HW4_zhiyuan

Zhiyuan Du

10/6/2020

Problem 2

The algorithm is called Gradient Descent. We can write a fcunction about based on a while loop to approximate the parameters. Then we compare the results with lm(h~0+X).

```
# define start values
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)

# write a function named 'GD' with step size and tolerance input
GD <- function(theta, alpha, tolerance) {

  # define the values
  theta0_i1 <- 0
  theta1_i1 <- 0
  theta0_i <- 1
  theta1_i <- 2
  n <- 0

  # while loop stop stop when less than the tolerance
  while (abs(theta0_i - theta0_i1) > tolerance && abs(theta1_i - theta1_i1) > tolerance) {
    theta0_i1 <- theta0_i
    theta1_i1 <- theta1_i
    theta0_i <- theta0_i1 - alpha * (1/100) * sum(X %*% matrix(c(theta0_i1, theta1_i1),
      nrow = 2) - h)
    theta1_i <- theta1_i1 - alpha * (1/100) * (t(X %*% matrix(c(theta0_i1, theta1_i1),
      nrow = 2) - h) %*% X[, 2])
    n <- n + 1
  }

  # return the results
  results <- cbind(theta0_i, theta1_i, n)
  colnames(results) <- c("theta_0", "theta_1", "iteration")
  return(results)

}

GD(theta, 0.01, 1e-07)
```

```
##        theta_0  theta_1 iteration
## [1,] 0.9699014 2.001516      2145
```

```
lm(h ~ 0 + X)
```

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

Here, we used tolerance step size as 0.01 and tolerance as 1e-6, the results we got about theta0 and theta1 are close to the codfficients we got by lm(h~0+X), which means our approximation by Gradient Descent was successful.

Problem 3

a. In this question, we are doing the Gradient Descent for 10000 different combinations of beta0 and beta1. The true beta0 and beta1 is 1 and 2, here, we get 100 start values for beta0 in (0,2) and (1,3). Then we do the Gradient Descent for the 10000 different combinations of beta0 and beta1.

```
theta0_start <- seq(0, 2, length.out = 100)
theta1_start <- seq(1, 3, length.out = 100)
theta_start <- rbind(rep(theta0_start, 1, each = 100), rep(theta1_start, 100))

cores <- detectCores() - 1
cluster <- makeCluster(cores, type = "SOCK")
registerDoSNOW(cluster)
clusterExport(cluster, c("X", "h"))
system.time({
  final_results <- foreach(n = 1:10000, .combine = rbind) %dopar% GD(theta_start[,
    n], alpha = 1e-07, tolerance = 1e-09)
})
```

```
##     user   system  elapsed
##     8.74     1.03 19904.33
```

```
stopCluster(cluster)
kable(final_results[1:10, ], caption = "first 10 values")
```

Table 1: first 10 values

theta_0	theta_1	iteration
0.9994725	1.997945	362204
0.9994725	1.997945	362204
0.9994725	1.997945	362204
0.9994725	1.997945	362204
0.9994725	1.997945	362204

theta_0	theta_1	iteration
0.9994725	1.997945	362204
0.9994725	1.997945	362204
0.9994725	1.997945	362204
0.9994725	1.997945	362204
0.9994725	1.997945	362204

```
min(final_results[, 3])
```

```
## [1] 362204
```

```
max(final_results[, 3])
```

```
## [1] 362204
```

```
mean(final_results[, 1])
```

```
## [1] 0.9994725
```

```
mean(final_results[, 2])
```

```
## [1] 1.997945
```

```
sd(final_results[, 1])
```

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

```
sd(final_results[, 2])
```

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

b. The stop rulr for this problem is not good which takes too much time. We can try try different tolerance. However, it is tough to make the right decision about the stop rule for that large tolerance may not make the loop converge to the true value, the small tolerance takes too much time.

c. This algorithm is not ideal based on the situation that we have no idea about the range of the starting value, wrong starting values may not converge to the true value. Also, the stepping size and the tolerance are hard to decide too.

Problem 4

John Cook suggests that instead of solving for the inverse of $(X^TX)^{-1}$ we use the solve function to find $\hat{\beta}$ by solving the following system $(X^TX)\hat{\beta} = X^T\underline{y}$. This will save unnecessary computation of $(X^TX)^{-1}$ since the goal is to really find $\hat{\beta}$.

Problem 5

a.

```
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
```

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

```
## 112347224 bytes
```

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

```
## 1816357208 bytes
```

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

```
##    user  system elapsed
##  899.33   14.02  919.98
```

```
cat("the size of A is", object.size(A))
```

```
## the size of A is 112347224
```

```
cat("the size of B is", object.size(B))
```

```
## the size of B is 1816357208
```

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

```
##    user  system elapsed
##  906.91   11.03  921.86
```

From the results, the size of A is 112347224 bytes(107.1 Mb) and the size of B is 1816357208 bytes(1.7 Gb). The time used on computation is 881.27 seconds.

b. The most time consuming part in the compution is A times the inverse of B. I would do this firstly, and then multiply it with (q-r), finally sum it with p. In R, we can convert the matrix into sparse matrix which will save the time on computing.

c. Let's do this and calculate the time used.

```
time <- system.time({
  Anew <- as(A, "sparseMatrix")
  Bnew <- as(B, "sparseMatrix")
  s <- q - r
  D <- solve(Bnew, s)
  y <- p + Anew %*% D
})

time
```

```
##    user  system elapsed
##    0.23    0.78    1.09
```

the time after the simplifications is only 0.52 seconds.

Problem 6

a. Let's denote the sample space of this event as {s,f,s,s,f,f...}, where "s" is success and "f" is failing, then the proportion of success is the number of "s" over the number of samples in sample space.

```
data1 <- c(1, 1, 1, 1, 1, 0, 0, 0, 0, 0)
pro_success <- function(data) {
  sum(data != 0)/length(data)
}
pro_success(data1)
```

```
## [1] 0.5
```

b. Just type the data.

```
set.seed(12345)
P6b_data <- matrix(rbinom(10, 1, prob = (31:40)/100), nrow = 10, ncol = 10, byrow = FALSE)
```

c. Run the code.

```
apply(P6b_data, 1, pro_success)
```

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

```
apply(P6b_data, 2, pro_success)
```

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

From the results, the propoertions by row are all 1 and 0 and the proportion by column are all same as 0.6. So the matrix is not random at all.

d. Fix the code to meke it random.

```
set.seed(12345)
p6d <- function(p) {
  rbinom(10, 1, p)
}
# use the apply function to fix
P6d_data <- sapply(seq(0.31, 0.4, by = 0.01), p6d)
apply(P6d_data, 1, pro_success)
```

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

```
apply(P6d_data, 2, pro_success)
```

