# Homework 4

*Ankur Patel*

*October 13, 2020*

Note: Because some problems were taking very long to run, I could not get the output. I generated the pdf by commenting out the parts that were running for a long time. I then uncommented them when I pushed the markdown to git.

## Problem 2: Using the dual nature to our advantage

```
set.seed(1256)
#generate the data
theta <- as.matrix(c(1,2),nrow=2)
X <- cbind(1,rep(1:10,10))
h <- X%*%theta+rnorm(100,0,0.2)
m <- 100

#fit the linear model using lm, print the coefficients
lm_fit <- lm(h ~ 0 + X)
print(lm_fit)
```

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

```
#tol is the tolerance
tol <- 1e-6
#alpha is the step size
alpha <- 1e-2
theta_curr <- as.matrix(c(0.5,1),nrow =2)

grad_descent <- function(theta_curr,alpha,m,X,h,tol,max_iter)
{
  iter <- 1
  theta_new <- theta_curr - (alpha/m)*t(X)%*%(X%*%theta_curr-h)
  while((abs(theta_curr[1]-theta_new[1])) > tol && (abs(theta_curr[2]-theta_new[2])) > tol)
  {
    iter <- iter + 1
    if (((abs(theta_curr[1]-theta_new[1])) < tol && (abs(theta_curr[2]-theta_new[2]) < tol)) || (iter >
    {
      theta_final <- theta_new
      return(c(theta_final,max_iter))
    }
    theta_curr <- theta_new
    theta_new <- theta_curr - (alpha/m)*t(X)%*%(X%*%theta_curr-h)
  }
}
print(grad_descent(theta_curr,alpha,m,X,h,tol,max_iter = 1000))
```

```
## [1]    0.9305694    2.0071652 1000.0000000
```

The tolerance I used was 1e-6 and $\alpha = 1e-2$. For lm,the estimate for $\theta_0$ was 0.9696 and for $theta_1$ it was 2.0016. I capped the number of iterations at 1000 and the final values were 0.8707 for $\theta_0$ and 2.0158 for $\theta_1$ for gradient descent; compared to lm, gradient descent did slightly worse.

## Problem 3: Gradient Descent

```
set.seed(12456)
library(foreach)
library(parallel)
#make the cluster, set the number of cores
# cores <- max(1,detectCores()-1)
# cl <- makeCluster(cores)
#maximum number of iterations
max_iter <- 5000
runs <- 100
#sample from a uniform (-1,1) which will helps us get theta_start
U <- runif(runs,-1,1)
#the vector of starting values
theta0_start <- 1 + U
theta1_start <- 2 + U
theta_start <- cbind(theta0_start,theta1_start)
#step size and tolerance
alpha <- 1e-7
tol <- 1e-9
#run gradient descent in parallel
# foreach (i=1:runs,.combine="rbind") %do%
# {
#   grad_descent(theta_start[i,],alpha,m,X,h,tol,max_iter = max_iter)
# }
#stopCluster(cl)
```

If we changed the stopping rule to include our knowledge of the true value, it could improve accuracy by ensuring that we converge to the right solution because the tolerance would be with respect to the solution and the true value. On the other hand, it could cause the gradient descent algorithm to run for a very long time to achieve convergence. This algorithm works well if we choose $\alpha$ and the tolerance reasonably well. However, if we don't it can run endlessly.

Even in parallel, this computation was taking my computer over 3 hours to run; I was not able to get any results.

## Problem 4: Inverting Matrices

We would need to compute $X'X$ and then factorize it to make the inversion less expensive. Since $X'X$ is symmetric, we could find a suitable decomposition for symmetric matrices. In the current computation, we are computing and then inverting $X'X$ which is expensive and then multiplying by $X'y$.

## Problem 5: Need for speed challenge

```
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)
#print the sizes of A and B
print(object.size(A))
```

```
## 112347208 bytes
```

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

```
## 1816357192 bytes
```

```
#system.time(y <- p + A %*% solve(B)%*%(q-r))
C<-NULL #save some memory space
#chol_B <- chol(B)
#system.time(y <- p + A %*% chol2inv(chol_B)%*%(q-r))
```

$A$ is 112347208 bytes, $B$ is 1816357192 bytes. Without any optimization tricks, it took about 12.4 minutes to calculate y. It makes the most sense to try to decompose B and then take the inverse to speed up the calculation. I tried using the cholesky decomposition and then chol2inv but it is still not able to compute it fast.

## Problem 3 Proportion of Successes:

```
calc_prop_success <- function(x)
{
  n <- length(x)
  prop <- sum(x)/n
  return(prop)
}
set.seed(12345)
P4b_data <- matrix(rbinom(10, 1, prob = (31:40)/100), nrow = 10, ncol = 10, byrow = FALSE)
row_prop <- apply(P4b_data,1,calc_prop_success)
col_prop <- apply(P4b_data,2,calc_prop_success)
print(row_prop)
```

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

```
print(col_prop)
```

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

```
#gen_flips is a function that takes in probability p and returns vector
#whose elements are the outcomes of 10 flips of a coin
gen_flips <- function(p)
{
  y <- rbinom(10,1,p)
  return(y)
}
test_gen_flips <- gen_flips(0.25)
print(test_gen_flips)
```

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

```
prob_vector <- seq(31,40)/100
P4b_data_corrected <- sapply(X = prob_vector,FUN = gen_flips)
row_prop_correct <- apply(P4b_data_corrected,1,calc_prop_success)
col_prop_correct <- apply(P4b_data_corrected,2,calc_prop_success)
print(row_prop_correct)
```

```
##  [1] 0.8 0.3 0.5 0.5 0.4 0.2 0.9 0.4 0.1 0.1
```

