HW4_Jingbin

Jingbin Xu

10/15/2020

Problem 1

Given the algorithm, we compare the results with linear regression. Here is our tolerance used and the step size α.

Tolerance	Step Size	theta 0	theta 1	lm theta 0	lm theta 1
1e-07	0.01	0.9699	2.0015	0.9696	2.0016

From the above table, we found that the coefficients computed from gradient descent algorithm are close to the fitted linear regression.

```r
# given the X and h below
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))
# gradient descent input with theta, alpha, tolerance and m
gradient_descent <- function(theta, alpha = 0.01, tolerance = 1e-07, m = 100) {
  # set up parameters for interation
  theta0i.old <- 0
  theta1i.old <- 0
  theta0i.new <- 1
  theta1i.new <- 2
  n <- 0
  # while loop stop when the absolute values less than the tolerance
  while (abs(theta0i.new - theta0i.old) > tolerance && abs(theta1i.new - theta1i.old) >
    tolerance) {
    theta0i.old <- theta0i.new
    theta1i.old <- theta1i.new
    theta.matrix <- matrix(c(theta0i.old, theta1i.old), nrow = 2)
    theta0i.new <- theta0i.old - alpha * (1/m) * sum(X %*% theta.matrix - h)
    theta1i.new <- theta1i.old - alpha * (1/m) * (t(X %*% theta.matrix - h) %*%
      X[, 2])
    n <- n + 1
  }
  # return our results
  results <- cbind(theta_0 = theta0i.new, theta_1 = theta1i.new, iteration = n)
  return(results)
}

# test our function gradient descent
```

```r
function.results <- gradient_descent(theta)
# lm parameter
lm(h ~ 0 + X)
```

Problem 2

part a

We use a step size of 1e-7, tolerace of 1e-9 and set a stopping rule for 5M. The total number of iteration is 362204. The average of theta0 is 0.9995, the average of theta1 is 1.9979.

```r
# parallel computing
cores <- detectCores() - 1
cluster <- makeCluster(cores, type = "SOCK")
registerDoSNOW(cluster)
clusterExport(cluster, c("X", "h"))
n <- 10000

# take advantage f parallel computing opportunities
theta0 <- seq(0, 2, length.out = 100)
theta1 <- seq(1, 3, length.out = 100)
theta.matrix <- rbind(rep(theta0, each = 100), rep(theta1, 100))

system.time({
  final_results <- foreach(n = 1:n, .combine = rbind) %dopar% gradient_descent(theta.matrix[,
    n], alpha = 1e-07, tolerance = 1e-09)
})

stopCluster(cluster)
```

part b

The above implementation is not a good way for gradient descent algorithm. Because it is time consuming. Although different tolence results differ, the loop takes a long run to converge.

part c

It is challenging to choose a proper parameter in our case. Maybe we should give this algorithm a smoother function to fit.

Problem 3

John Cook suggests that instead of solving for the inverse of $(X^T X)^{-1}$ we use the solve function to find $\hat{\beta}$ by solving the following system $(X^T X)\hat{\beta} = X^T \underline{y}$. However, we could also do $\hat{\beta} = (X^T X)^{-1} X^T \underline{y}$, which will take much more computation efforts. But the result will be the same.

Problem 4

part a

- The size of A is 112347224 bytes.
- The size of B is 1816357208 bytes.
- The time used on computation is 881.27 seconds.

```r
# set up our parameters
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

# the size of A and B and compute system time
object.size(A)
object.size(B)
system.time({
  y <- p + A %*% solve(B) %*% (q - r)
})
```

part b

The time consuming part is the inverse of the matrix. If we could come up with a method to speed up the matrix inverse.

part c

We inverse the matrix by converting the matrix by diagnosis elements, which bring down the time to 98 seconds.

```r
inverse.R <- solve(R)
new.C <- kronecker(R, diag(1600))
inverse.B <- new.C[-id, -id]
# system.time({y <- p + A%*%B_inv%*%(q-r)})
```

Problem 5

part a

```r
# Create a function
success.pro <- function(x = c(0, 1, 1, 0)) {
  n <- length(x)
  success <- sum(x)
  proportion <- success/n
  return(proportion)
}
```

part b

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

part c

The matrix is not random.

```
# First we compute the proportion of success by column
for (i in 1:10) {
  pro <- success.pro(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"
```

```
# compute the proportion of success by row
for (i in 1:10) {
  pro <- success.pro(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"
```

part d

From the results, we use sapply function to create random matrix. And now the function works.

```
create.vect <- function(p = 0.5) {
  data <- rbinom(10, 1, prob = p)
  return(data)
}
prob <- c(seq(0.31, 0.4, by = 0.01))
P4b_data <- sapply(prob, create.vect)
proportion_P4b_col <- apply(P4b_data, 2, success.pro)  # by column
proportion_P4b_row <- apply(P4b_data, 1, success.pro)  # by column
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
```