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

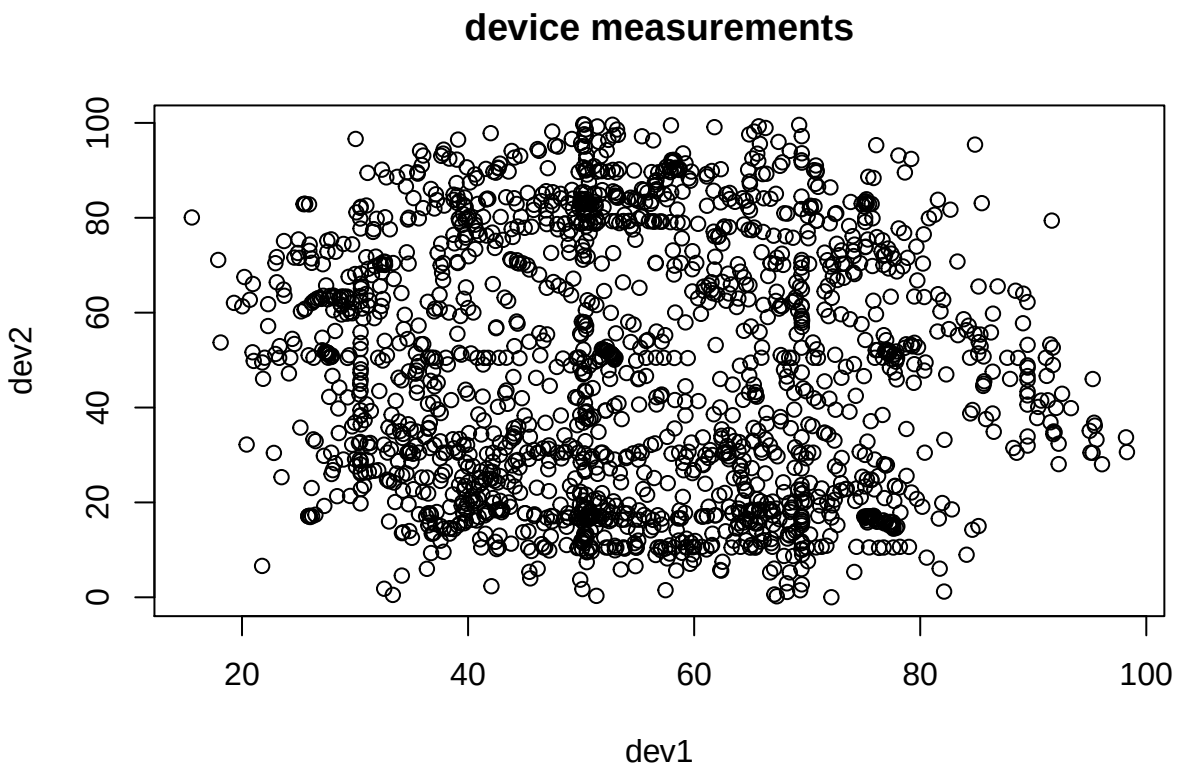
From the results, we used sapply function to create a function by which a random matrix could be created. The proportions of success of the matrix we created are different by both row and column.

Problem 7

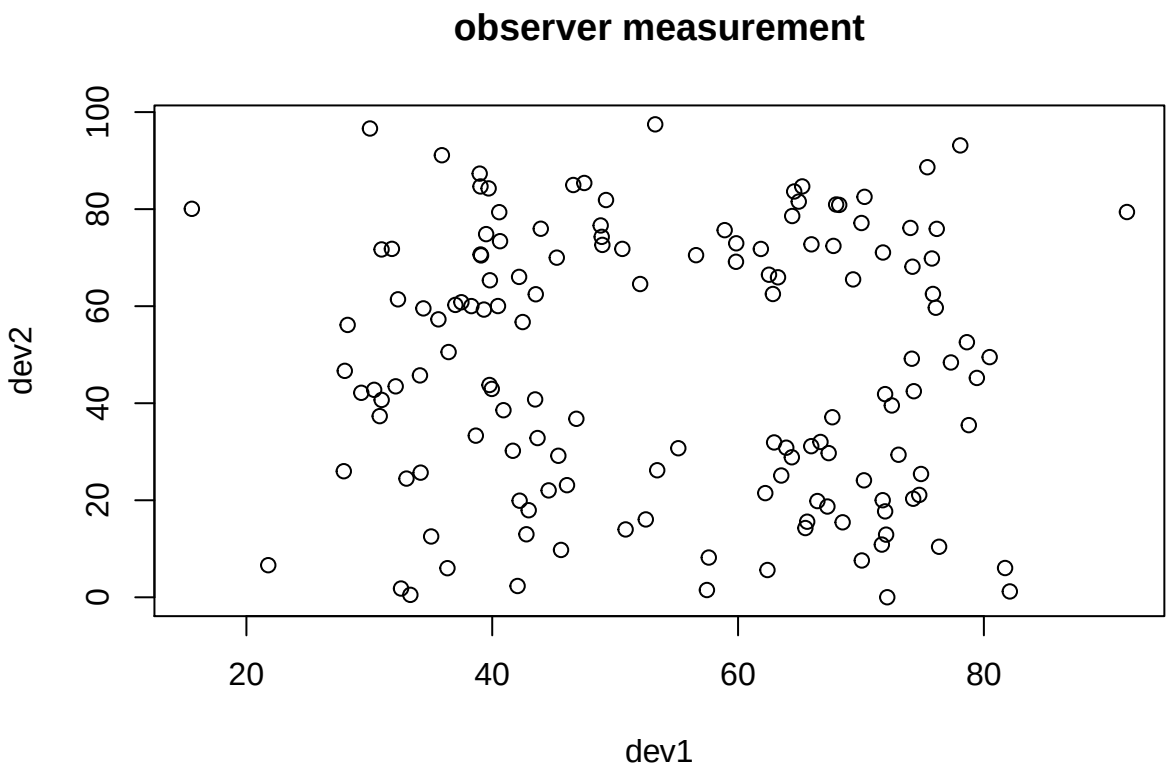
Firstly, let's import the data.

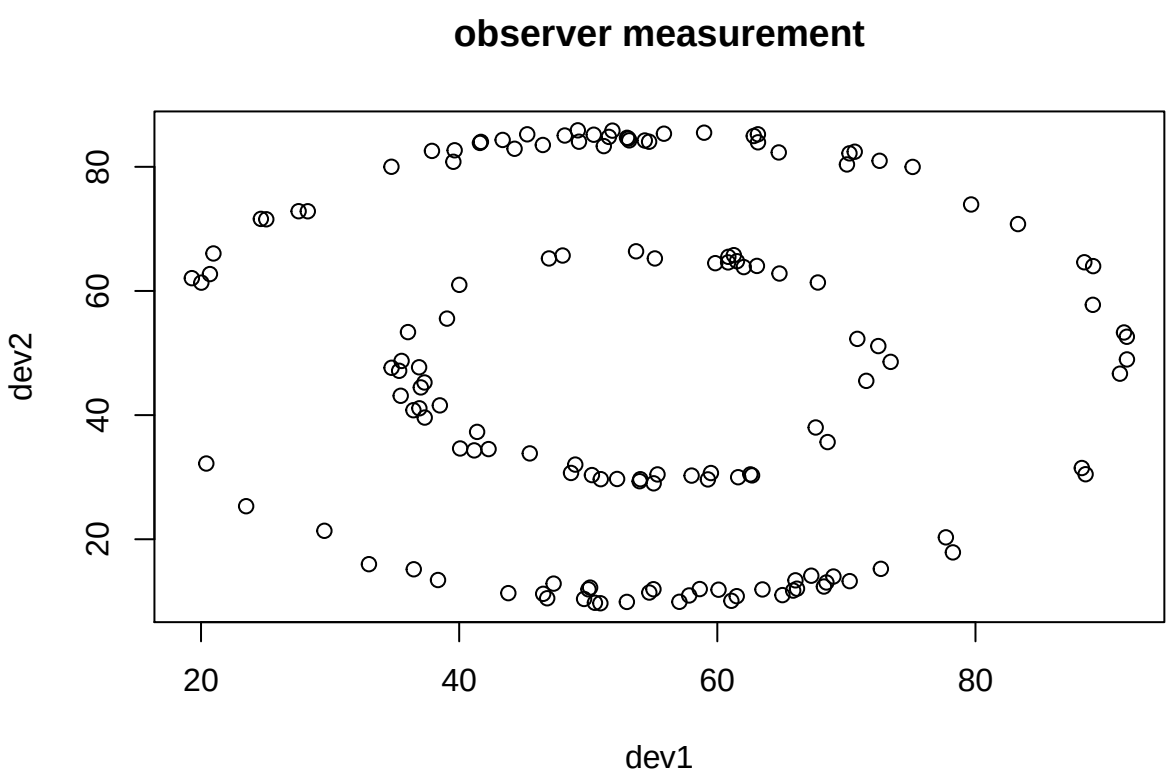
Then, let's create the function and make the plots.