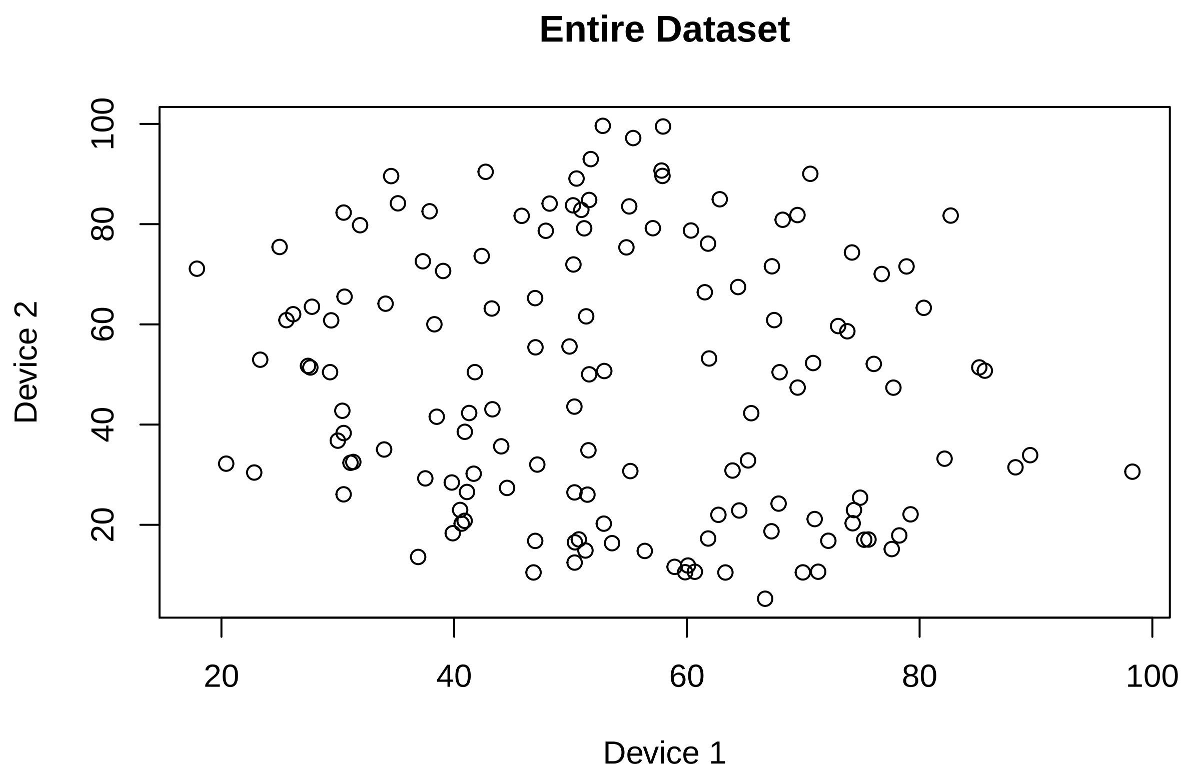
```
print(col_prop_correct)
```

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

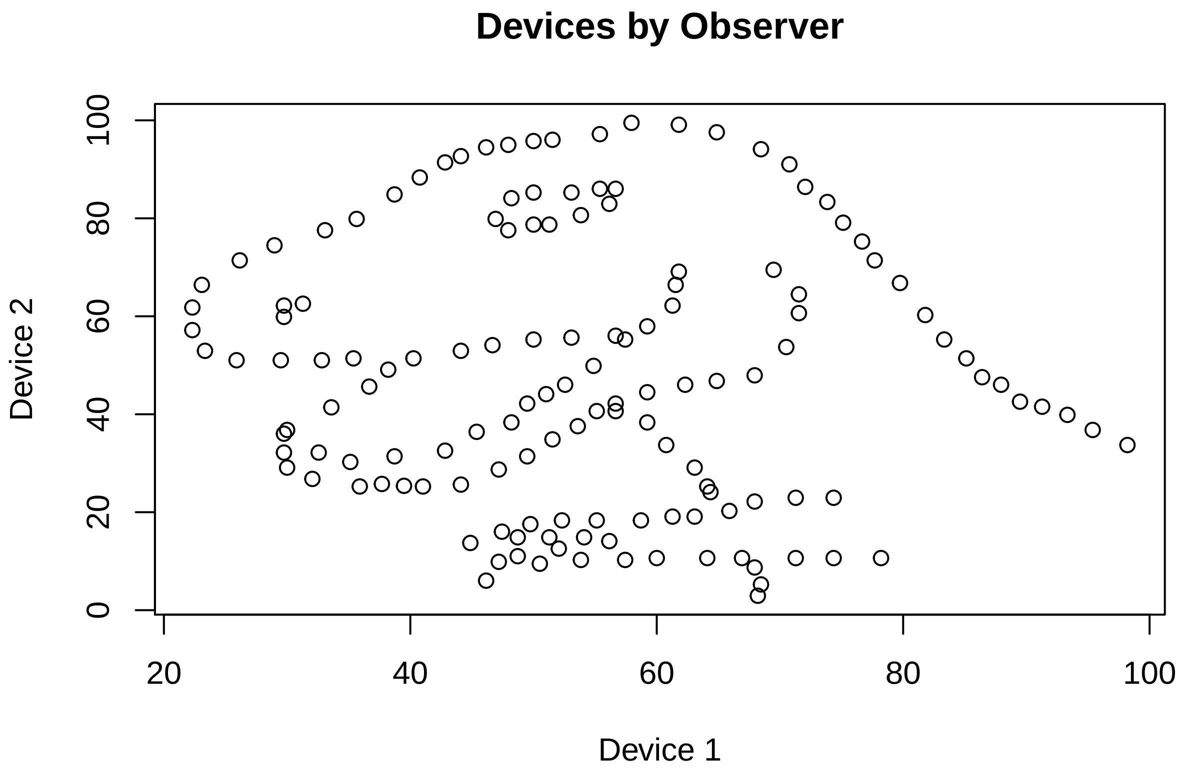
For the proportion of success by row, we get 0's and 1's whereas for the proportion of success by column, we get 0.6 every time. It is using the first probability, 31/100 each time and since the seed is fixed, we get the same outcome.

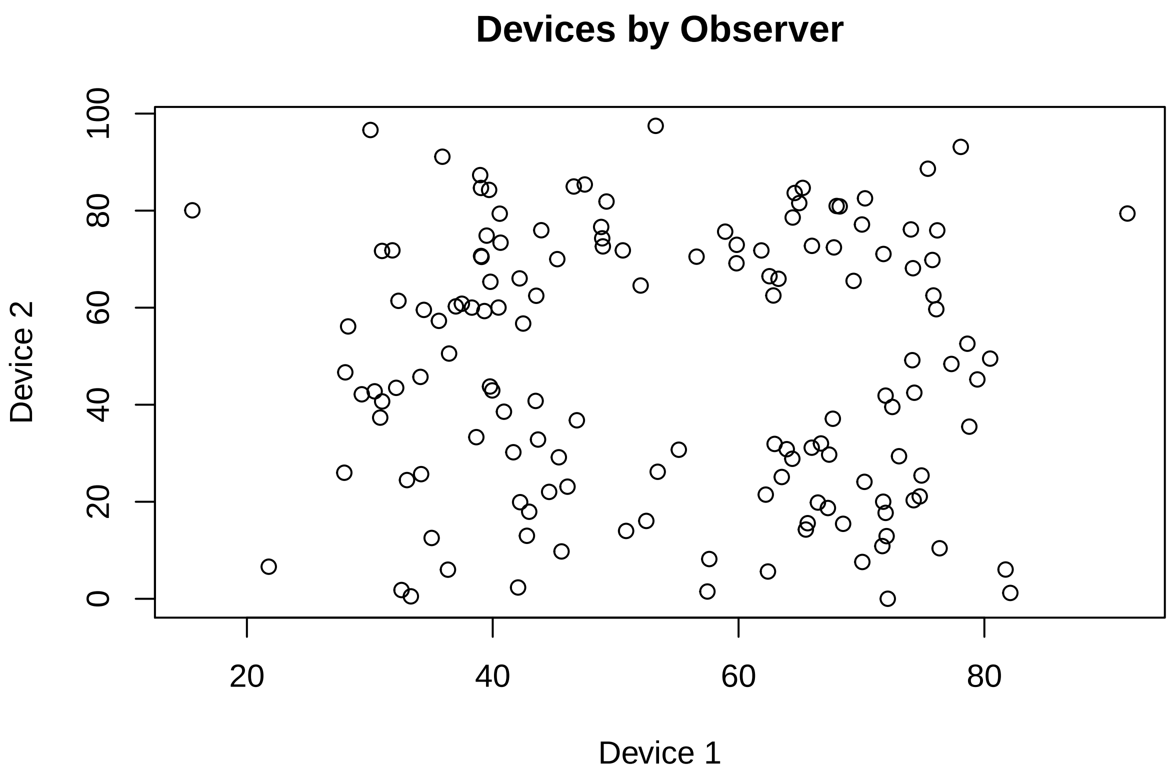
### Problem 4 Observers Data:

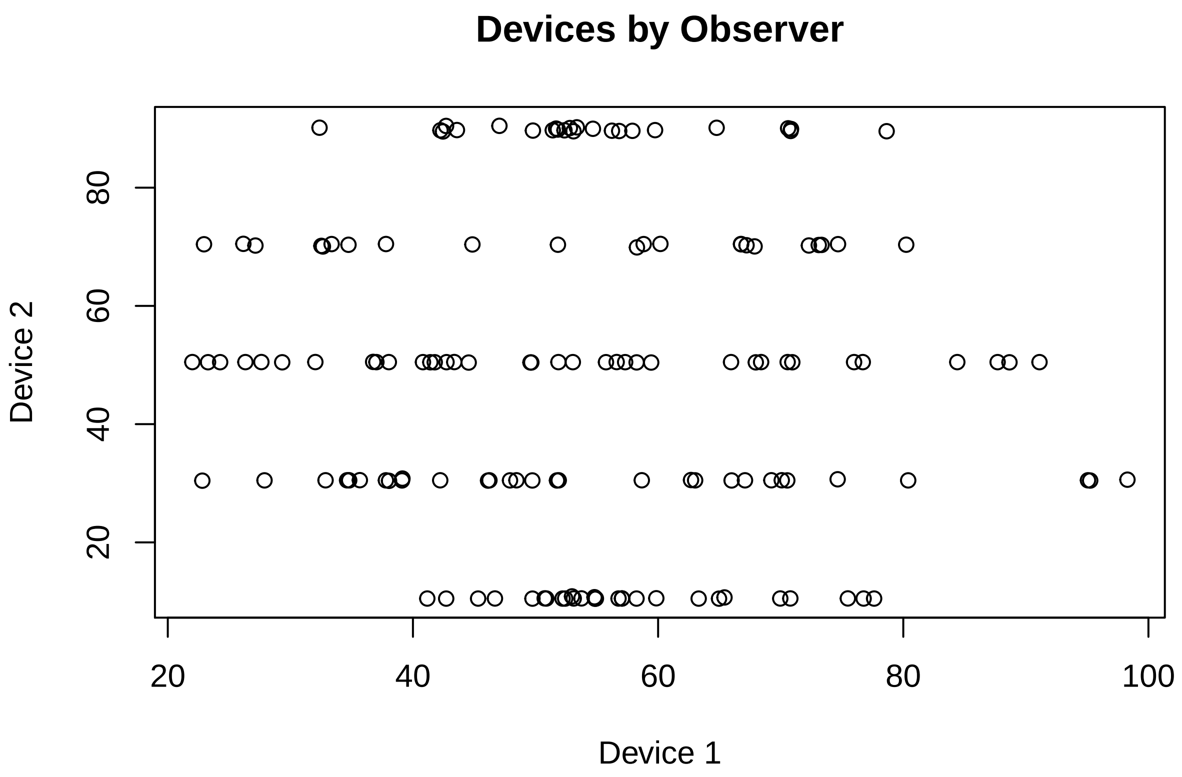
```
#read the data, rename columns and extract the Observers
devices_dat <- readRDS("D:/Downloads/HW3_data.rds")
names(devices_dat) <- c("Observer","x","y")
Observer_cat <- unique(devices_dat$Observer)

#create my plotting function
my_plotfun <- function(Y = devices_dat,index,title = "Devices by Observer",xlab = "Device 1",ylab = "De
{
  X <- subset(Y, Observer == index)
  plot(x = X$x, y = X$y, main = title, xlab = xlab, ylab = ylab)
}

#plot the entire dataset and then use sapply to plot by Observer
my_plotfun(index = Observer_cat, title = "Entire Dataset")
```