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

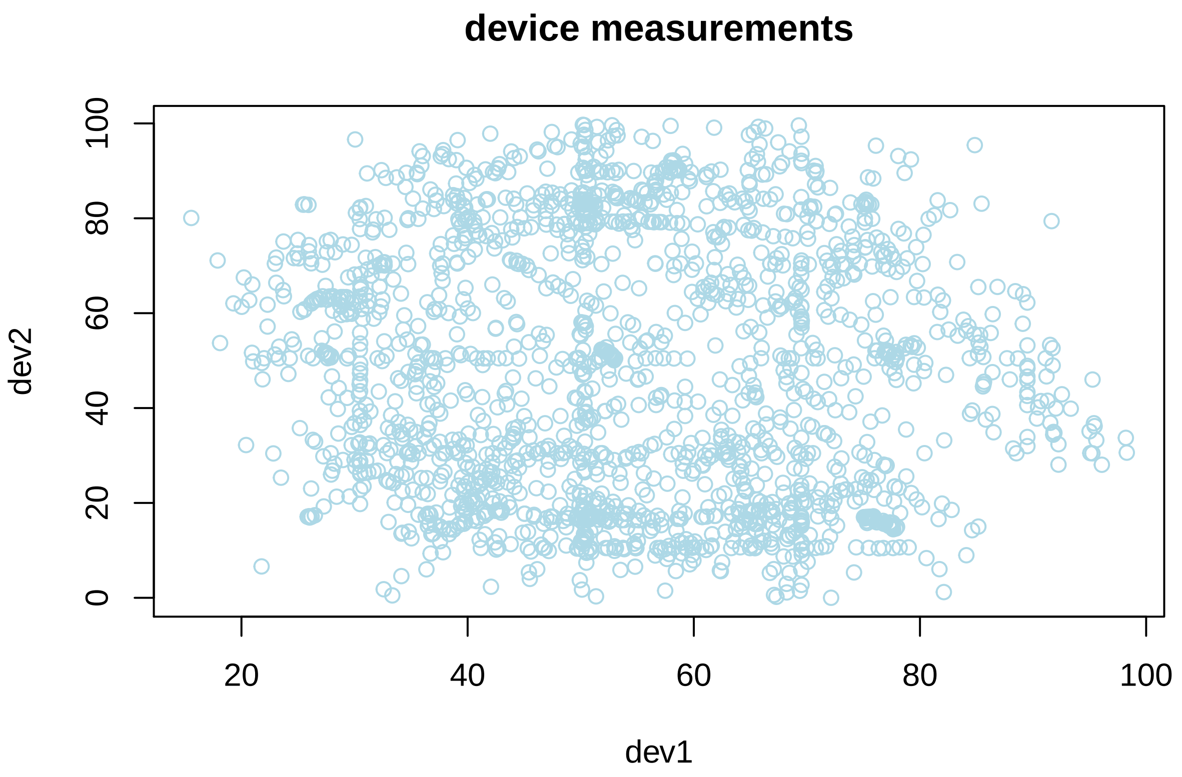
Problem 6

part 1

```
setwd("~/Desktop")
# Import the data from last assignment
data.device <- data.frame(readRDS("HW3_data.rds"))
data.device <- data.frame(data.device)
colnames(data.device) <- c("observer", "x", "y")
# Create a scatter plot based on x and y
scatterplot <- function(X = data.device) {
  x <- X[, 2]
  y <- X[, 3]
  plot <- return(plot(x, y, col = "lightblue"))
}
```

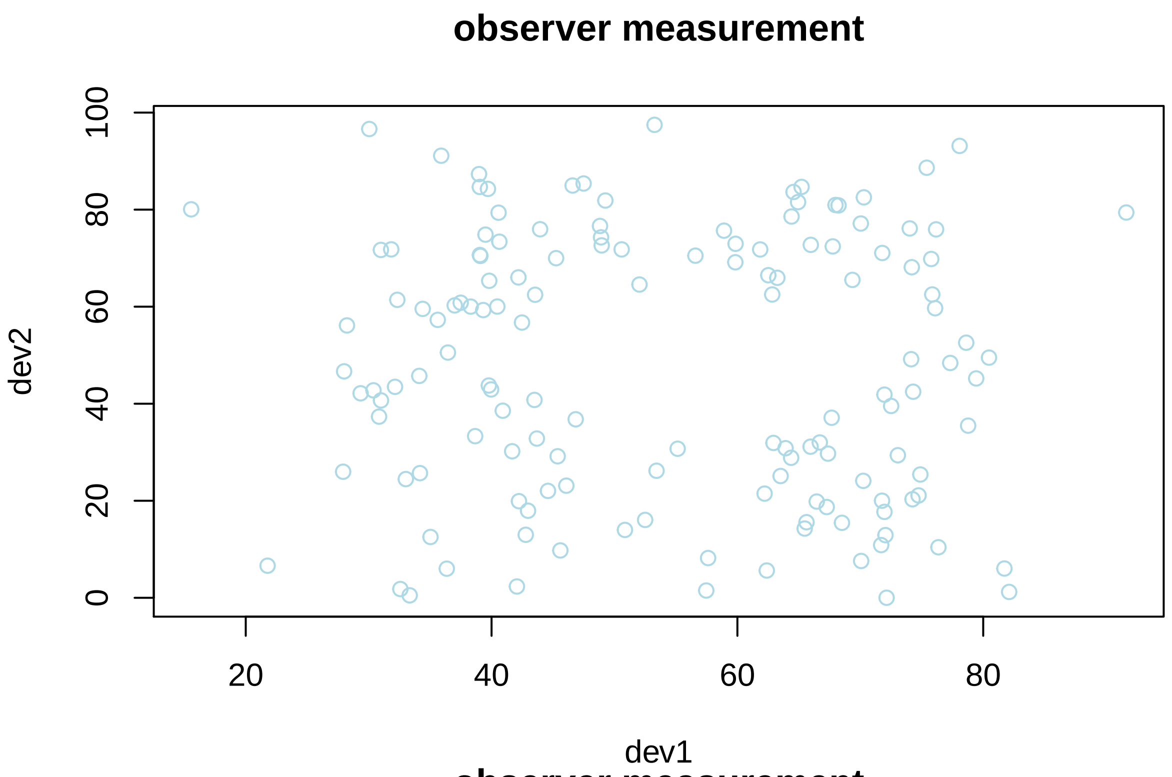
part 2

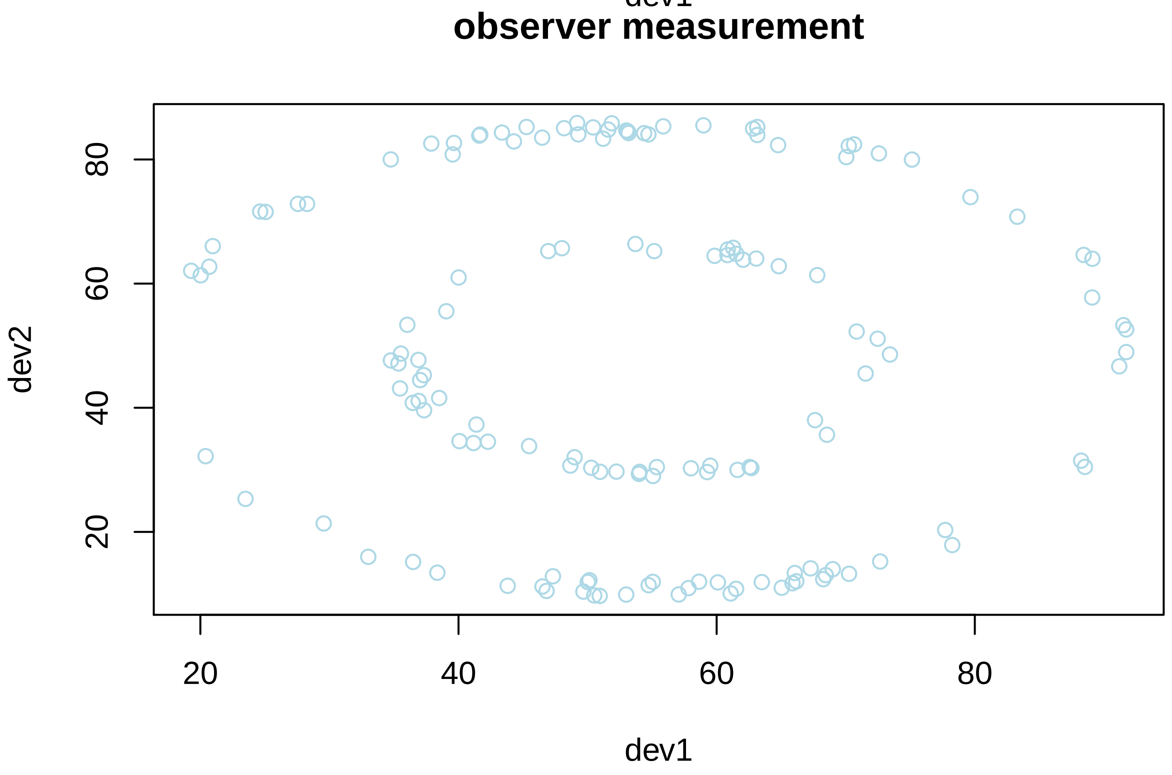
```
# the function make the plot
scatter.plot <- function(data, title, xlab, ylab) {
  plot <- plot(data$x, data$y, main = title, xlab = xlab, ylab = ylab, col = "light blue")
  return(plot)
}

scatter.plot(data.device, "device measurements", "dev1", "dev2")
```