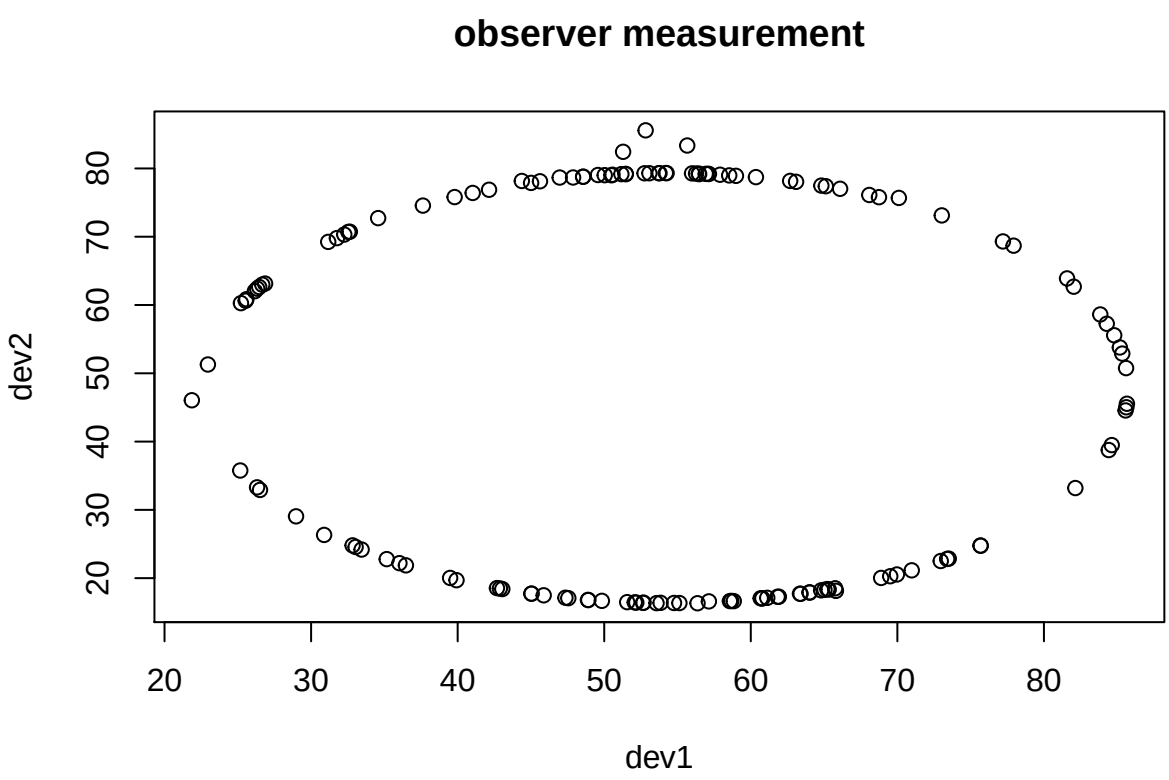
```
# the function make the plot
sc_plot <- function(data5, title, xlable, ylable) {
  plot(data5$x, data5$y, main = title, xlab = xlable, ylab = ylable)
}
sc_plot(data5, "device measurements", "dev1", "dev2")
```

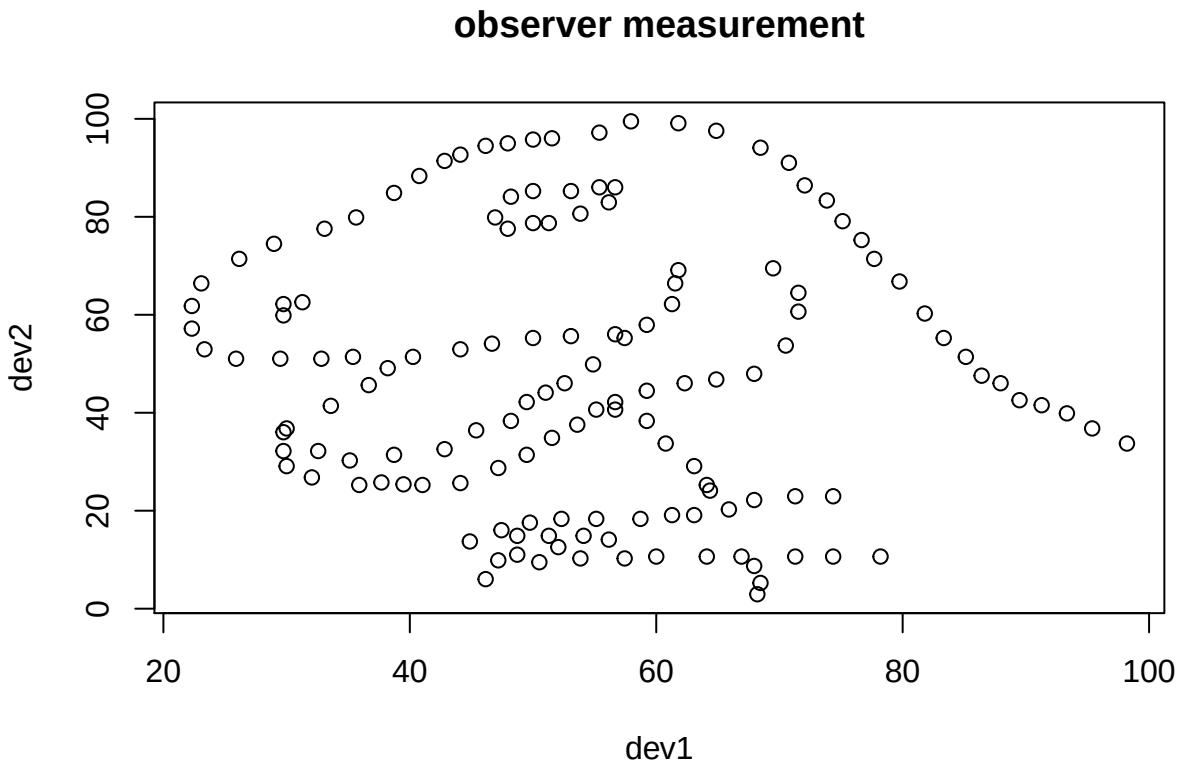


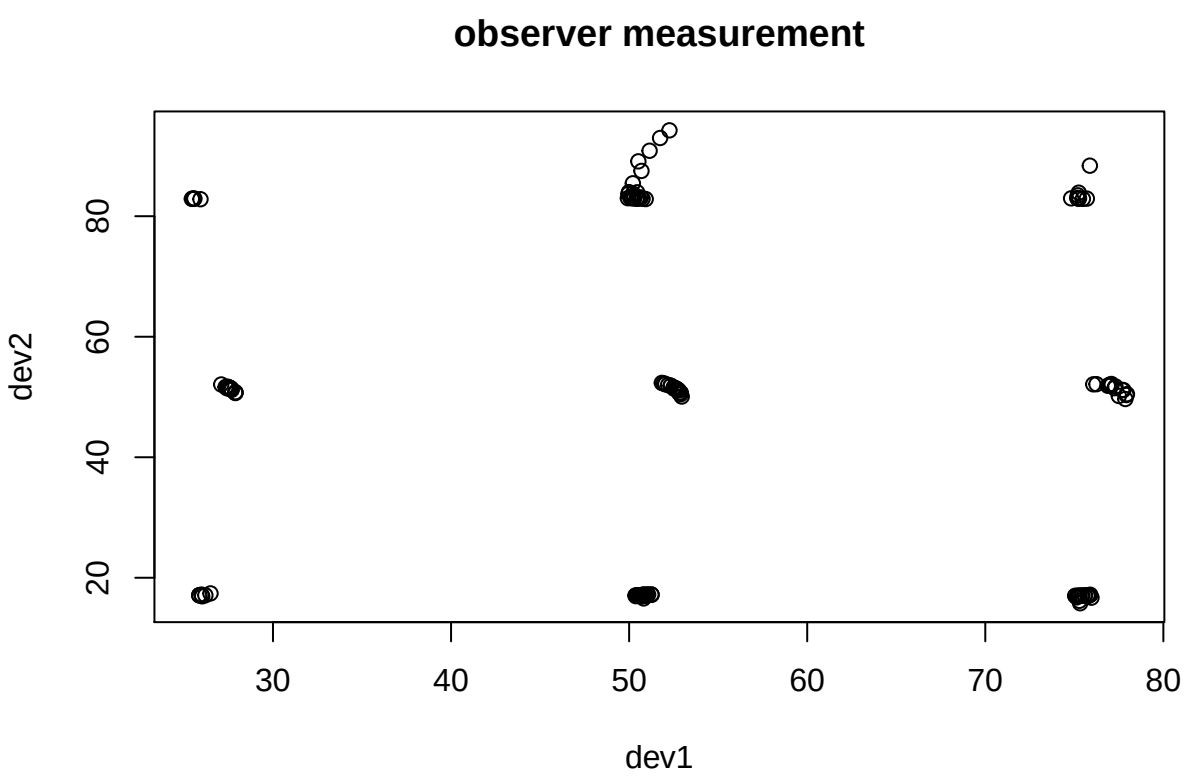

```
# the plots for 13 observers.
lapply(1:13, function(n) {
  sc_plot(data5[data5$Observer == n, ], "observer measurement", "dev1", "dev2")
})
```