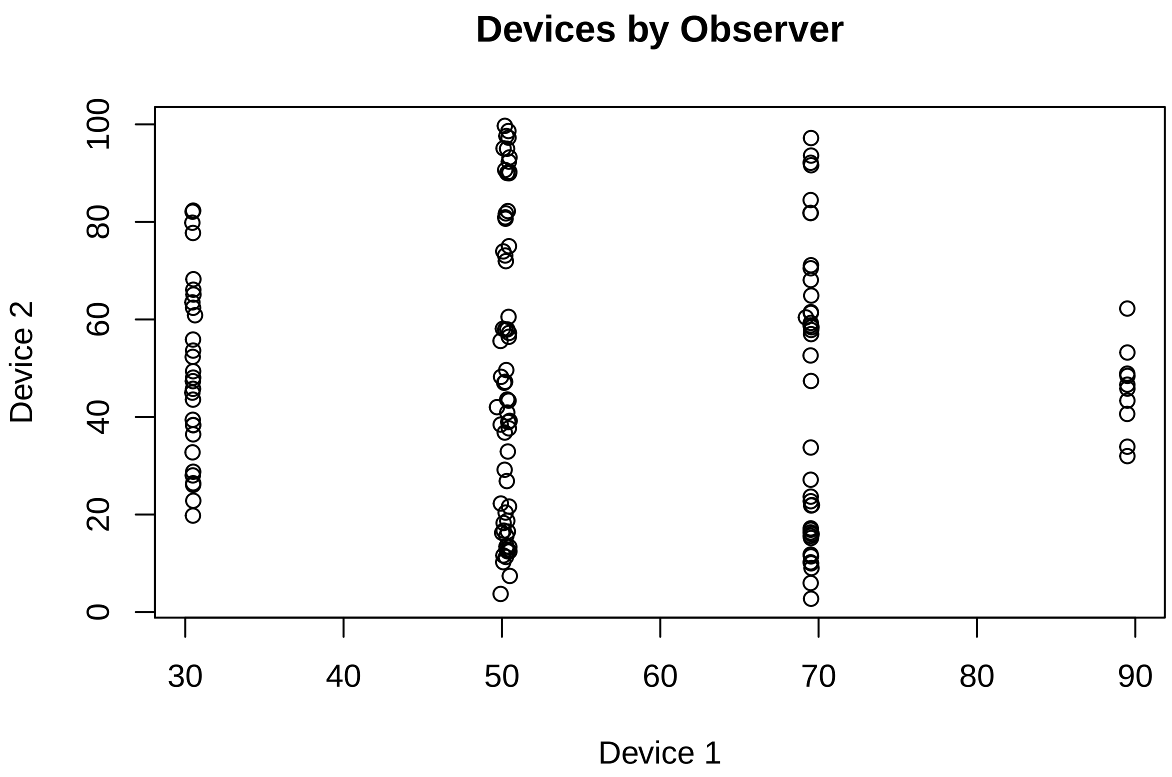


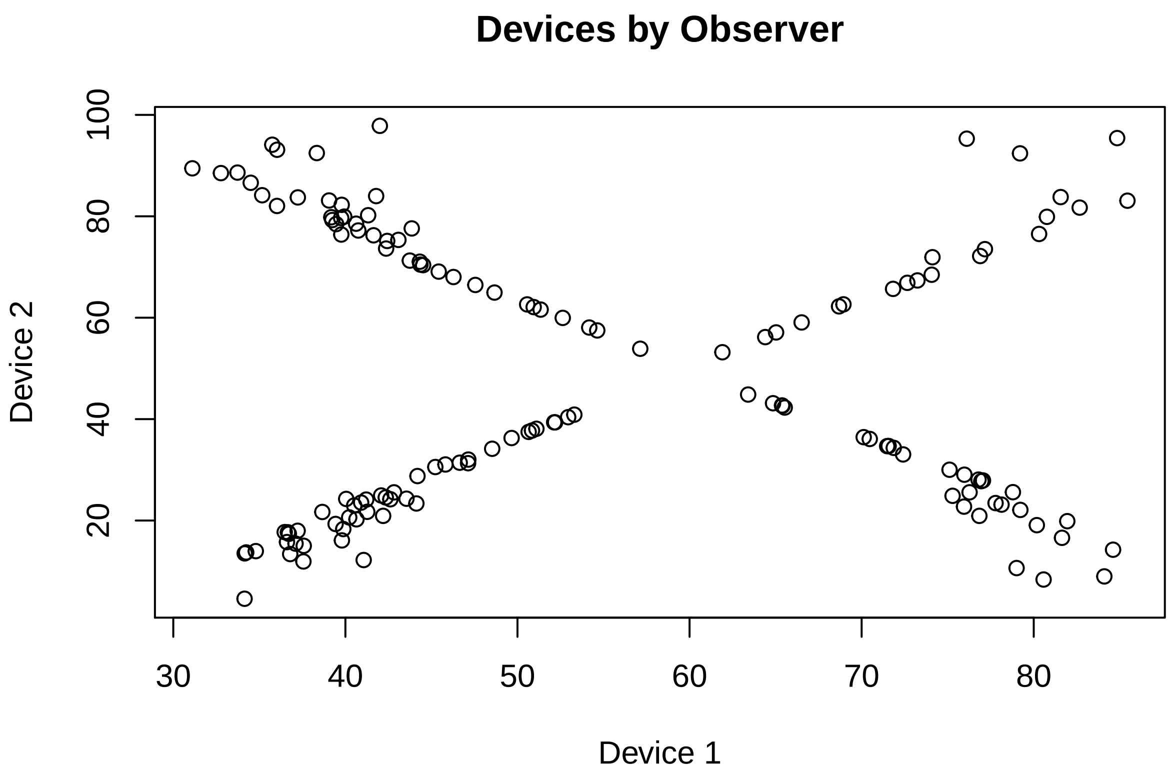
```
#plot the different Observers
multiplot <- sapply(X = Observer_cat,FUN = my_plotfun,Y=devices_dat,xlab="Device 1", ylab = "Device 2",
```