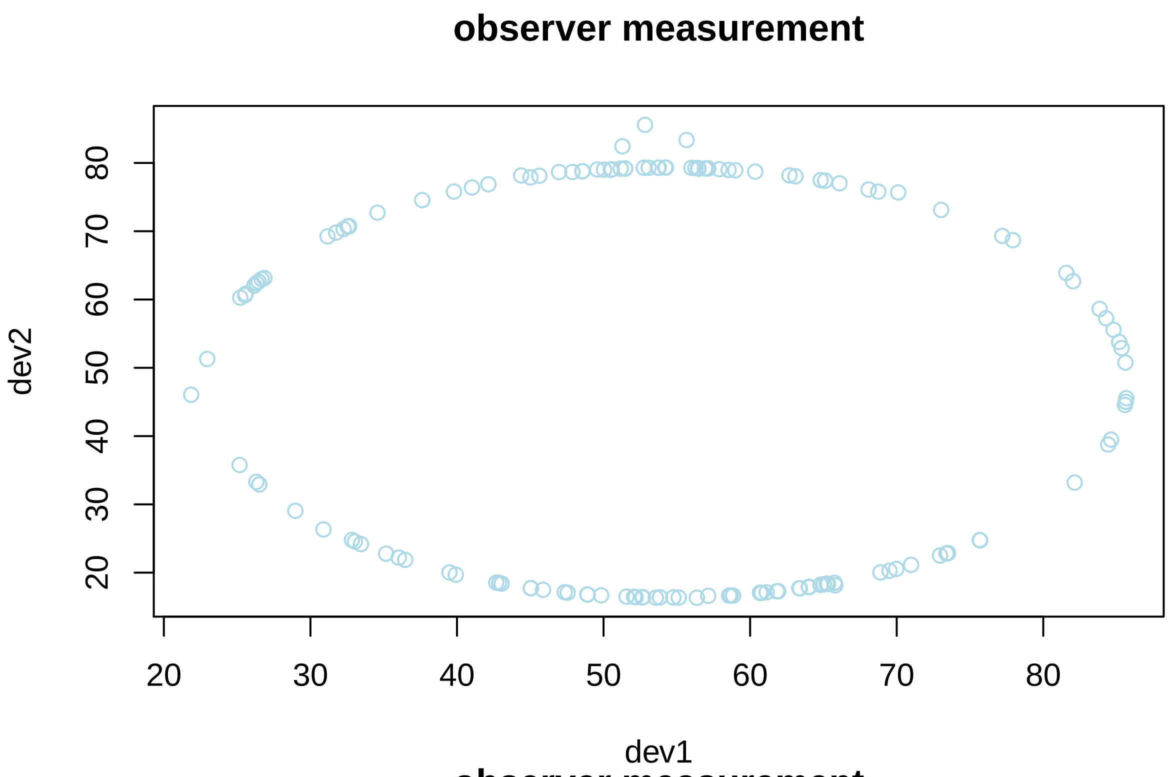


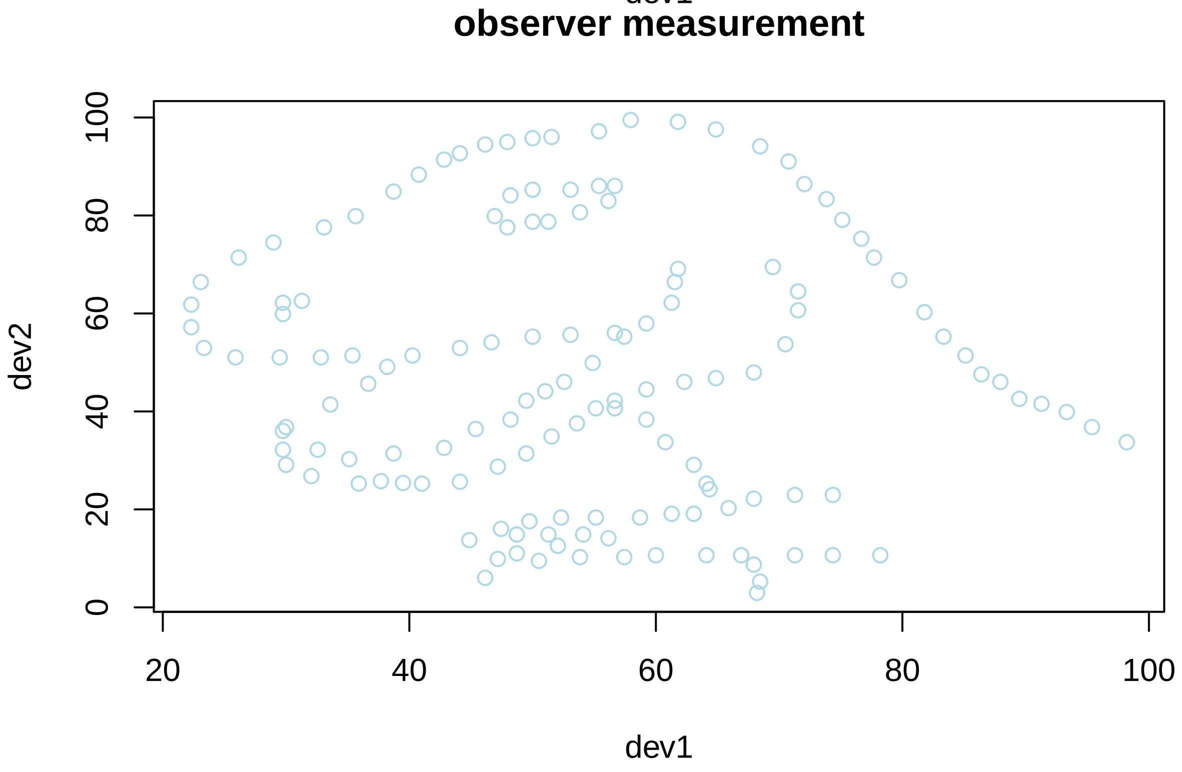
```
## NULL
```

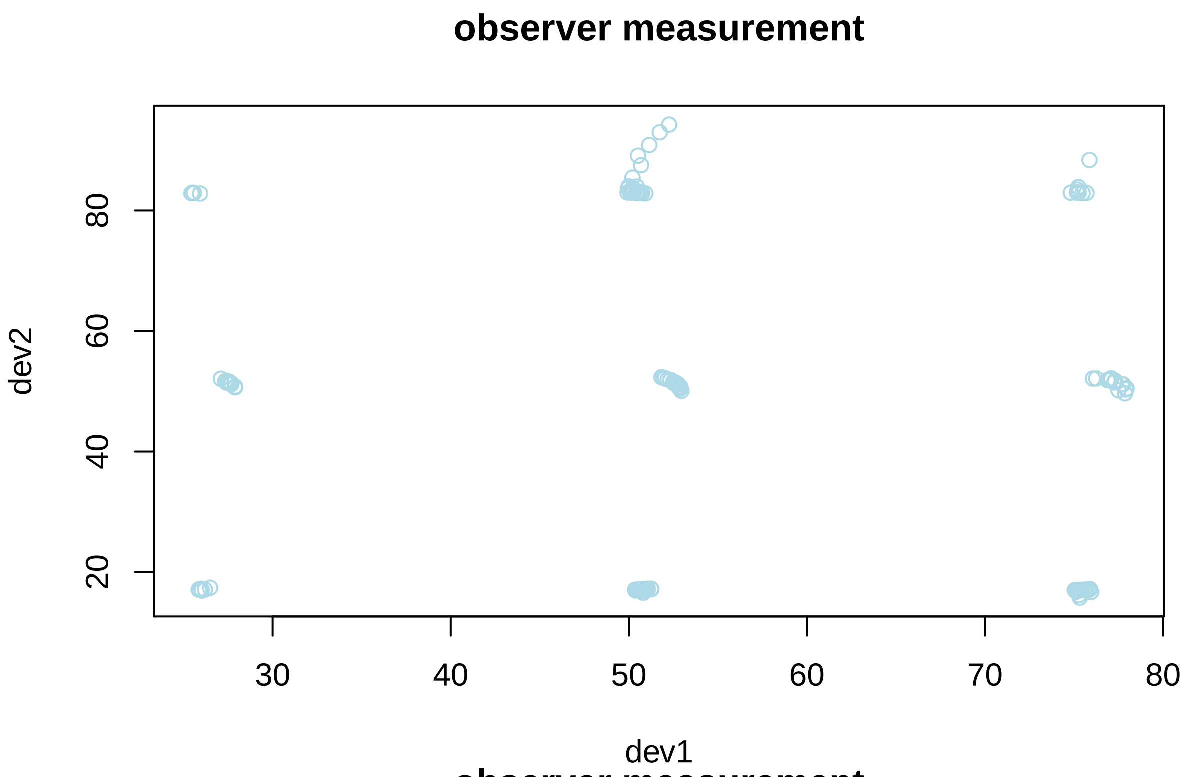
```
# the plots for 13 observers
lapply(1:13, function(n) {
  scatter.plot(data.device[data.device$observer == n, ], "observer measurement",
    "dev1", "dev2")
})
```