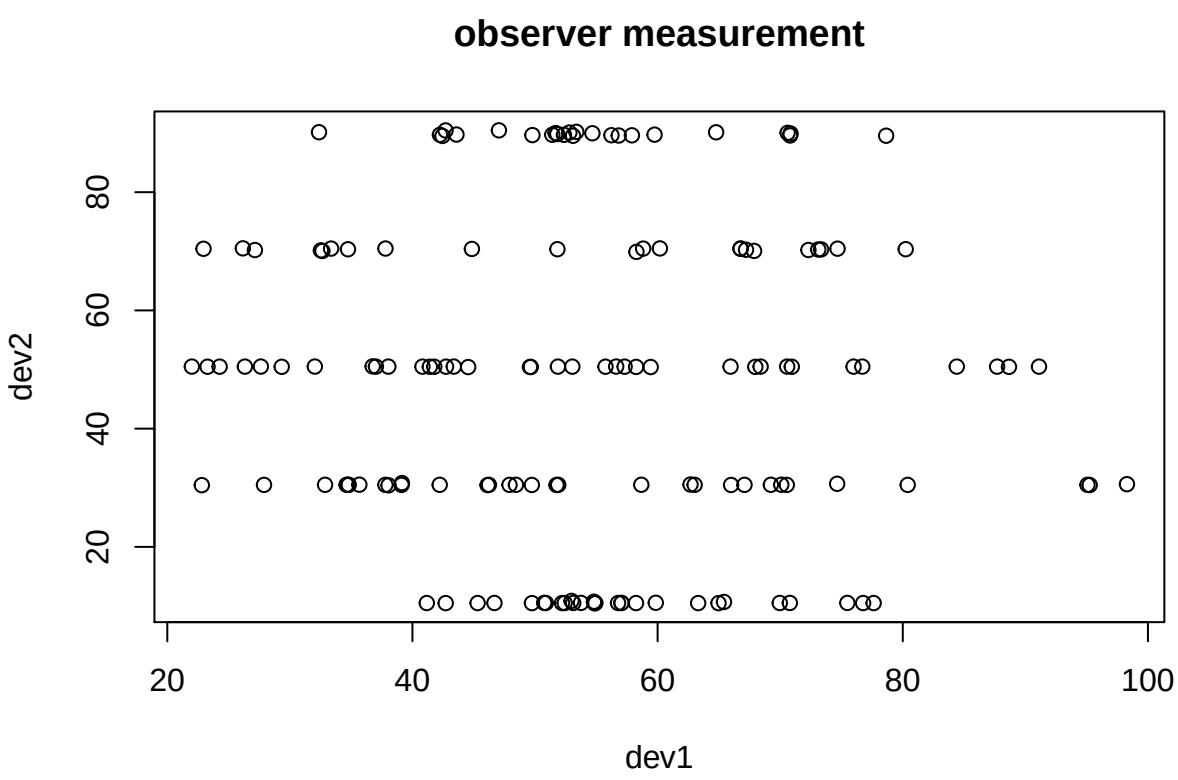

observer measurement
dev2
0
20
40
60
80
100
20
40
60
80
dev1

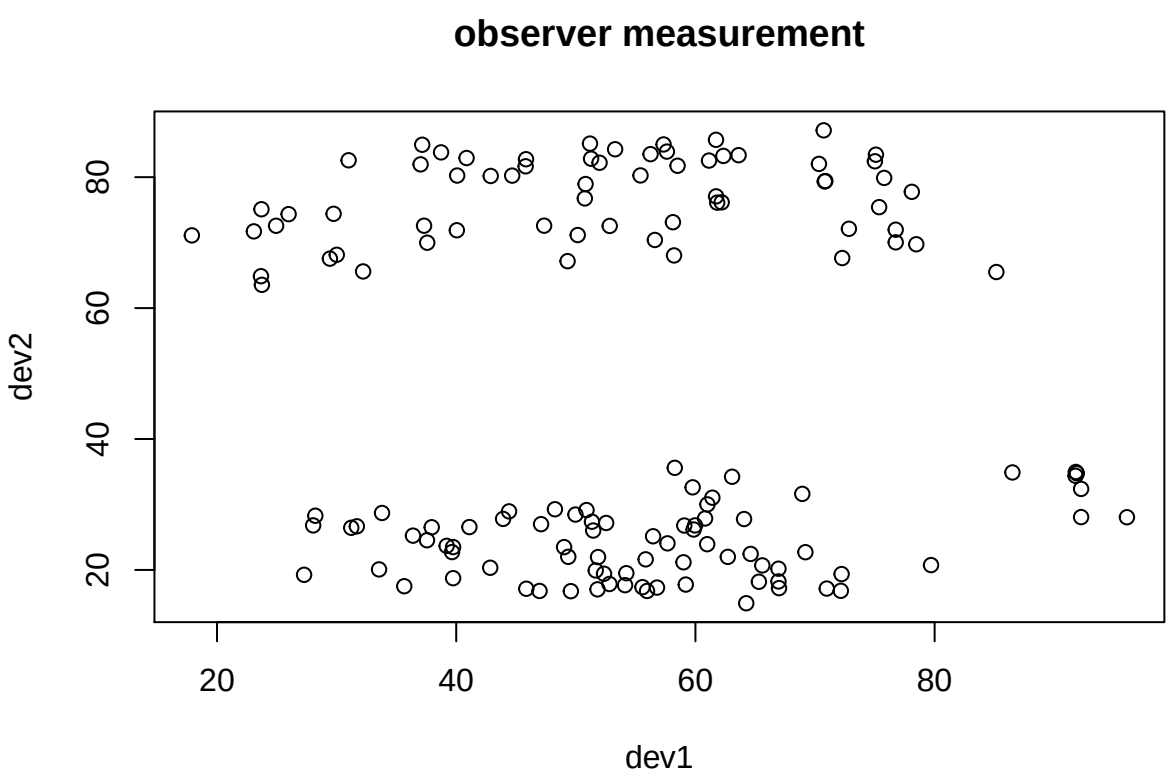

observer measurement
dev2
20
40
60
80
20
40
60
80
dev1

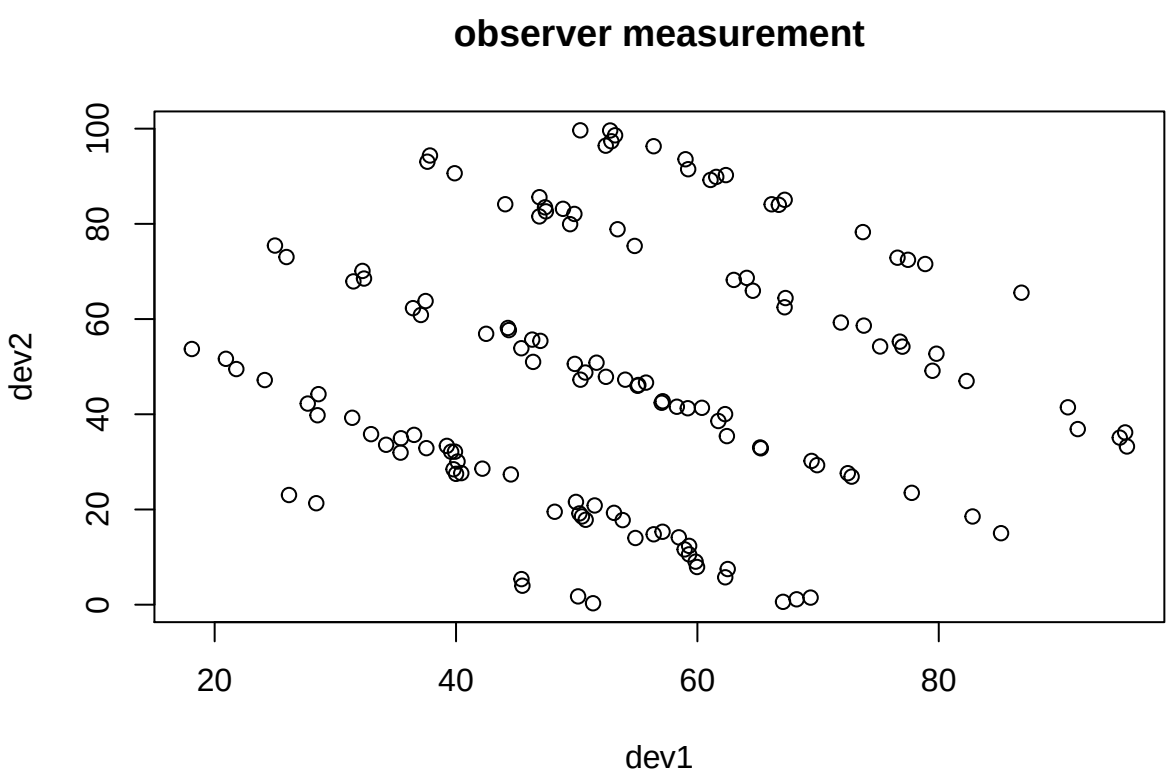

observer measurement
dev2
20
30
40
50
60
70
80
dev1
20
30
40
50
60
70
80

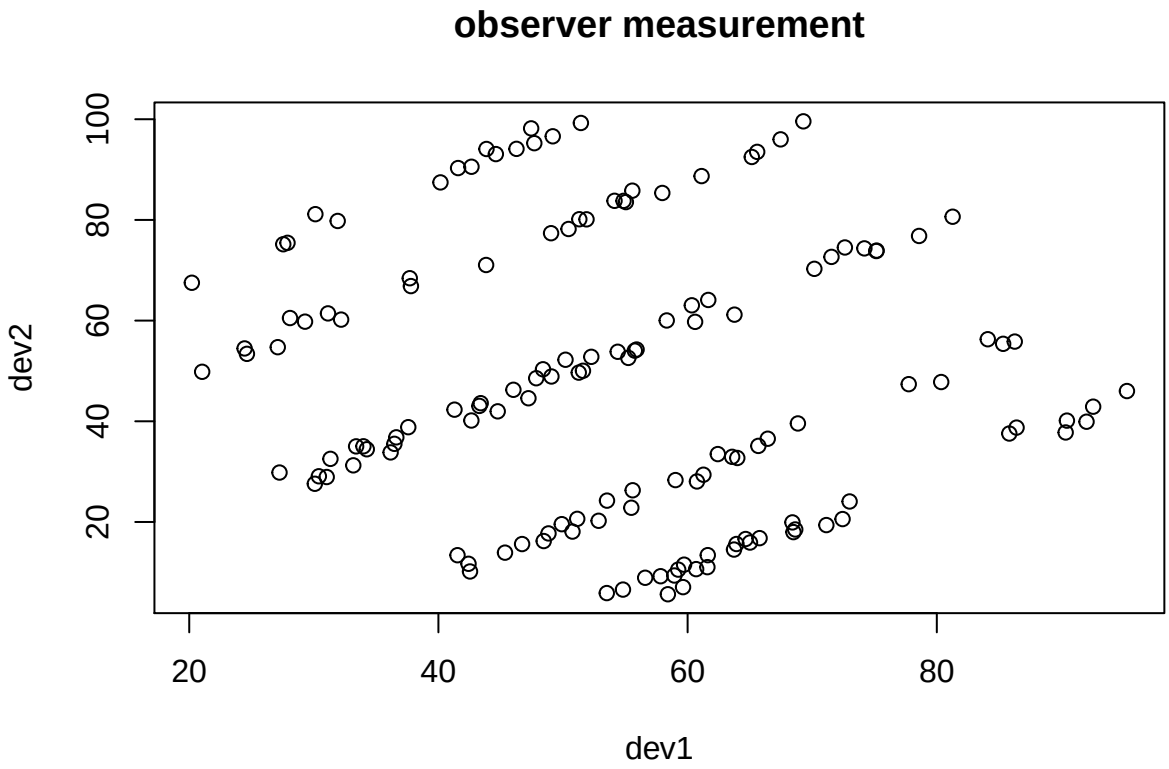

observer measurement
dev2
0
20
40
60
80
100
20
40
60
80
100
dev1

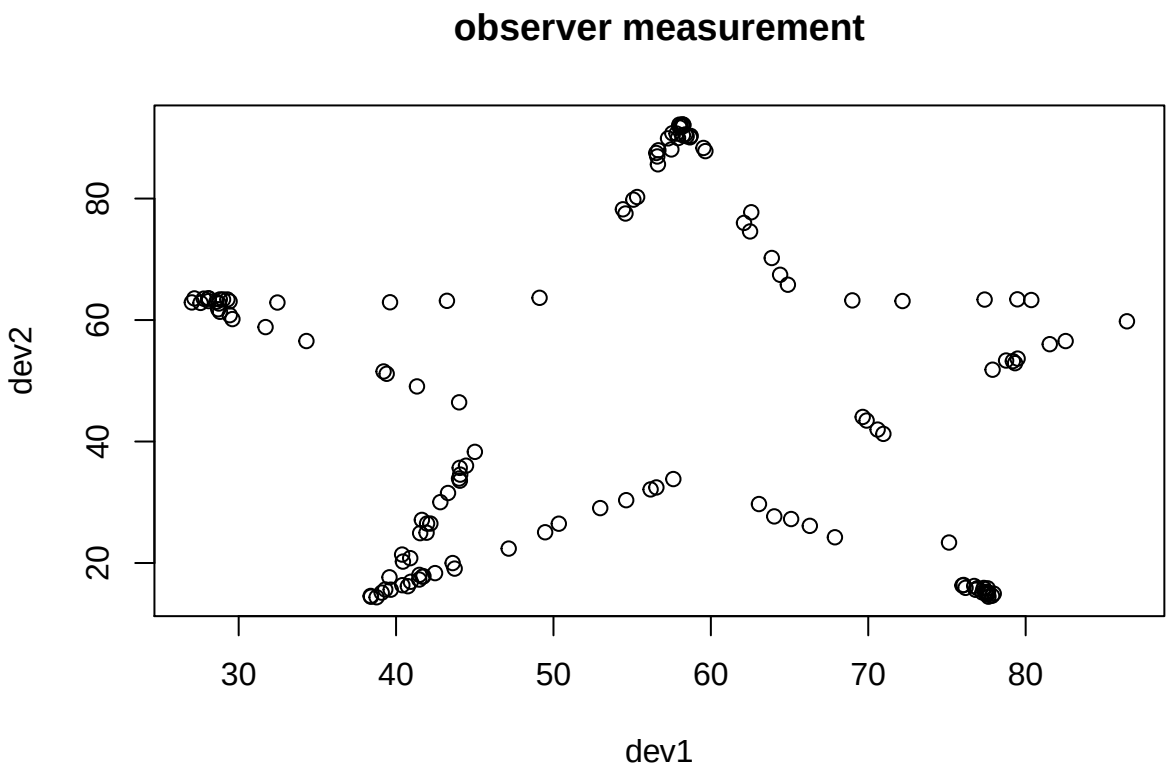

observer measurement
dev2
20
40
60
80
30
40
50
60
70
80
dev1

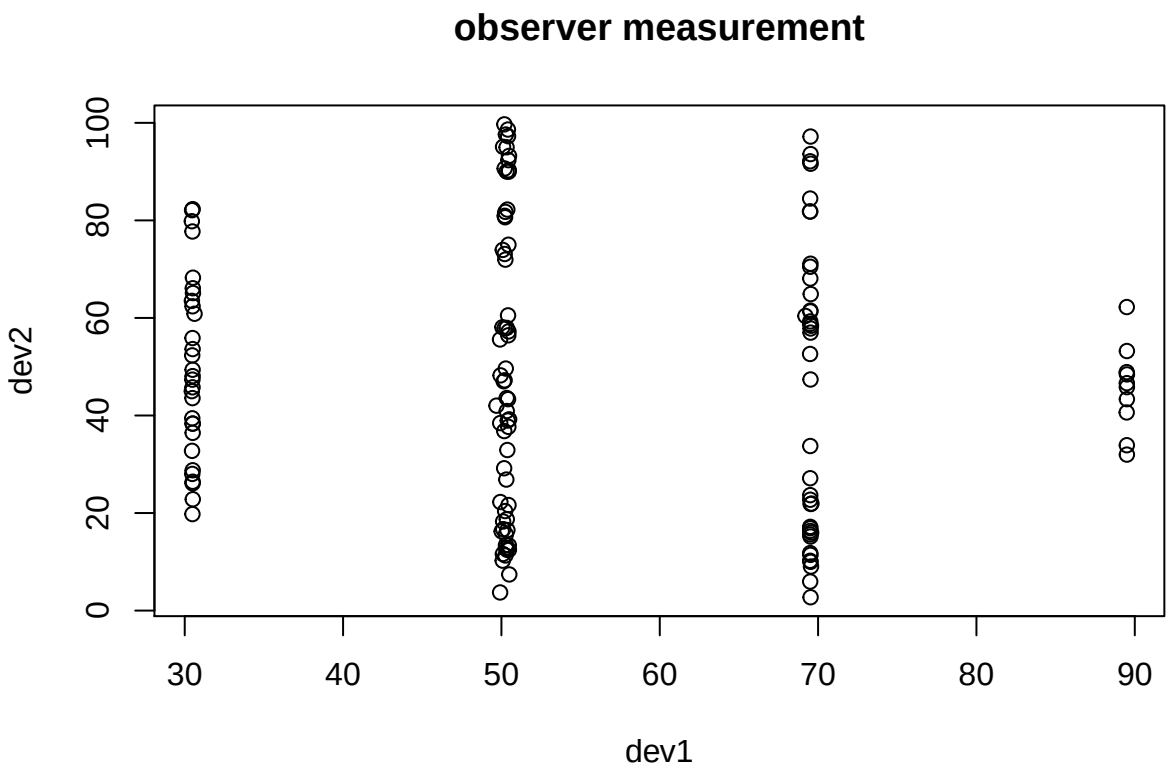

observer measurement
dev2
20
40
60
80
20
40
60
80
100
dev1

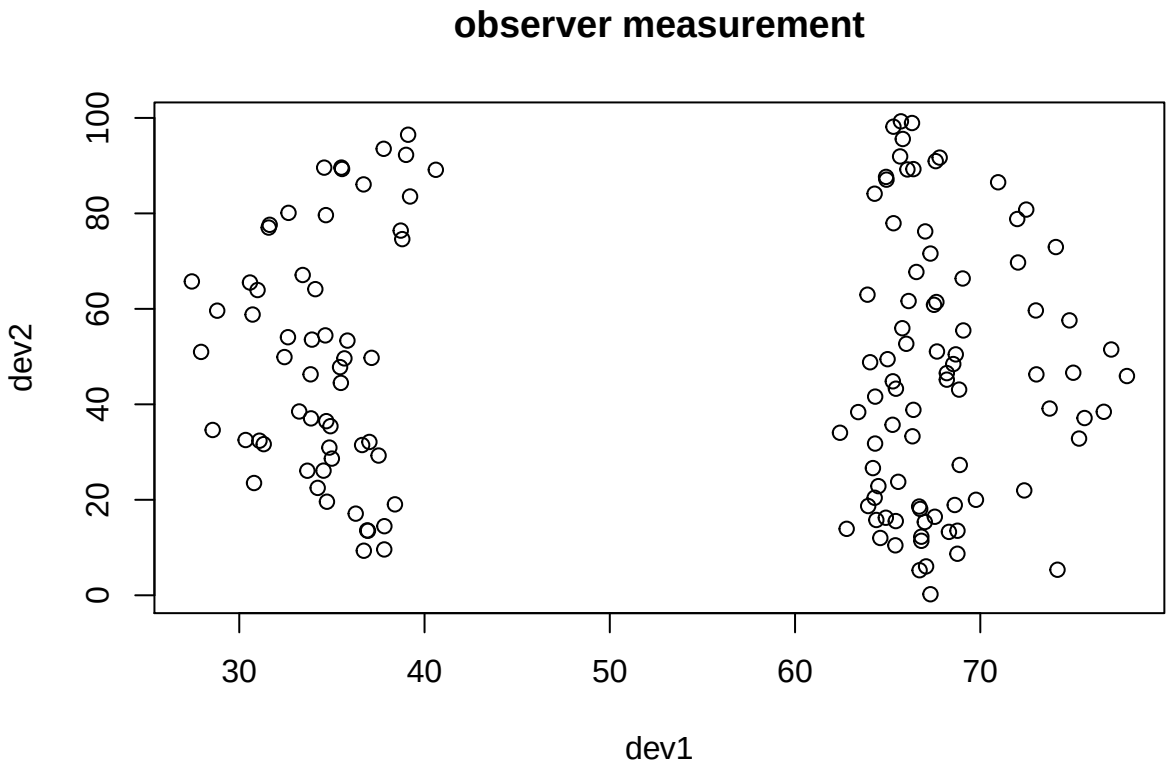

observer measurement
dev2
20
40
60
80
dev1
20
40