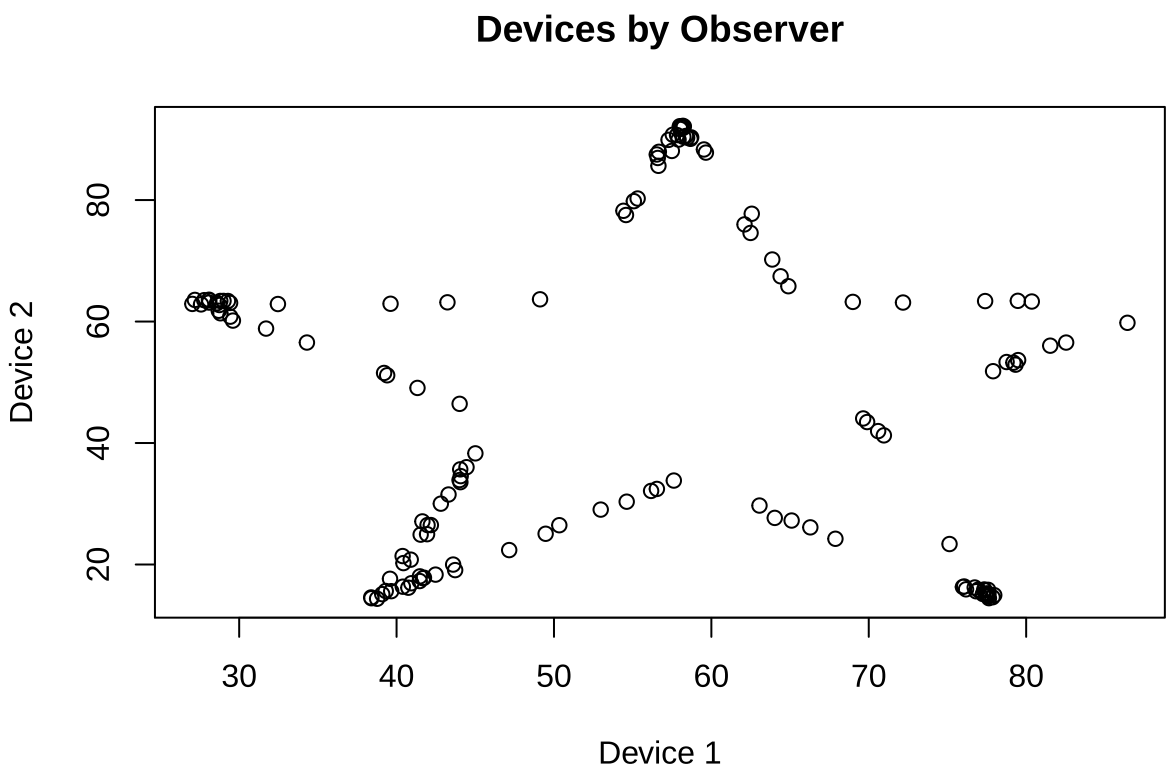

Devices by Observer
Device 2
Device 1
0
20
40
60
80
100
20
40
60
80
100

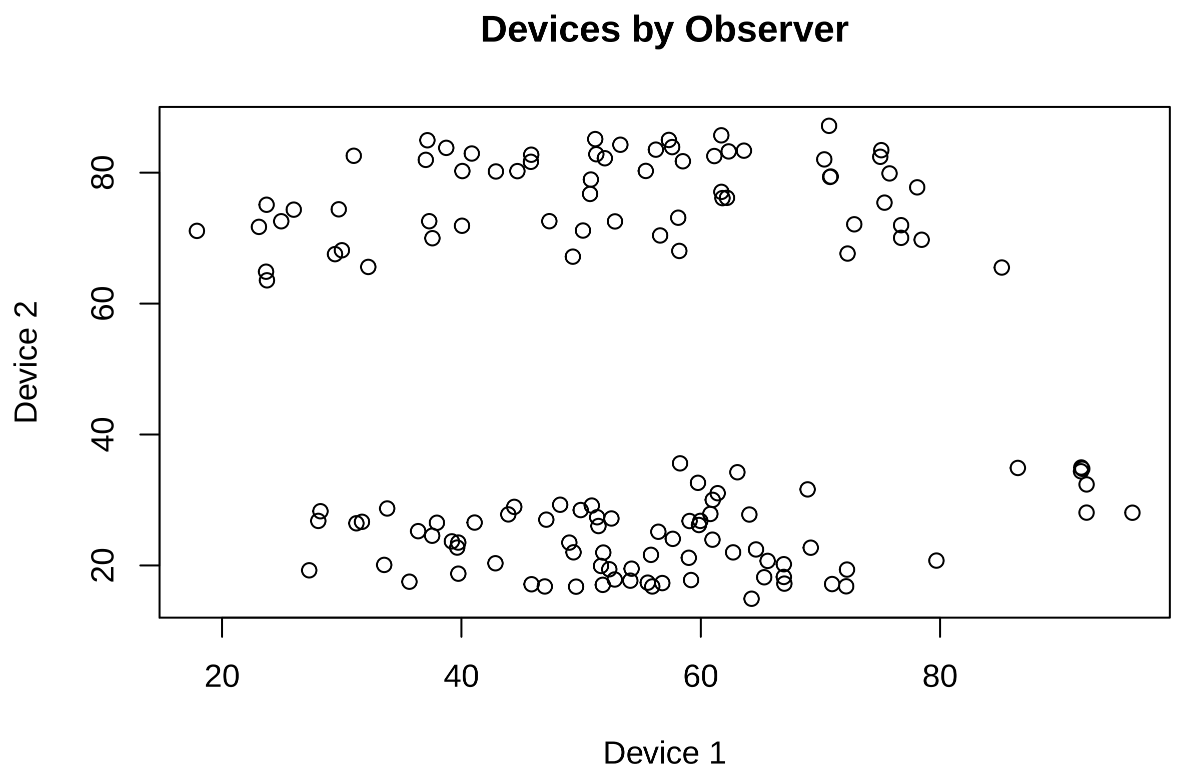

Devices by Observer
Device 2
Device 1
0
20
40
60
80
100
20
40
60
80

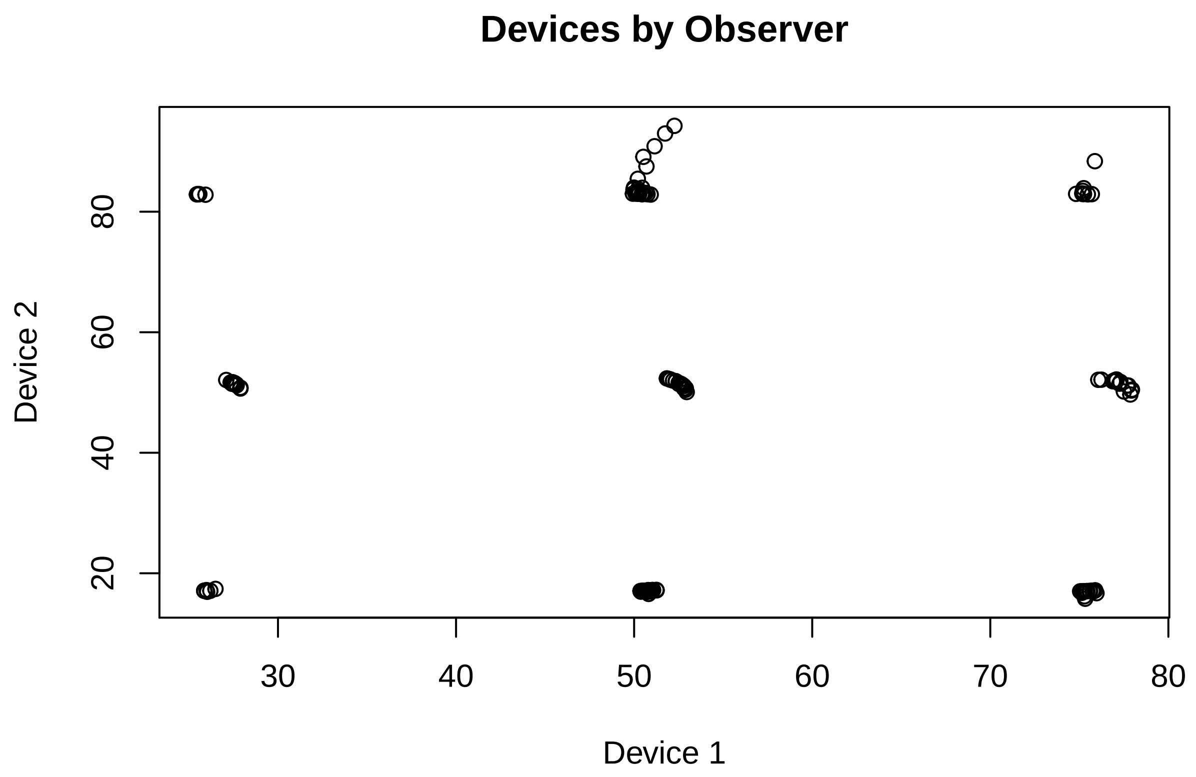

Devices by Observer
Device 2
Device 1
20
40
60
80
100

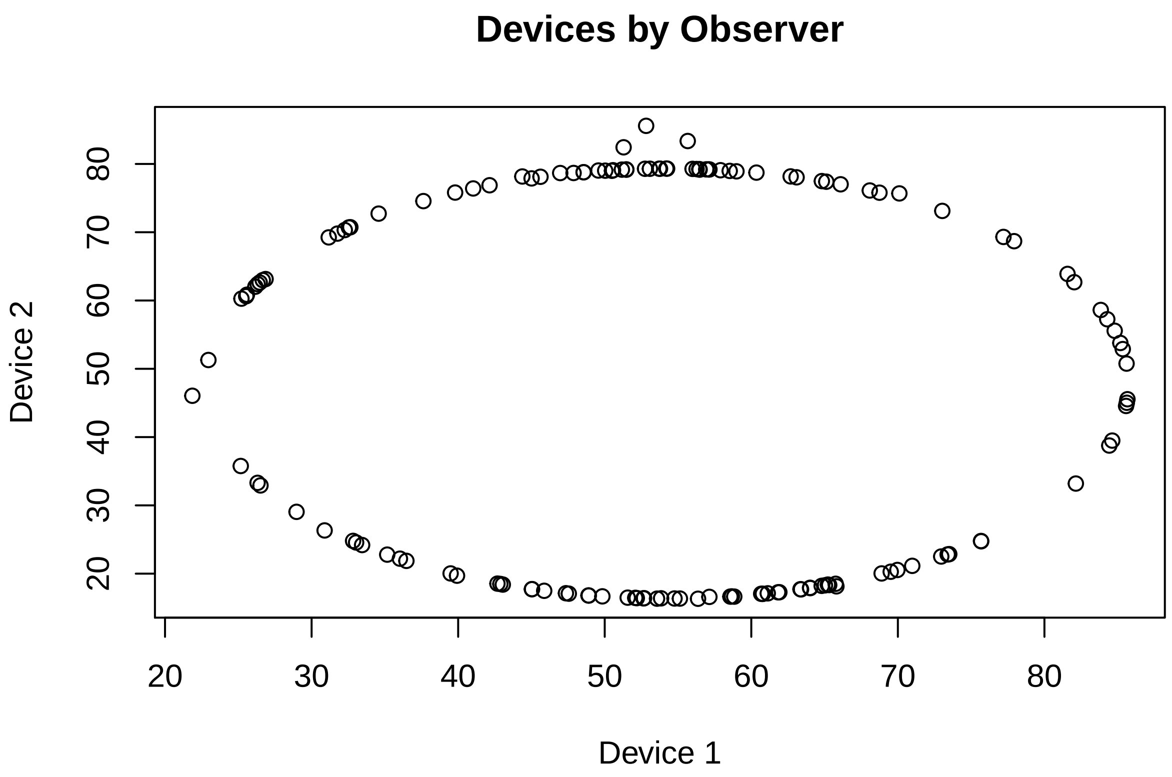

Devices by Observer
Device 2
Device 1
0
20
40
60
80
100
30
40
50
60
70
80
90

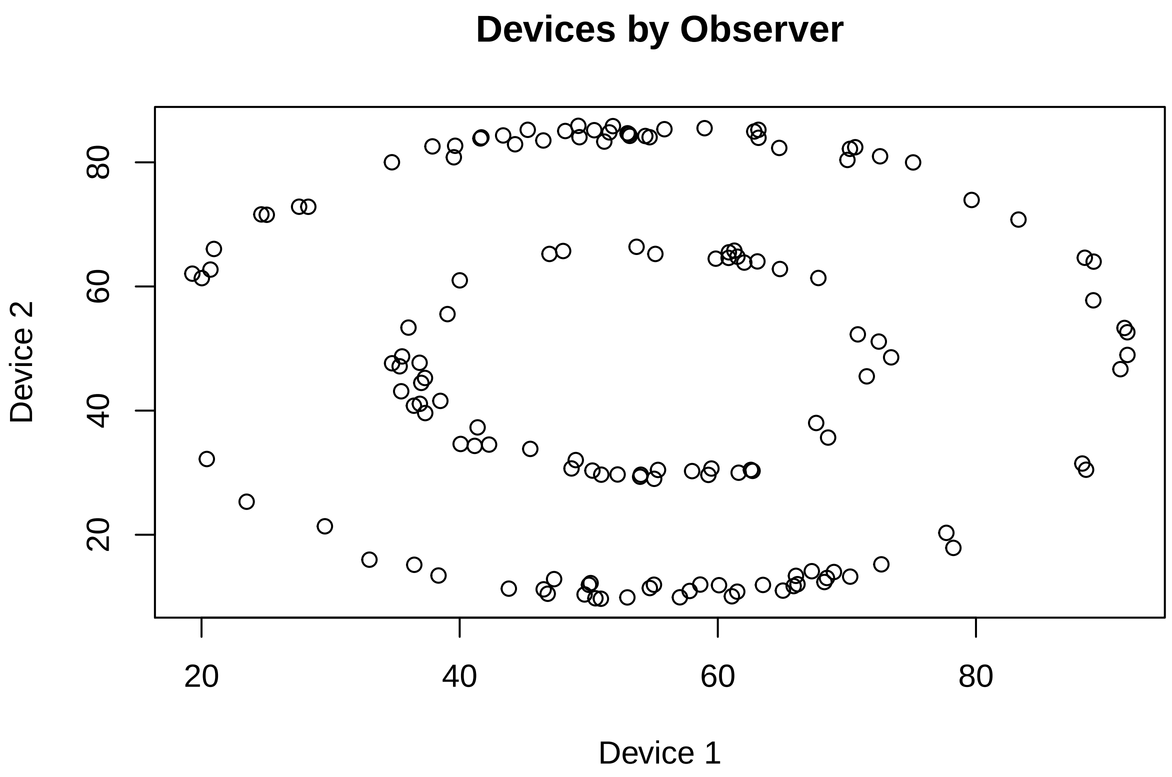

Devices by Observer
Device 2
100
80
60
40
20
30
40
50
60
70
80
Device 1

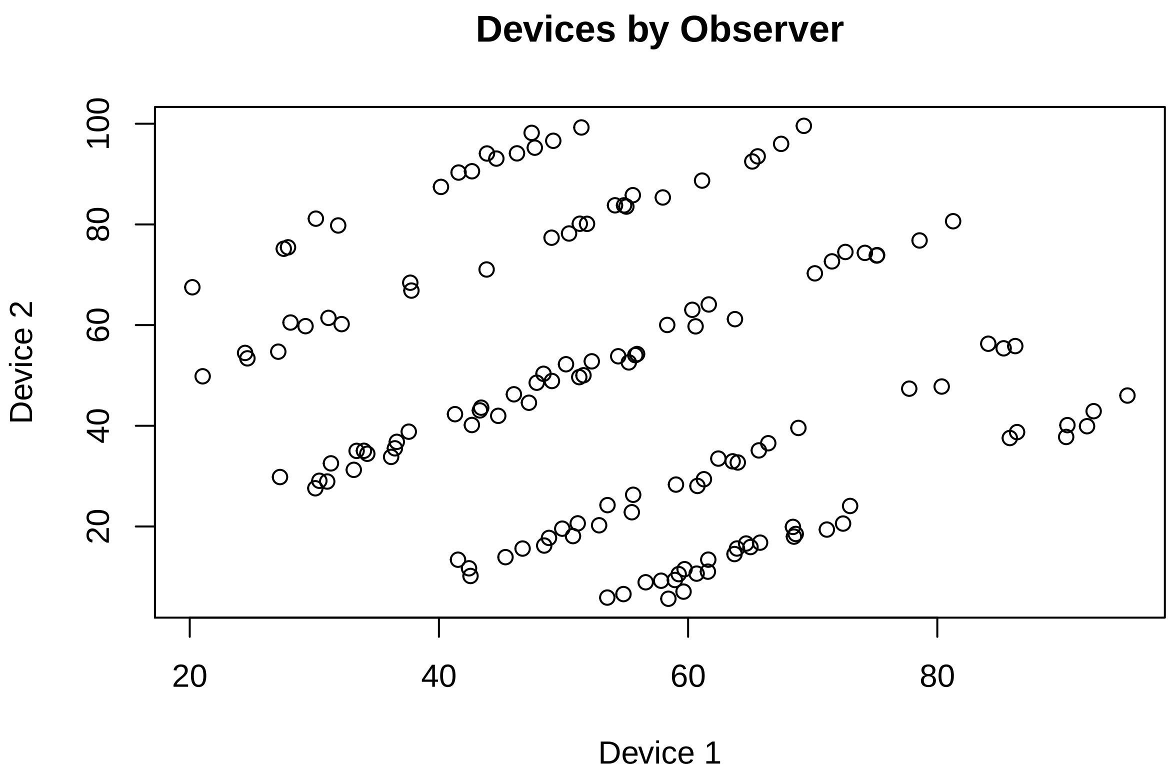

Devices by Observer
Device 2
Device 1
20
40
60
80
30
40
50
60
70
80

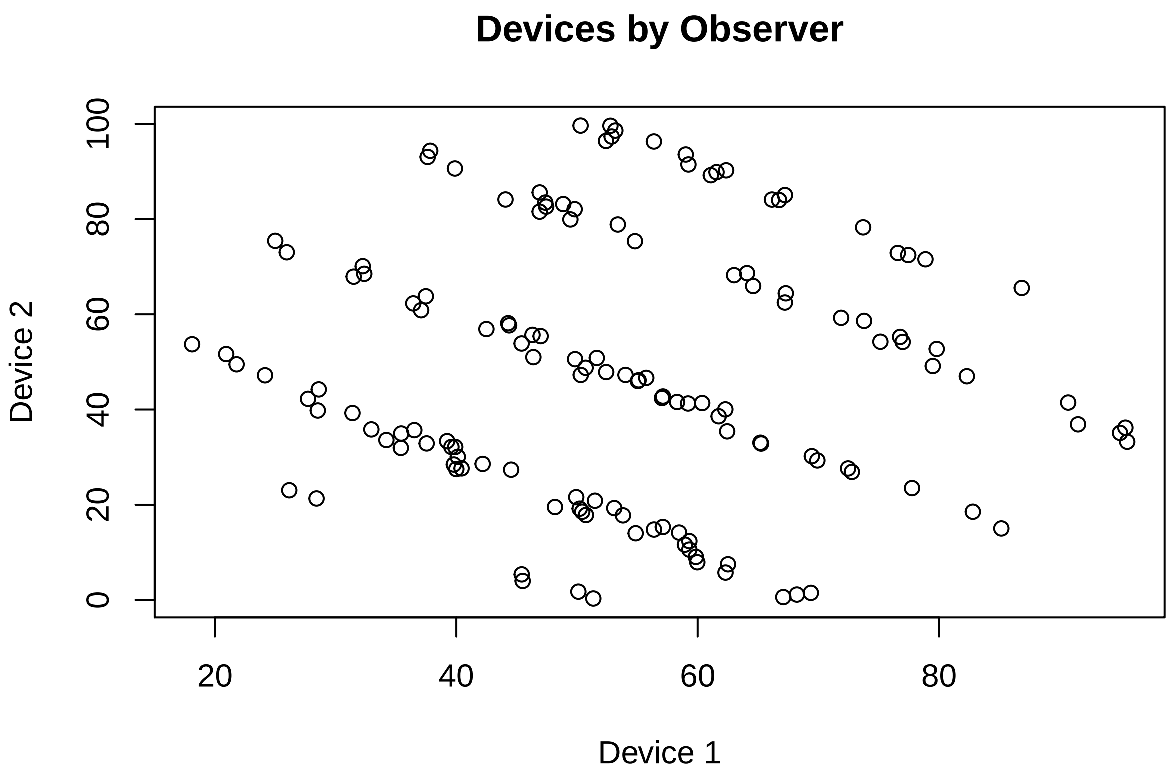

Devices by Observer
Device 2
Device 1
20
40
60
80
20
40
60
80