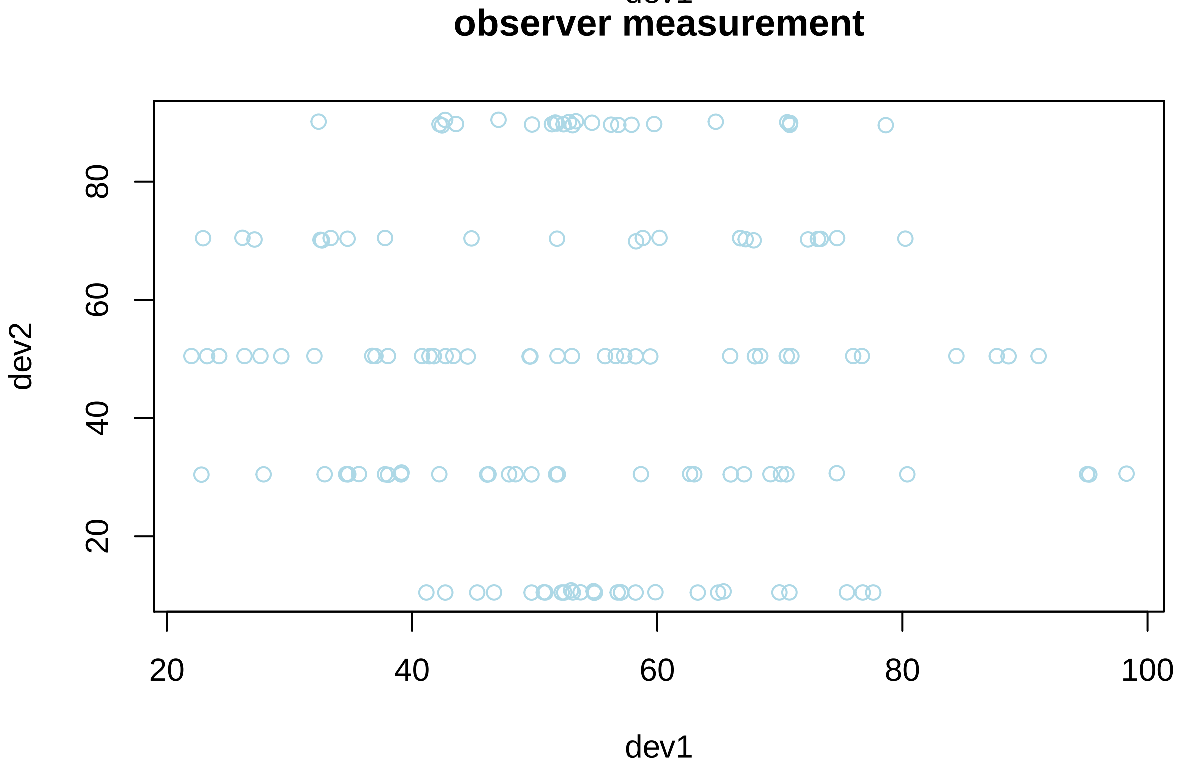

observer measurement
dev2
dev1
0
20
40
60
80
100
20
40
60
80

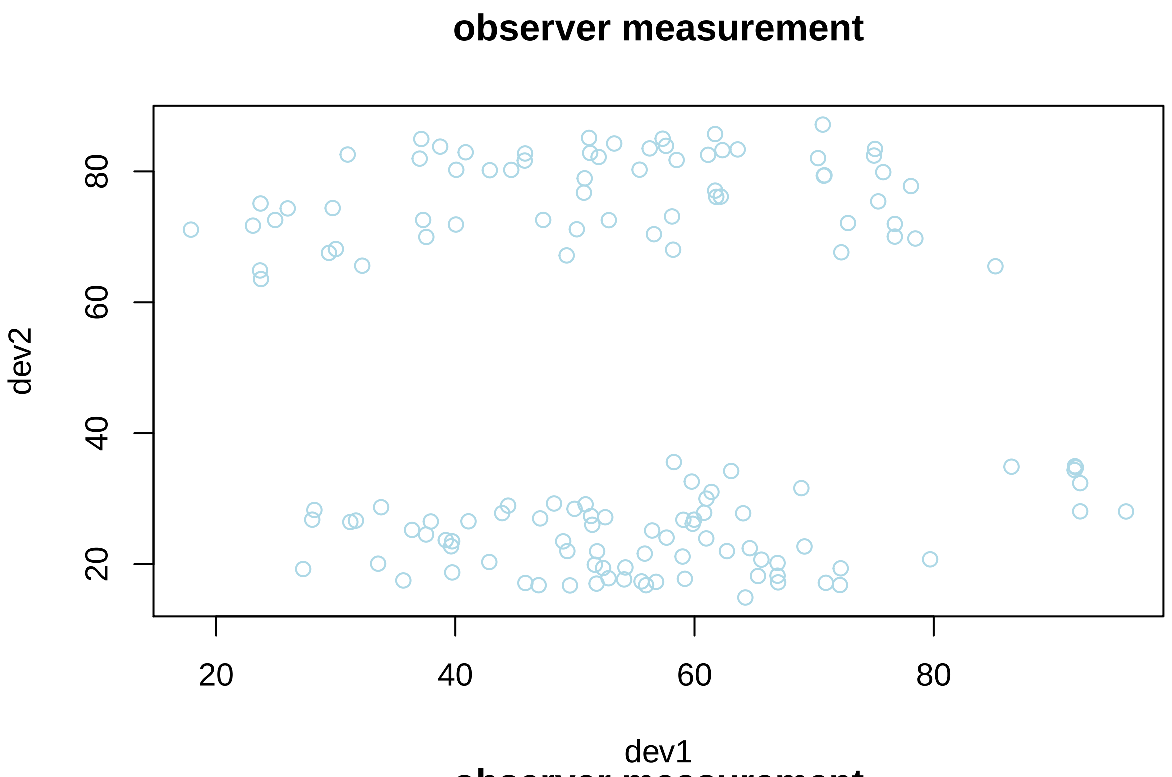

observer measurement
dev2
dev1
20
40
60
80
20
40
60
80

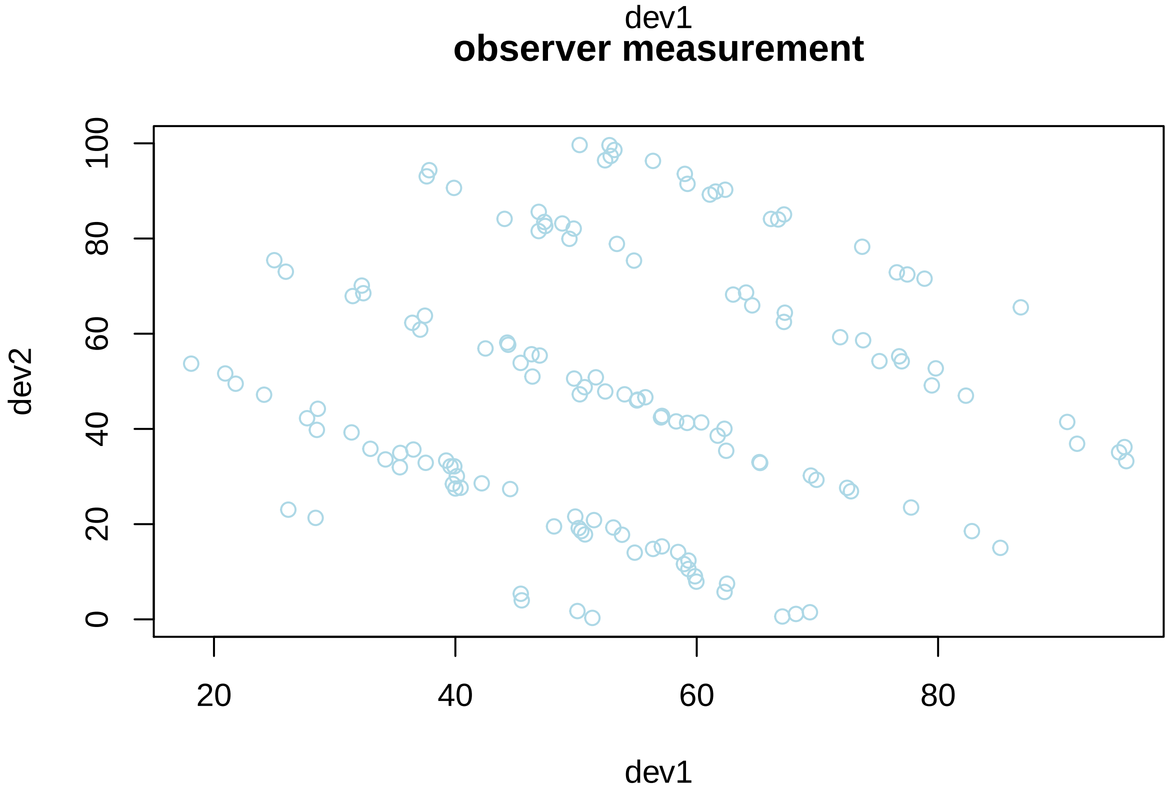

observer measurement
dev2
dev1
20
30
40
50
60
70
80

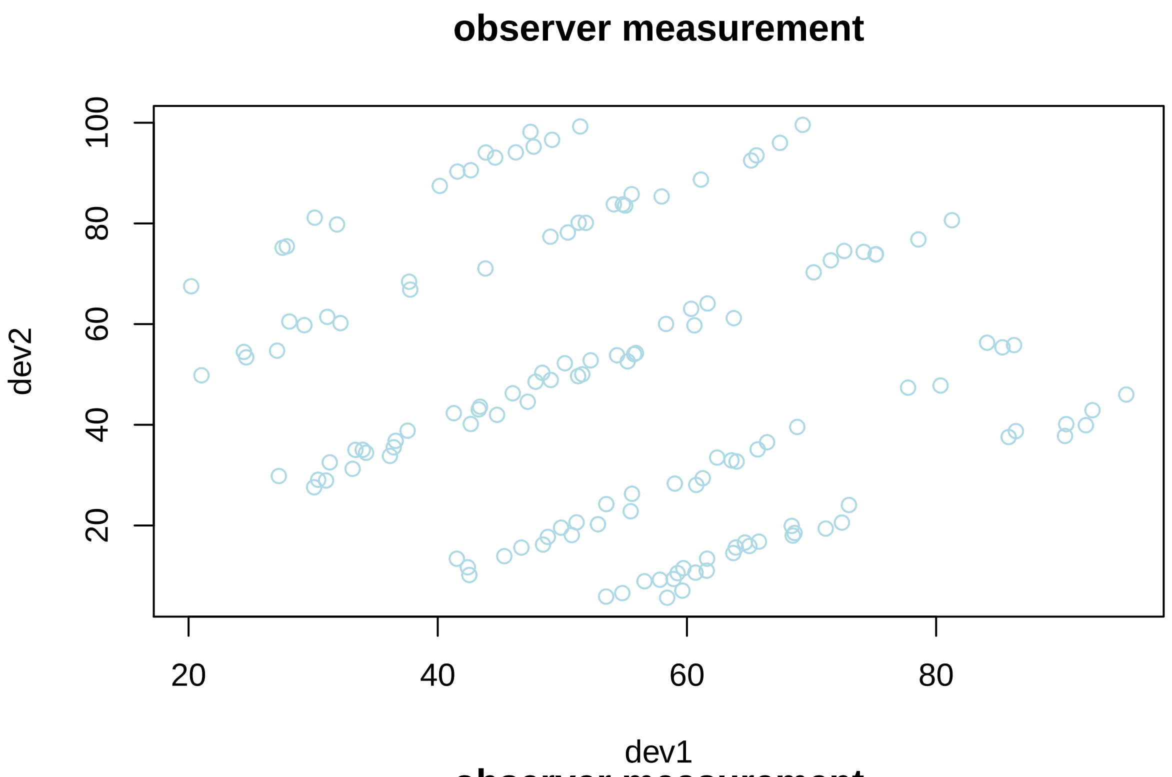

observer measurement
dev2
dev1
0
20
40
60
80
100

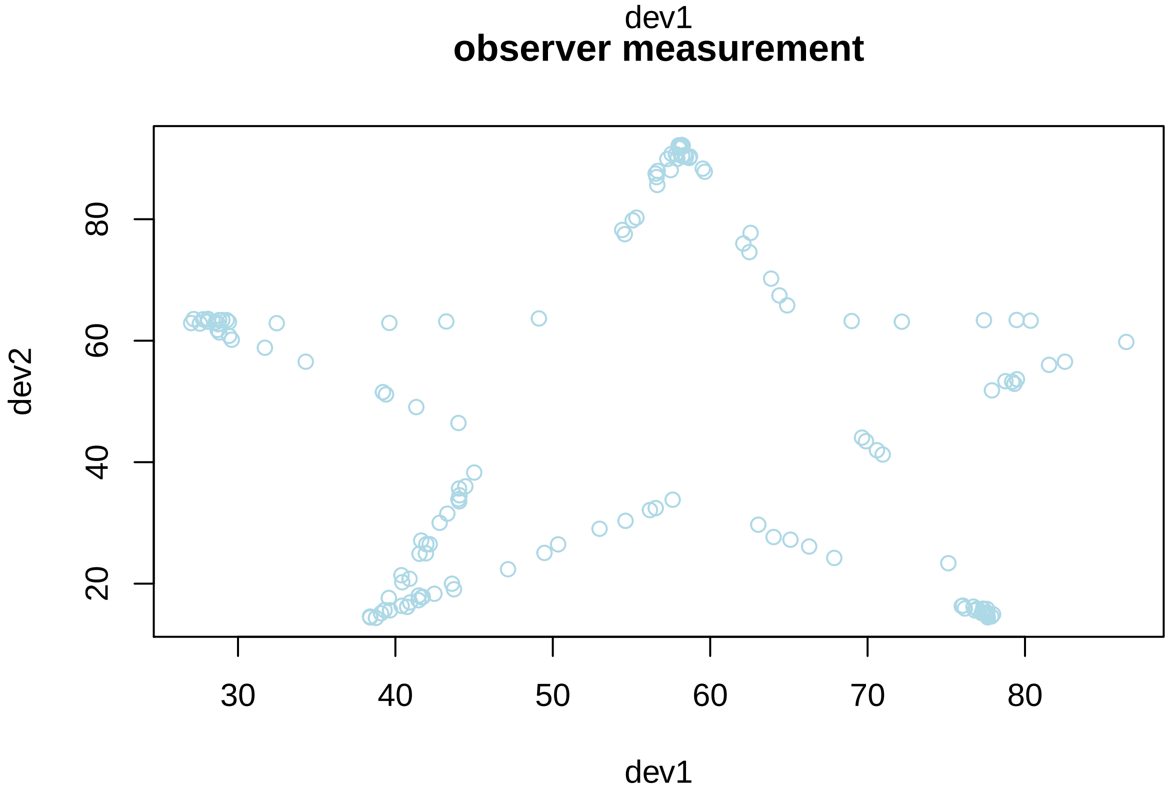