60
80

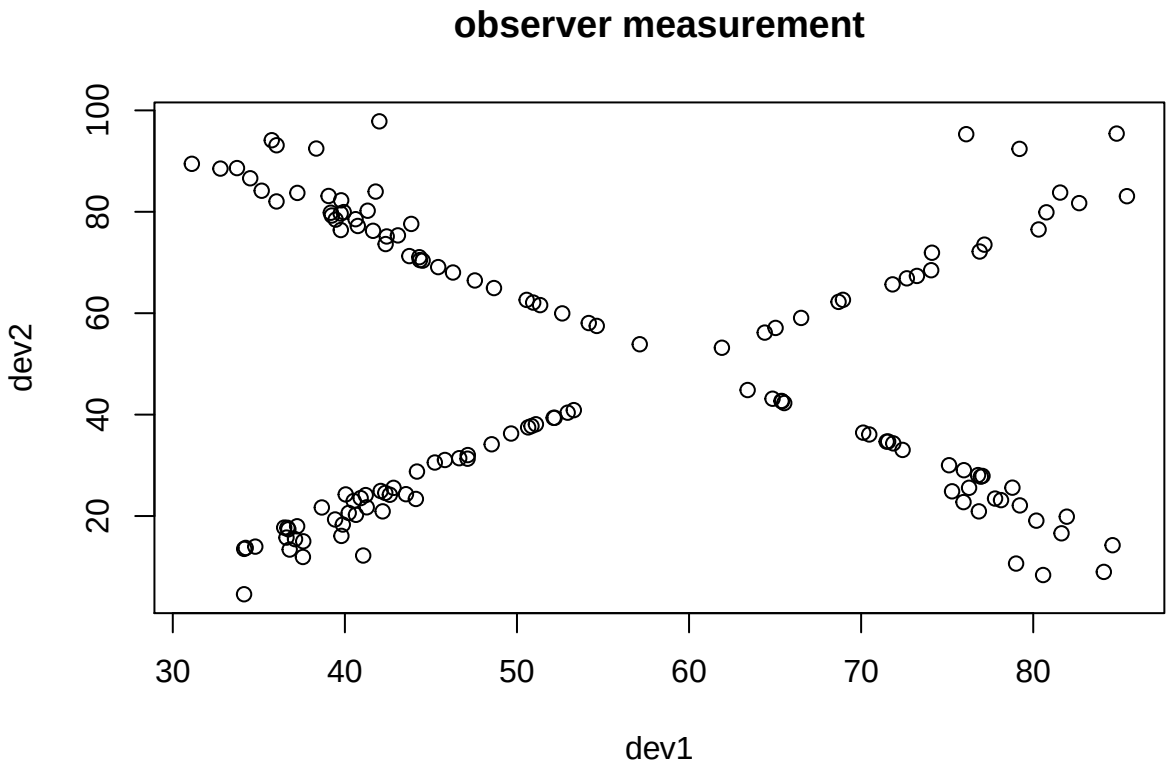

observer measurement
dev2
0
20
40
60
80
100
20
40
60
80
dev1


observer measurement
dev2
20
40
60
80
100
20
40
60
80
dev1


observer measurement
dev2
20
40
60
80
30
40
50
60
70
80
dev1


observer measurement
dev2
0
20
40
60
80
100
30
40
50
60
70
80
90
dev1


observer measurement
dev2
0
20
40
60
80
100
30
40
50
60
70
dev1




```
## [[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 8

a. Let's load the data.

b. let's create a summary table of the number of cities included by state.

```
city_count <- table(cities$State_Code)
```

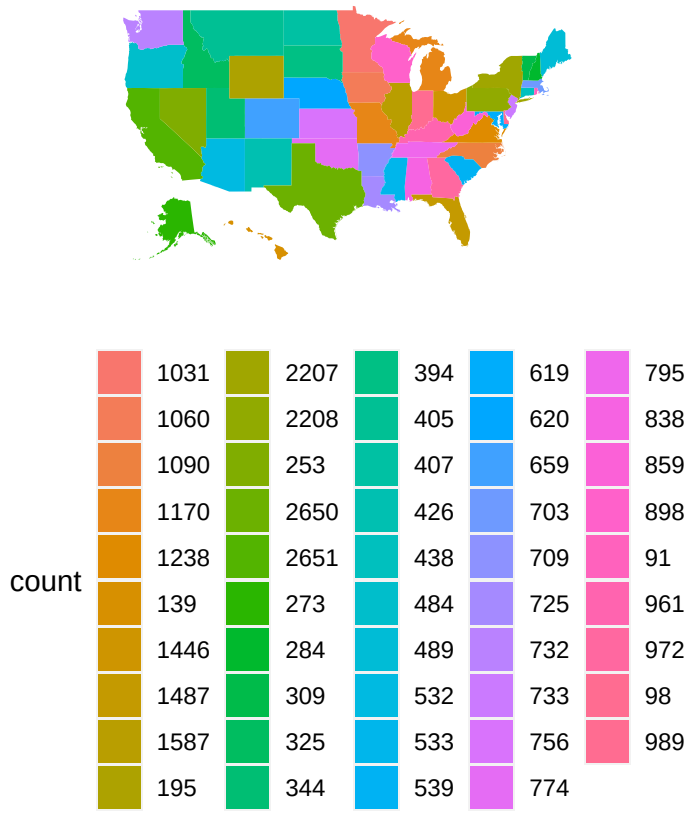
c. Create a dunction to count the letter.

```
getCount <- function(state_name, letter) {