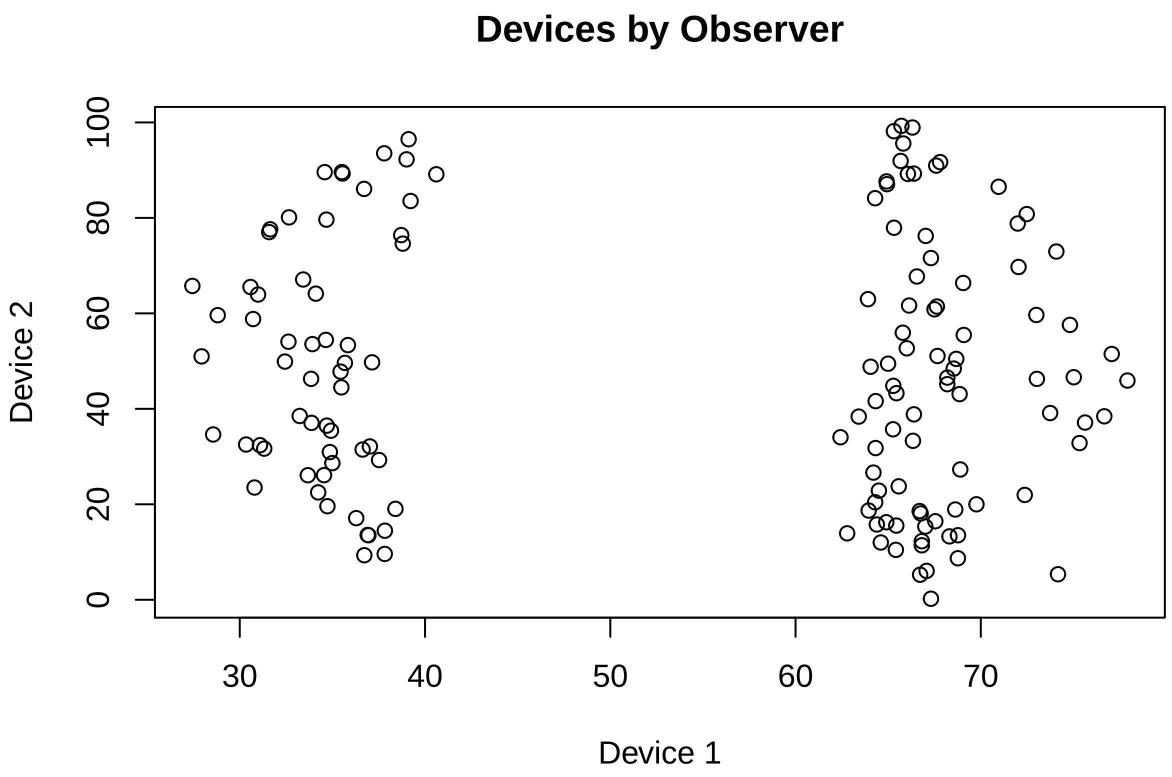

Devices by Observer
Device 2
Device 1
20
40
60
80
30
40
50
60
70
80


Devices by Observer
Device 2
Device 1
20
30
40
50
60
70
80


Devices by Observer
Device 2
Device 1
20
40
60
80
20
40
60
80


Devices by Observer
Device 2
Device 1
20
40
60
80
100


Devices by Observer
Device 2
Device 1
0
20
40
60
80
100
20
40
60
80



```
multiplot
```

```
## [[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: Map

```
#I unzipped the file directly and put the data directly into my project #file. Then I just read in the
library(data.table)
states <- fread("states.sql",skip = 23,sep = "'", sep2 = ",", header = F, select = c(2,4))
cities_ext <- fread("cities.sql",skip = 23,sep = "'", sep2 = ",", header = F, select = c(2,4))