observer measurement
dev2
80
60
40
20
30
40
50
60
70
80
dev1

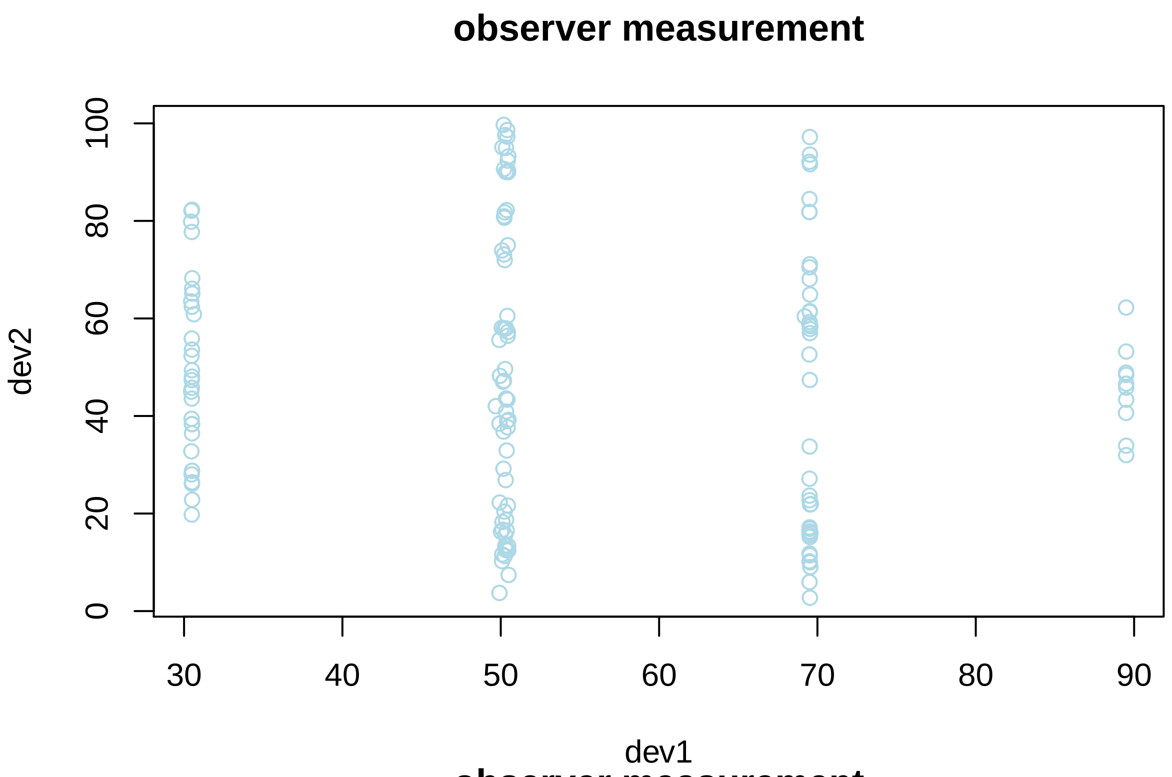

observer measurement
dev2
80
60
40
20
20
40
60
80
100
dev1

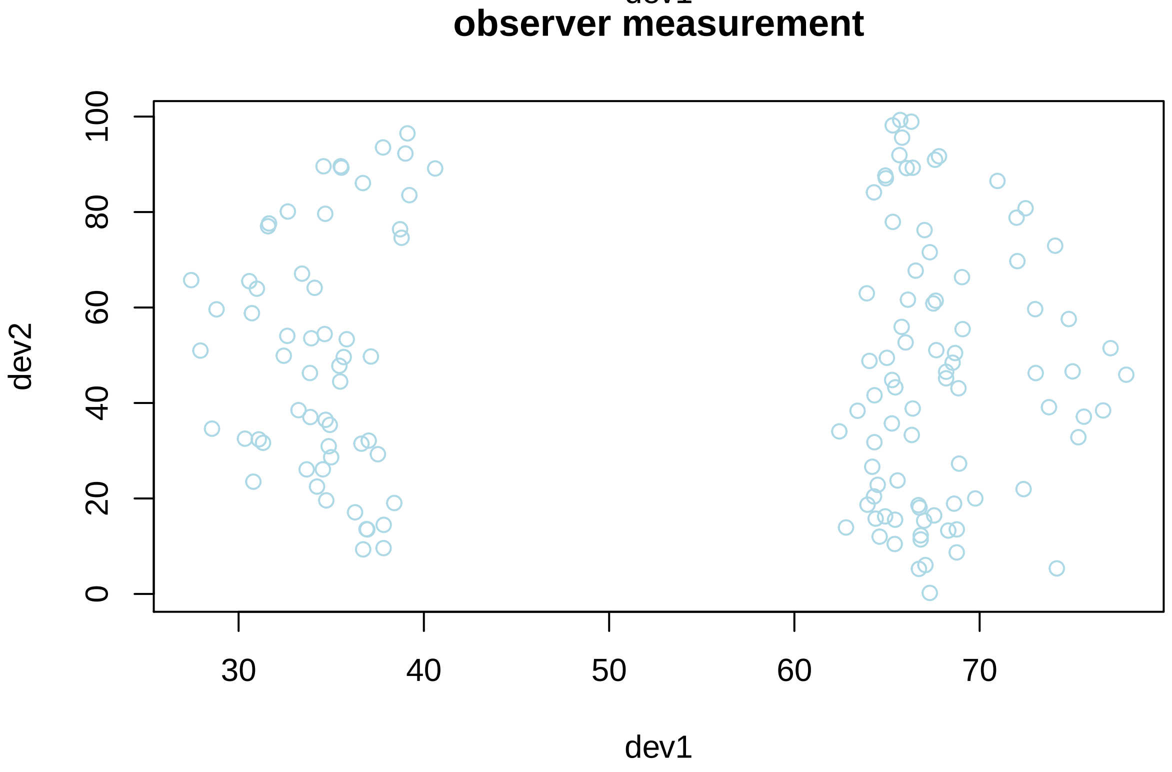

observer measurement
dev2
dev1
20
40
60
80
20
40
60
80

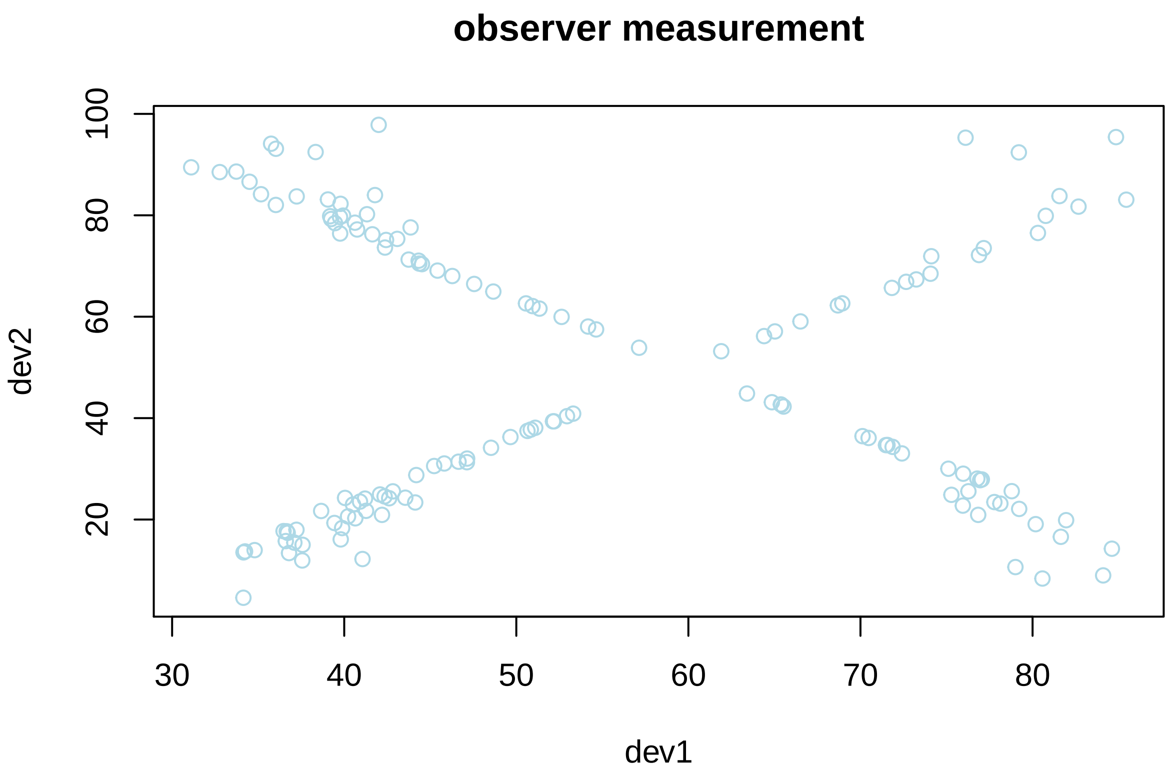

observer measurement
dev2
dev1
0
20
40
60
80
100
20
40
60
80


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


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


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


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



```
## [[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 7

Part a

```r
download("http://www.farinspace.com/wp-content/uploads/us_cities_and_states.zip",
  dest = "us_cities_states.unzip")
unzip("us_cities_states.unzip", exdir = "D:/VT/Rstudio/directory")
states <- fread(input = "D:/VT/Rstudio/directory/us_cities_and_states/states.sql",
  skip = 23, sep = "'", sep2 = ",", header = F, select = c(2, 4))
cities <- fread(input = "D:/VT/Rstudio/directory/us_cities_and_states/cities_extended.sql",
  skip = 26, sep = "'", sep2 = ",", header = F, select = c(2, 4, 6, 8, 10, 12))
colnames(cities) <- c("City", "State_Code", "Zip", "Latitude", "Longitude", "County")
```

part b

We show the first five frequency of cities for states.