  temp <- unlist(strsplit(tolower(state_name), split = ""))
  count <- 0
  for (i in 1:length(temp)) {
    if (temp[i] == letter) {
      count <- count + 1
    }
  }
  return(count)
}


letter_count <- data.frame(matrix(NA, nrow = 51, ncol = 26))
letter_count_results <- for (i in 1:51) {
  letter_count[i, ] <- sapply(1:26, function(n) {
    getCount(states$V2[i], letters[n])
  })
}
```

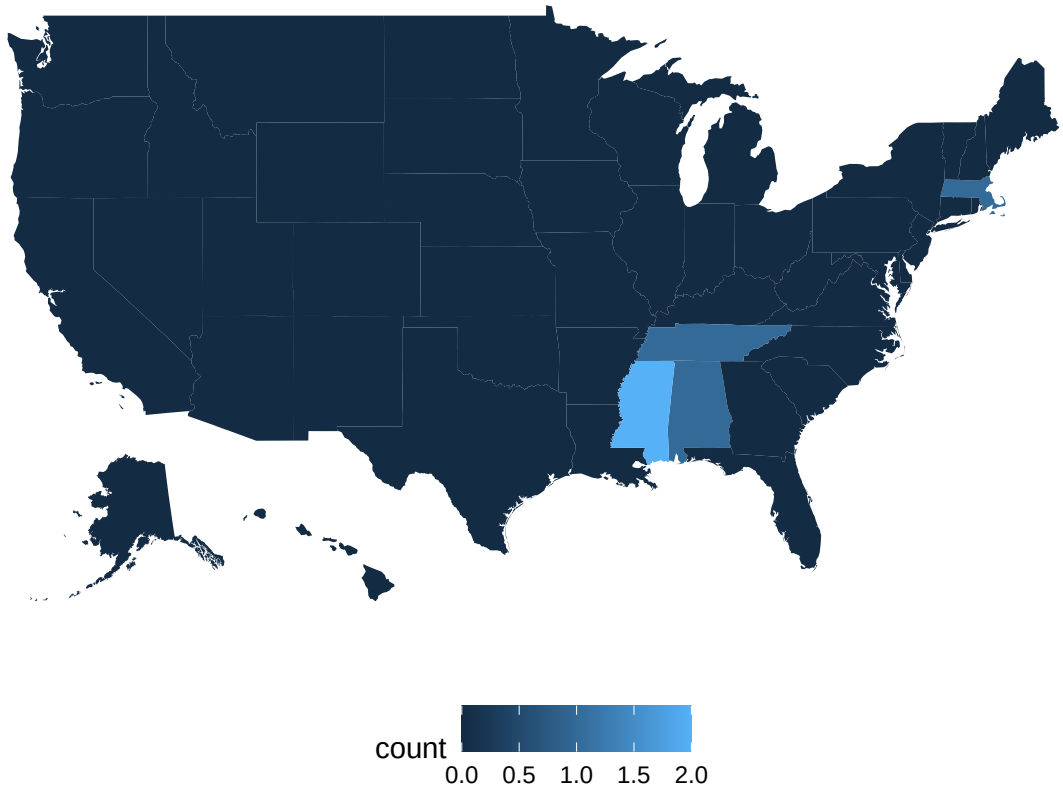
d. Let's meke two maps for U.S.

```
load("D:/VT/Rstudio/directory/fifty_states.rda")

# map1
city_count1 <- data.frame(state = tolower(rownames(city_count)), city_count)
city_count2 <- as.data.frame(cbind(tolower(states$V2), city_count1[-40, ]$Freq))
colnames(city_count2) <- c("state", "count")
```

```
# map_id creates the aesthetic mapping to the state name column in your data
map1 <- ggplot(city_count2, aes(map_id = state)) + # 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 =
  labs(x = "", y = "") + theme(legend.position = "bottom", panel.background = element_blank())
map1
```




```
# map2
state_letter <- data.frame(state = tolower(states$V2), rowSums(letter_count > 3))
colnames(state_letter) <- c("state", "count")

map2 <- ggplot(state_letter, aes(map_id = state)) + # 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 =
  labs(x = "", y = "") + theme(legend.position = "bottom", panel.background = element_blank())
map2
```



Problem 9

a. Let's try to run that code and find the errors.

