    0.00    0.72
```

Part c

Redo the last problem but run the bootstraps in parallel.

```
# Go parallel
library(parallel)
library(foreach)
library(doParallel)

cores <- max(1, detectCores() - 1)

# Create a cluster via makeCluster
```

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

# Parallelize
Boot_times<-1000
sd.boot<-rep(0,Boot_times)
set.seed(777)
clusterExport(cl, list("sensory","Boot_times","sd.boot"))

# Bootstrap using the sensory data

time2<-system.time(foreach(i=1:Boot_times) %dopar% {
 #nonparametric bootstrap
    bootdata<-sensory[sample(nrow(sensory), size = 100, replace = TRUE),]
    sd.boot[i]<- coef(summary(lm(y~operator, data = bootdata)))[2,2]
})


#time2<-system.time(foreach(i=1:Boot_times)%dopar%{
# nonparametric bootstrap
#  bootdata<-sensory[sample(nrow(sensory), size = 100, replace = TRUE),]
#  sd.boot[i]<- coef(summary(lm(y~operator, data = bootdata)))[2,2]
#})

time2
```

```
##    user  system elapsed
##    0.17    0.03    0.36
```

```
stopCluster(cl)
```

Create a single table summarizing the results and timing from part a and b.

```
# Create table for comparing time
time_all <-rbind(time1,time2)[,-c(4,5)]
print(time_all)
```

```
##       user.self sys.self elapsed
## time1      0.71     0.00    0.72
## time2      0.17     0.03    0.36
```

Based on the table, we can see that using parallel computing greatly decrease the time of running codes. It help increase computing efficiency especially when data size is very large.

Problem 3

Part a

First build up the newton method algorithm, and create a vector from -1 to 1. Then apply the function and calculate the system time it takes.

```r
#We build up the newton's method algorithm as in the last homework.
# We input the function

fx <- function(x=1){
  f <- 3^x-sin(x)+cos(5*x)
  return(f)
}

# find its derivative
dr <- function(x=1){
  d <- (3^x)*log(3, base = exp(1)) - cos(x) - 5*sin(5*x)
}

newton <- function(m=1){
  # perform newton method
  tolerance <- 0.5
  while(abs(fx(m))>tolerance){
    new_point <- m - (fx(m)/dr(m))
    m <- new_point
  }
  return(m)
}

# Create a vector
vec <- seq(-1,1, by=0.002)

# Using on of the apply functions and find the roots
#system.time(result <- lapply(vec, newton))
```

Part b

Now we are going to use parallep computing to improve the calculation efficiency. In this case, we are going to use 8 workers to do the job.

```r
# Go parallel
library(parallel)

cores <- 8
# Create a cluster via makeCluster
cl <- makeCluster(cores)
clusterExport(cl, list("fx", "dr"))

# Parallelize
# Create a vector
vec <- seq(-1,1, by=0.002)

# Using on of the apply functions and find the roots
system.time(result <- parSapply(cl,vec, newton))
```

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

```r
stopCluster(cl)
```