```r
city.count <- table(cities$State_Code)
head(city.count)
```

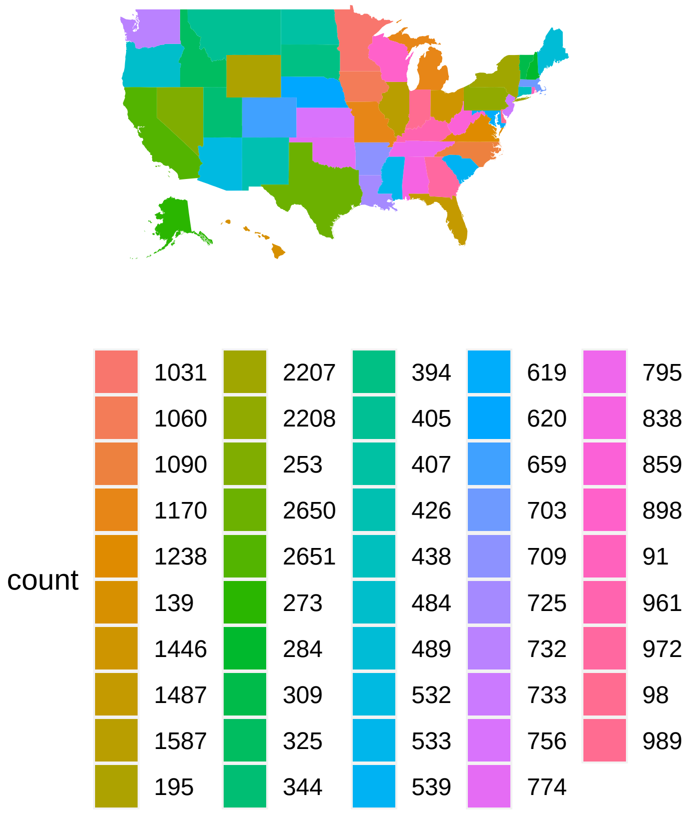
```
##
##   AK   AL   AR   AZ   CA   CO
##  273  838  709  532 2651  659
```

part c

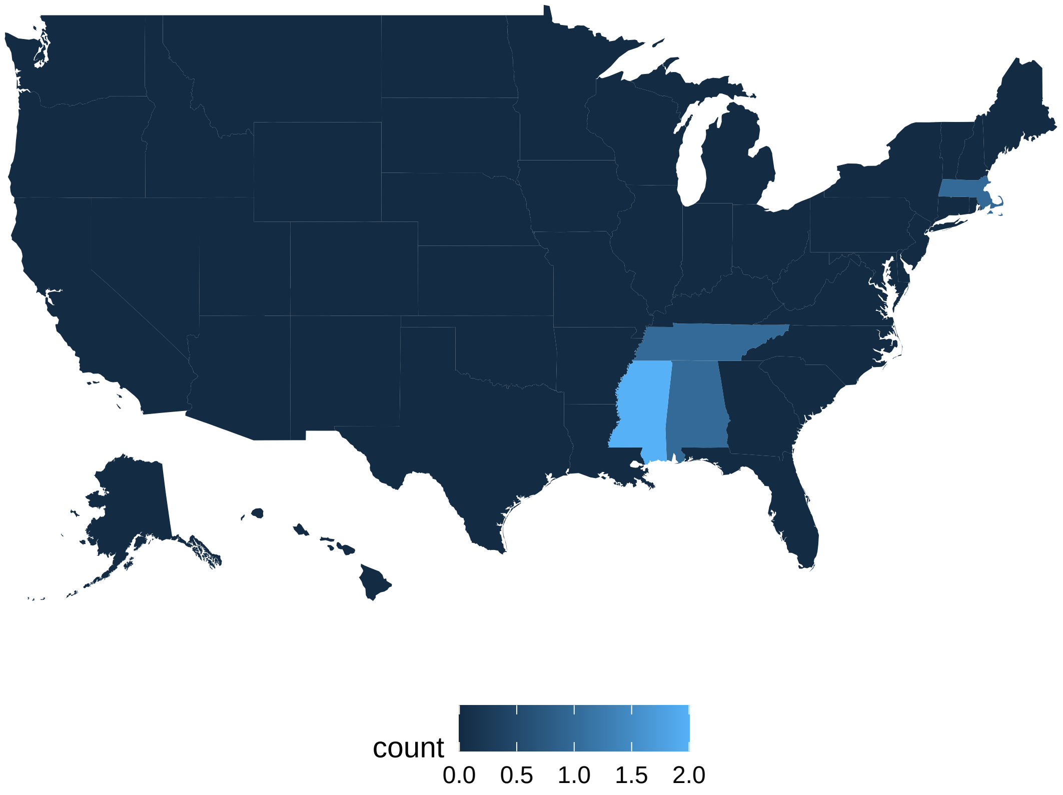
```r
getCount <- function(state_name, letter) {
  temp <- unlist(strsplit(tolower(state_name), split = ""))
  count <- 0
  for (i in 1:length(temp)) {
    if (temp[i] == letter) {
      count <- count + 1
    }
  }
  return(count)
}
letter.count <- data.frame(matrix(NA, nrow = 51, ncol = 26))
letter.count.results <- for (i in 1:51) {
  letter.count[i, ] <- sapply(1:26, function(n) {
    getCount(states$V2[i], letters[n])
  })
}
```

part d

```
# https://cran.r-project.org/web/packages/fiftystater/vignettes/fiftystater.html
# crimes <- data.frame(state = tolower(rownames(USArrests)), USArrests) map_id
# creates the aesthetic mapping to the state name column in your data
library(remotes)
library(fiftystater)

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
p <- ggplot(city.count2, aes(map_id = state)) + geom_map(aes(fill = count), map = fifty_states) +
  expand_limits(x = fifty_states$long, y = fifty_states$lat) + coord_map() + scale_x_continuous(breaks =
  scale_y_continuous(breaks = NULL) + labs(x = "", y = "") + theme(legend.position = "bottom",
  panel.background = element_blank())
p
```



```
state.letter <- data.frame(state = tolower(states$V2), rowSums(letter.count > 3))
colnames(state.letter) <- c("state", "count")
p2 <- ggplot(state.letter, aes(map_id = state)) + geom_map(aes(fill = count), map = fifty_states) +
  expand_limits(x = fifty_states$long, y = fifty_states$lat) + coord_map() + scale_x_continuous(breaks =
  scale_y_continuous(breaks = NULL) + labs(x = "", y = "") + theme(legend.position = "bottom",
  panel.background = element_blank())
p2
```



Problem 8

part a

The re-sampling result of each iteration is the same. So we have to correct the code. The author mistype the "Boot" as "Boot_times". "logapple08" and "logrm08" are not the variable names of "df08". When sampling, there is no need to use argument: "replace=TRUE".

```
set.seed(19941028)
# 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)
colnames(df08) <- c("logapple08", "logrm08")  #Add one line here
```

```r
Boot.times <- 1000
sd.boot <- rep(0, Boot.times)
for (i in 1:Boot.times) {
  bootdata <- df08[sample(nrow(df08), size = 251, replace = TRUE), ]
  sd.boot[i] <- coef(summary(lm(logapple08 ~ logrm08, data = bootdata)))[2, 2]
}
```

part b

```r
# import the sensory data
set.seed(19941028)
library(data.table)
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)
# Bootstrap using the sensory data
time1 <- system.time(for (i in 1:Boot.times) {
  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
##   0.862   0.013   0.877
```

parb c

```r
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)
```

```r
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
stopCluster(cl)
```

Problem 9

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
fun <- function(x = 1) {
  f <- 3^x - sin(x) + cos(5 * x)
  return(f)
}
dr <- function(x = 1) {
  d <- (3^x) * log(3, base = exp(1)) - cos(x) - 5 * sin(5 * x)
}
newton.method <- function(m = 1) {
  tolerance <- 0.5
  while (abs(fx(m)) > tolerance) {
    new.point <- m - (fun(m)/dr(m))
    m <- new.point
  }
  return(m)
}
vector <- seq(-1, 1, by = 0.002)
# Using on of the apply functions and find the roots system.time(result <-
# lapply(vec, newton))
```

part b

We use 8 workers to accelerate computation speed.

```r
cores <- 8
cl <- makeCluster(cores)
clusterExport(cl, list("fx", "dr"))
vecctor <- seq(-1, 1, by = 0.001)
# Using on of the apply functions and find the roots
system.time(result <- parSapply(cl, vector, newton.method))
stopCluster(cl)
```