#take out DC from states and make a new dataframe
states_noDC <- states[-8,]
#build a dataframe for number of cities by state
num_cities_bystate <- vector()
for (i in 1:50)
{
  curr_state <- states_noDC[i,2]
  curr_state_long <- states_noDC[i,1]
  curr_cities <- subset(cities_ext, V4 == toString(unlist(curr_state)))
  curr_num_cities <- dim(curr_cities)[1]
  curr_row <- c(tolower(toString(unlist(curr_state_long))),toString(unlist(curr_state)),curr_num_cities
  num_cities_bystate <- rbind(num_cities_bystate,curr_row)
}
num_cities_bystate <- data.frame(num_cities_bystate)
```

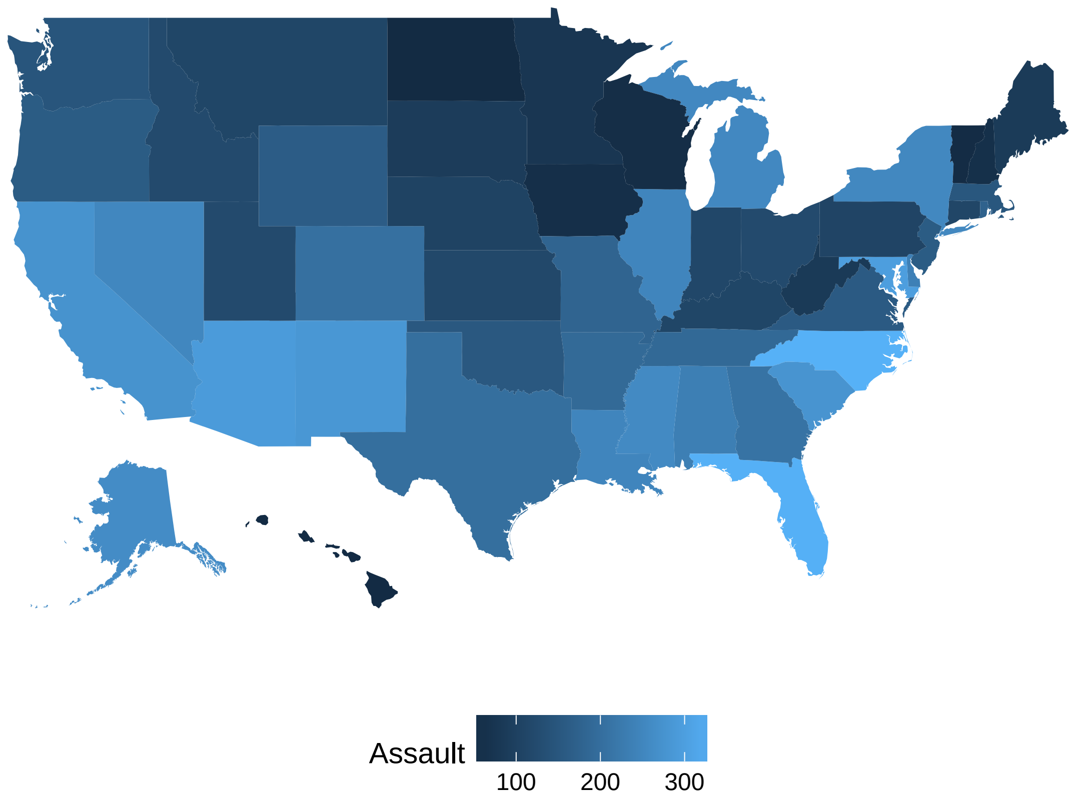
```
## Warning in data.row.names(row.names, rowsi, i): some row.names duplicated:
## 2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,
## --> row.names NOT used
```