```
library(quantreg)
library(quantmod)
# AAPL prices
apple08 <- getSymbols("AAPL", auto.assign = FALSE, from = "2008-1-1", to = "2008-12-31")[,
  6]
# 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)
set.seed(666)
Boot <- 1000
```

```r
sd.boot <- rep(0, Boot)
for (i in 1:Boot) {
  # nonparametric bootstrap
  bootdata <- df08[sample(nrow(df08), size = 251, replace = TRUE), ]
  sd.boot[i] <- coef(summary(lm(AAPL.Adjusted ~ IXIC.Adjusted, data = bootdata)))[2,
    2]
}
head(sd.boot)
```

```
## [1] 0.06861697 0.05623268 0.05940469 0.07227114 0.04946579 0.06534476
```

```r
summary(df08)
```

```
##       Index             AAPL.Adjusted       IXIC.Adjusted
##  Min.   :2008-01-03   Min.   :-19.74696   Min.   :-9.5877
##  1st Qu.:2008-04-03   1st Qu.: -2.42956   1st Qu.:-1.4093
##  Median :2008-07-02   Median : -0.09715   Median :-0.1532
##  Mean   :2008-07-02   Mean   : -0.32449   Mean   :-0.2074
##  3rd Qu.:2008-09-30   3rd Qu.:  1.88793   3rd Qu.: 1.0241
##  Max.   :2008-12-30   Max.   : 13.01939   Max.   :11.1594
```

After running and reading the errors we found two problems: 1. The author mistyped the "Boot" as "Boot_times". 2. "logapple08" and "logrm08" are not the variable names of "df08". He should checked the names of those two variables. 3. When sampling, there is no need to use argument: "replace=TRUE".

b. First, let's import the data and clean it.

```r
sensory_raw <- readRDS("D:/VT/Rstudio/directory/operator_data_raw.RDS")

# use the for circle to fill the NA values and correct the item number.
for (i in 2:length(sensory_raw$V6)) {
  if (is.na(sensory_raw$V6[i])) {
    sensory_raw$V6[i] <- sensory_raw$V1[i]
    sensory_raw$V1[i] <- sensory_raw$V1[i - 1]
  }
}
sensory <- sensory_raw[-1, ]
sensory <- data.frame(sensory$V1, operator = as.character(rep(c(1, 2, 3, 4, 5), 2)),
  sapply(stack(sensory[, -1]), as.numeric))
sensory <- sensory[, -4]
colnames(sensory) <- c("item", "operator", "values")
```

Now lets's bootstrap the analysis to get the parameter estimates using 100 bootstrapped samples.

```r
# system time
time1 <- system.time({
  set.seed(4578)
  beta.boot <- matrix(NA, nrow = 100, ncol = 2)
  for (j in 1:100) {

    boot_data <- matrix(NA, nrow = 75, ncol = 2)
```

```
    for (i in 1:5) {
      boot_data[((i - 1) * 15 + 1):(15 * i), ] <- as.matrix(sensory[sensory$operator ==
        i, ][sample(c(1:30), 15, FALSE), 2:3])
    }

    # nonparametric bootstrap
    boot_data <- data.frame(sapply(boot_data[, 1], as.numeric), boot_data[, 2])
    colnames(boot_data) <- c("operator", "values")
    beta.boot[j, ] <- lm(values ~ operator, data = boot_data)$coefficients
  }
})

# The coefficients
kable(beta.boot[1:10, ], caption = "first 10 coefficients' value")
```

Table 2: first 10 coefficients' value

4.443333	0.0740000
4.583333	0.0380000
4.422667	0.1866667
4.214667	0.1066667
4.663333	0.0313333
4.776667	-0.0473333
4.256667	0.1486667
4.271333	0.1580000
5.401333	-0.1893333
5.242000	-0.1446667

c. Do the bootstraps in parallel and count the time.

```
# do in parallel

cluster <- makeCluster(4, type = "SOCK")
registerDoSNOW(cluster)

# write the bootstrap into a function
time2 <- system.time({
  bootstrap <- function(a) {
    beta.boot <- matrix(NA, nrow = 1, ncol = 2)
    boot_data <- matrix(NA, nrow = 75, ncol = 2)
    for (i in 1:5) {
      boot_data[((i - 1) * 15 + 1):(15 * i), ] <- as.matrix(sensory[sensory$operator ==
        i, ][sample(c(1:30), 15, FALSE), 2:3])
    }

    # nonparametric bootstrap
    boot_data <- data.frame(sapply(boot_data[, 1], as.numeric), boot_data[, 2])
    colnames(boot_data) <- c("operator", "values")
    beta.boot <- lm(values ~ operator, data = boot_data)$coefficients
    return(beta.boot)
  }
```

```
  results <- foreach(n = 1:100, .combine = rbind) %dopar% bootstrap(n)
})

stopCluster(cluster)
kable(results[1:10, ], caption = "first 10 coefficients' value")
```

Table 3: first 10 coefficients' value

	(Intercept)	operator
result.1	4.798000	0.0180000
result.2	4.526667	0.0893333
result.3	4.220667	0.0406667
result.4	4.062667	0.2520000
result.5	4.577333	-0.0173333
result.6	4.885333	-0.0493333
result.7	4.883333	-0.0926667
result.8	4.086667	0.1693333
result.9	4.220667	0.1846667
result.10	3.890000	0.1846667

```
kable(rbind(time1, time2))
```

	user.self	sys.self	elapsed	user.child	sys.child
time1	0.22	0.01	0.24	NA	NA
time2	0.06	0.04	0.32	NA	NA

Now we reran the bootstraps in parallel and created a table comparing the time used in partb and partc, the system tiem used is less than when doing not in parallel.

Problem 10

a. Do the bootstraps based on Newton's method to get the roots.

```
# the vector covering all of the roots with lenth as 1000
X <- seq(-2, -22, length.out = 1000)

# Define the f(x)
fun1 <- function(x) 3^x - sin(x) + cos(5 * x)

# The derivative of f(X)
fun_de <- function(x) 3^x * log(3) - cos(x) - 5 * sin(5 * x)
# Define the solution
solution <- function(x1) {

  # Define the matrix with NA values firstly
  n <- 2
  x <- matrix(NA, nrow = 5000, ncol = 4)
```

```
  x0 <- matrix(c(1, 2, 3, 4, 1, 2, 4, 3), nrow = 2, byrow = TRUE)
  x <- rbind(x0, x)
  x_n <- pi
  # When there is few difference between the X_n and X_n-1, it means x converges,
  # x_n is the solution.
  while (abs(x[n, 3] - x[n - 1, 3]) > 1e-06) {
    x_n <- x1 - fun1(x1)/fun_de(x1)
    n <- n + 1
    x[n, ] <- c(n, x1, x_n, fun1(x_n))
    x1 <- x_n
  }
  x_solutions <- as.data.frame(cbind(seq(1, 4998, 1), x[3:5000, ]))[1:(n - 2),
    -2]
  colnames(x_solutions) <- c("iterations", "x_n-1", "x_n", "f(x)")

  return(x_n)

}
root <- unique(sapply(1:1000, function(n) {
  solution(X[n])
}))
time3 <- system.time({
  unique(sapply(1:1000, function(n) {
    solution(X[n])
  }))
})
kable(root[1:10], caption = "first 10 roots")
```

Table 5: first 10 roots

x
-5.107437
-3.930114
-3.528723
-2.887058
-2.887058
-3.528723
-2.887058
-27.096237
-2.887058
-9.162986

```
time3
```

```
##    user  system elapsed
##    0.62    0.02    0.66
```

b. Do in parallel.

```
cores <- 8
cl <- makeCluster(cores)
clusterExport(cl = cl, c("fun1", "fun_de", "solution"), envir = environment())
time4 <- system.time({
  parSapply(cl, X, solution)
})
stopCluster(cl)
kable(rbind(time3, time4))
```

	user.self	sys.self	elapsed	user.child	sys.child
time3	0.62	0.02	0.66	NA	NA
time4	0.02	0.00	0.29	NA	NA

Now we reran the bootstraps in parallel and created a table comparing the time used in part a and part b, the system tiem used is less than when doing not in parallel.