```
#name the columns
colnames(num_cities_bystate) <- c("State","State_Short", "Num_Cities")
num_cities_bystate$Num_Cities <- as.numeric(num_cities_bystate$Num_Cities)
#make a table
knitr::kable(num_cities_bystate)
```

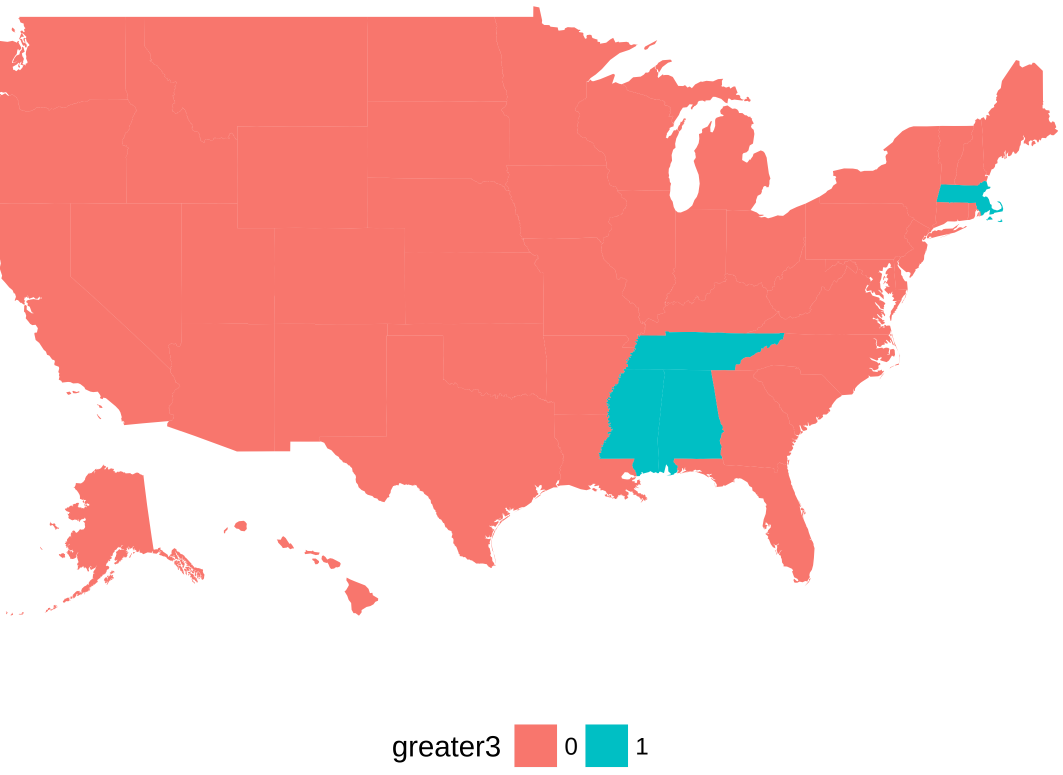
| State | State_Short | Num_Cities |
|---|---|---:|
| alaska | AK | 8 |
| alabama | AL | 32 |
| arkansas | AR | 34 |
| arizona | AZ | 11 |
| california | CA | 2 |
| colorado | CO | 21 |

| State | State_Short | Num_Cities |
|---|---|---:|
| connecticut | CT | 13 |
| delaware | DE | 31 |
| florida | FL | 28 |
| georgia | GA | 35 |
| hawaii | HI | 45 |
| iowa | IA | 46 |
| idaho | ID | 12 |
| illinois | IL | 3 |
| indiana | IN | 38 |
| kansas | KS | 36 |
| kentucky | KY | 41 |
| louisiana | LA | 25 |
| massachusetts | MA | 27 |
| maryland | MD | 22 |
| maine | ME | 24 |
| michigan | MI | 44 |
| minnesota | MN | 42 |
| missouri | MO | 47 |
| mississippi | MS | 23 |
| montana | MT | 16 |
| north carolina | NC | 40 |
| north dakota | ND | 18 |
| nebraska | NE | 29 |
| new hampshire | NH | 10 |
| new jersey | NJ | 32 |
| new mexico | NM | 15 |
| nevada | NV | 48 |
| new york | NY | 5 |
| ohio | OH | 1 |
| oklahoma | OK | 33 |
| oregon | OR | 20 |
| pennsylvania | PA | 7 |
| rhode island | RI | 37 |
| south carolina | SC | 19 |
| south dakota | SD | 17 |
| tennessee | TN | 30 |
| texas | TX | 4 |
| utah | UT | 9 |
| virginia | VA | 43 |
| vermont | VT | 14 |
| washington | WA | 26 |
| wisconsin | WI | 39 |
| west virginia | WV | 39 |
| wyoming | WY | 6 |

```
letter_count <- data.frame(matrix(nrow = 50, ncol = 26))
getCount <- function(letter,state_name)
{
  #make the letter and state the same case
  letter <- toupper(letter)
  state_name <- toupper(state_name)
  #split the state into letters, set the counter
  temp <- unlist(strsplit(state_name,split=""))
  count <- 0
  #do the comparison and counting
  for (i in 1:length(temp))
  {
    if (temp[i] == letter)
    {
      count <- count + 1
    }
  }
  return(count)
}
#use a for loop and sapply to fill in letters_count
for (i in 1:50)
{
  curr_state <- toString(unlist(states_noDC[i,1]))
  letter_count[i,] <- sapply(LETTERS,FUN = getCount,state_name = curr_state)
}
#add column and rownames to the letter_count data frame
names(letter_count) <- LETTERS
rownames(letter_count) <- as.vector(num_cities_bystate[,1])

#greater3 is a vector which indicates whether a state has more than 3 #occurrences for any letter
greater3 <- rep(0,50)
for (i in 1:50)
{
  curr_state <- toString(unlist(states_noDC[i,1]))
  curr_letter_count <- letter_count[i,]
  for (j in 1:26)
  {
    #1 indicates a state has more than 3 occurrences for any letter
    if (curr_letter_count[j] > 3)
    {
      greater3[i] <- 1
      break
    }
  }
}

states_greater3 <- data.frame(cbind(as.vector(num_cities_bystate$State),greater3))

library(ggplot2)
library(remotes)
#install_version("fiftystater", "1.0.1")
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(crimes, aes(map_id = state)) +
      # map points to the fifty_states shape data
      geom_map(aes(fill = Assault), 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
```



```
p_numcities <- ggplot(num_cities_bystate, aes(map_id = State)) +
  # map points to the fifty_states shape data
  geom_map(aes(fill = Num_Cities), 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())

names(states_greater3)[1] <- "State"

p_greater3 <- ggplot(states_greater3, aes(map_id = State)) +
  # map points to the fifty_states shape data
  geom_map(aes(fill = greater3), 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_greater3
```



### Problem 2: Bootstrapping

```
#Code to get "http://www2.isye.gatech.edu/~jeffwu/wuhamadabook/data/Sensory.dat"
sensory_url <- "http://www2.isye.gatech.edu/~jeffwu/wuhamadabook/data/Sensory.dat"
sensory_data_raw <- fread(sensory_url,skip = 2,data.table = FALSE,header = FALSE, fill = TRUE)
#create an items_column which will be on the left side of the dataframe
items_column <- vector(length = 0)
for (i in 1:10)
{
  items_column <- append(items_column, rep(i,3))
}
#the item numbers are woven into the first column so we replace them with NA's #using the two lines of
first_col_clean <- sensory_data_raw$V1
first_col_clean[seq(1,30,3)] <- NA
sensory_data_tidy_baseR <- cbind(first_col_clean,sensory_data_raw[,2:6])
#we store the rows to fix in rows_to_fix
rows_to_fix <- seq(1,30,3)
```

```
#the for loop below shifts the rows to fix one entry to the left and the last entry is NA, ensuring tha
for (i in 1:length(rows_to_fix))
{
  curr_row <- rows_to_fix[i]
  cleaned_row <- c(sensory_data_tidy_baseR[curr_row,2:6],NA)
  sensory_data_tidy_baseR[curr_row,] <- cleaned_row
}
#rename the first column
names(sensory_data_tidy_baseR)[1] <- "V1"
#drop the last column since it is all NA
sensory_data_tidy_baseR <- sensory_data_tidy_baseR[,1:5]
#bind the items_column and rename the columns
sensory_data_tidy_baseR <- cbind(items_column,sensory_data_tidy_baseR)
names(sensory_data_tidy_baseR)<- c("Item","1","2","3","4","5")
#the current form of sensory_data_tidy_baseR is what we will use for the tidyverse version so we make a
sensory_data_tidy_tidyverse <- copy(sensory_data_tidy_baseR)
#operator_column repeats the operator values in a sequence 1,...,5 30 times which will be one of the co
operator_column <- rep(seq(1,5),30)
#the measurement column turns the  data in the rows into a vector
measurement_column <- vector(length = 0)
for (i in 1:dim(sensory_data_tidy_baseR)[1])
{
  #unlist is used to turn the rows into vectors
  measurement_column <- append(measurement_column,unlist(sensory_data_tidy_baseR[i,2:6],use.names = FALS
}
#the final items column should have 150 entries
items_column_final <- vector(length = 0)
for (i in 1:10)
{
  items_column_final <- append(items_column_final, rep(i,15))
}
sensory_data_tidy_baseR <- data.frame(cbind(items_column_final,operator_column,measurement_column))
names(sensory_data_tidy_baseR) <-c("Item", "Operator","Data")

#beta0_vector stores the beta0s, beta1_vector stores the beta1s
beta0_vector <- vector(length = 100)
beta1_vector <- vector(length = 100)
system.time(for (i in 1:100)
{
  i <- sample(x = 1:150,size = 150,replace = TRUE)
  boot_sensory_data <- sensory_data_tidy_baseR[i,]
  lm_fit <- lm(Data ~ Operator, data = boot_sensory_data)
  beta0_vector <- append(beta0_vector,summary(lm_fit)$coef[1,1])
  beta1_vector <- append(beta1_vector,summary(lm_fit)$coef[1,2])
})
```

```
##    user  system elapsed
##    0.18    0.01    0.24
```

```
beta0_mean <- mean(beta0_vector)
beta1_mean <- mean(beta1_vector)

library(foreach)
library(parallel)
```

```r
#make the cluster, set the number of cores
cores <- max(1,detectCores()-1)
cl <- makeCluster(cores)

system.time(foreach(i=1:100) %do%
{
  i <- sample(x = 1:150,size = 150,replace = TRUE)
  boot_sensory_data <- sensory_data_tidy_baseR[i,]
  lm_fit <- lm(Data ~ Operator, data = boot_sensory_data)
})
```

```
##    user  system elapsed
##    0.30    0.00    0.31
```

```r
stopCluster(cl)

beta <- c(beta0_mean,beta1_mean)
print(beta)
```

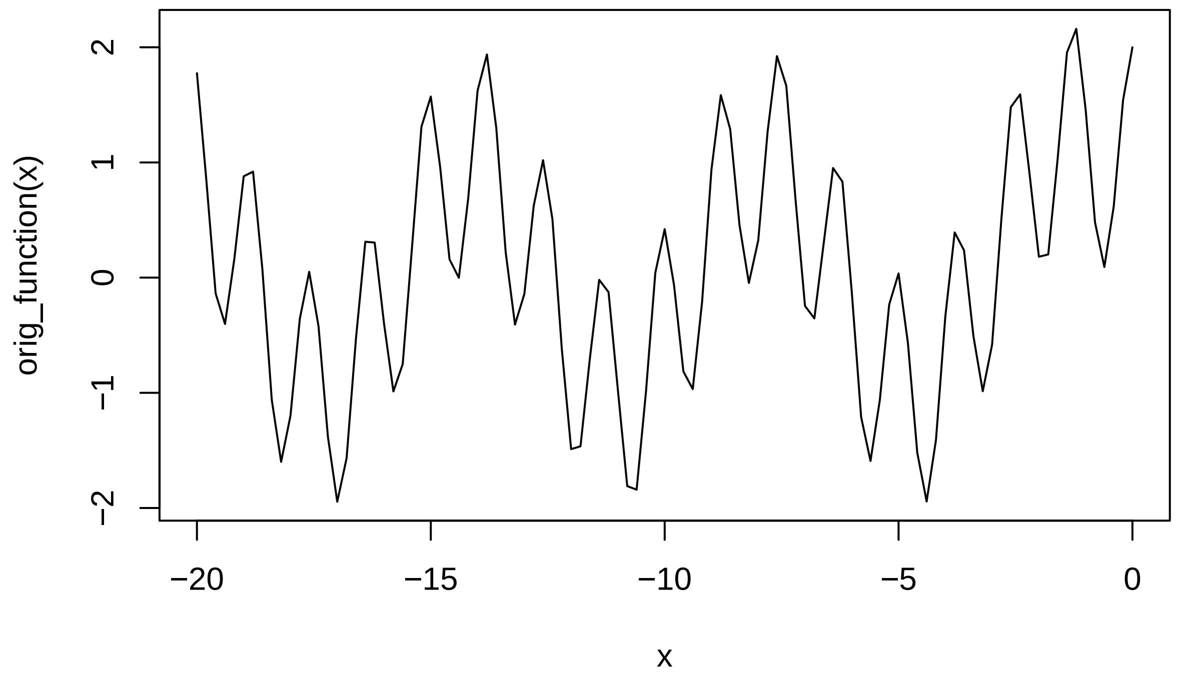
```
## [1] 2.3976351 0.2062974
```

```r
time <- rbind(0.11,0.10)
method <- rbind("Series","Parallel")
method.time <- cbind(method,time)
method.time <- as.data.frame(method.time)
knitr::kable(method.time, col.names = c("Method","Time"))
```

| Method | Time |
|---|---|
| Series | 0.11 |
| Parallel | 0.1 |

The supplied code in stack exchange failed to generate different samples on each run; for some reason it remained stuck at the same sample thus giving the same results. It is possible to do the bootstraps in parallel because the bootstrap sampling is done independently of other samples. Based on the difference in elapsed time, the small number of bootstrap samples, and the small data size I would say that it makes little difference computing in series vs parallel for this problem. The mean of the estimated $\beta_0$ was 2.4194 and the mean of the estimated $\beta_1$ was 0.2046.

```r
grid <- seq(-20,0,length.out = 100)
#orig_function is the original function
orig_function <- function(x)
  {
    orig <- 3^x-sin(x)+cos(5*x)
    return(orig)
}
#plot the original function from -20 to 0
curve(orig_function,from = -20,to = 0)
```



```
#deriv_function is the derivative of deriv_function
deriv_function <- function(x)
  {
    deriv <- 3^x*log(3) - cos(x) - 5*sin(5*x)
    return(deriv)
  }

#The tolerance is 10^-6
tolerance <- 1e-6

#newton_method applies the newton method using the original function, the #derivative of the original f
newton_method <- function(tolerance,orig_function,deriv_function,start_x)
  {
    #before the while loop, we set x_vector which stores x_n for iteration n
    x_vector <- start_x
    #curr_x is the current approximation to the root
    curr_x <- start_x
    #iter is the current iteration and iter_vector stores how many iterations     #there have been
    iter <- 1
    iter_vector <- 1

    #since we are looking for the root, we run a while loop as long as the        #original function ev
    while(abs(orig_function(curr_x)) > tolerance)
      {
        #set the new current x using newton's method
        curr_x <- curr_x-orig_function(curr_x)/deriv_function(curr_x)
```

```r
        #if the original function at the current approximation is less than           #the tolerance, w
        if (abs(orig_function(curr_x)) <= tolerance)
          {
            x_vector <- append(x_vector,curr_x)
            iter <- iter + 1
            iter_vector <- append(iter_vector,iter)
            solution <- curr_x
            break
          }
        x_vector <- append(x_vector,curr_x)
        iter <- iter + 1
        iter_vector <- append(iter_vector,iter)
  }

    #return the vector of iterations, approximations to the root, the original     #function evaluated 
    return(ls = list(iter_vector = iter_vector,x_vector =                             x_vector,orig_functio
}

#use sapply on the grid
# system.time(roots <- sapply(grid,newton_method,tolerance = tol, orig_function = orig_function, deriv_

#use parSapply on the grid
library(parallel)
# cores <- max(1,detectCores()-1)
# cl <- makeCluster(cores)
# parSapply(cl,grid,newton_method,tolerance = tol, orig_function = orig_function, deriv_function = deri
# stopCluster(cl)
```

There seems to be an infinite number of roots so I just chose the interval [-20,0] to apply the Newton method on. Unfortunately, it was taking my computer way too long to get results using sapply and parSapply. However, I suspect that in this problem, parSapply may be more useful since we are dealing with a large grid and so parallelism would be more appropriate to use.