Homework 4

Youhui Ye

10/5/2020

Problem 1

```r
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))
## initialization for loop
m <- nrow(X)
theta0old <- Inf
theta1old <- Inf
theta0new <- 0
theta1new <- 0
alpha <- 0.05
tol <- 1e-05
while ((abs(theta0old - theta0new) > tol) || (abs(theta1old - theta1new) > tol)) {
  theta0old <- theta0new
  theta1old <- theta1new
  pred <- X[,1] * theta0old + X[, 2] * theta1old
  theta0new <- theta0old - alpha / m * sum(pred - h)
  theta1new <- theta1old - alpha / m * sum((pred - h) * X[,2])
}
cat("Tolerance: ", tol)
```

```
## Tolerance:  1e-05
```

```r
cat("Step size: ", alpha)
```

```
## Step size:  0.05
```

```r
cat("estimated theta0: ", theta0new)
```

```
## estimated theta0:  0.968629
```

```r
cat("estimated theta1: ", theta1new)
```

```
## estimated theta1:  2.001698
```

```r
m1 <- lm(h ~ 0 + X)
m1$coefficients
```

```
##        X1        X2
## 0.9695707 2.0015630
```

The answer is very close to what “lm” function gives.

user	system	elapsed
20.56	11.47	72362.59

Problem 2

Part a

```r
## set up
theta0s <- seq(0, 2, length.out = 100)
theta1s <- seq(1, 3, length.out = 100)
thetas <- expand.grid(theta0s, theta1s)
## wrap up the function to implement parallel computing
init <- thetas[1,]
my_gradient_descent <- function(init, X, h) {
  ## set up
  m <- 100
  alpha <- 1e-07
  tol <- 1e-09
  theta0old <- Inf
  theta1old <- Inf
  theta0new <- as.numeric(init[1])
  theta1new <- as.numeric(init[2])
  ## iteration time
  i <- 0
  while ((abs(theta0old - theta0new) > tol) || (abs(theta1old - theta1new) > tol)) {
    theta0old <- theta0new
    theta1old <- theta1new
    pred <- X[,1] * theta0old + X[, 2] * theta1old
    theta0new <- theta0old - alpha / m * sum(pred - h)
    theta1new <- theta1old - alpha / m * sum((pred - h) * X[,2])
    i <- i + 1
    if(i > 5e06) break
  }
  print("yes")
  return(c(theta0new, theta1new, i))
}
## Using parallel computing
library(parallel)
cores <- detectCores() - 1
## log.txt to trace how many runs already completed
cl <- makeCluster(cores, outfile = "log.txt")
clusterExport(cl, "X")
clusterExport(cl, "h")
system.time(result <- parApply(cl, thetas, 1, my_gradient_descent, X, h))
stopCluster(cl)
```

```r
min(result[3,]) - 1
```

```
## [1] 633612
```

```r
max(result[3,]) - 1
```

```
## [1] 5e+06
## beta_0 estimation
mean(result[1,])
```

```
## [1] 0.9966097
```

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

```
## [1] 0.5197305
```

```
## beta_1 estimation
mean(result[2,])
```

```
## [1] 1.997679
```

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

```
## [1] 0.07465446
```

Part b

I do not think it is a good stopping rule. The problem is that the loop may never stop when it reaches a local

Part c

The algorithm is better used for smooth function. When the function value varies vigorously and the step size is too small, it is very likely to get stuck in a local minimum. Also, there is a lot of work in choosing start values.

Problem 3

```
beta_hat <- solve(t(X) %*% X) %*% t(X) %*% h
```

Least square estimation is subject to find minimizer of $(y - X\hat{\beta})'(y - X\hat{\beta})$. If we take derivative with respect to this function, we have $-2X'(y - X\hat{\beta}) = 0$. The answer is exactly \$ (X'X)^{-1}X'y \$.

Problem 4

```
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

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

```
## 112347224 bytes
```

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

```
## 1816357208 bytes
```

```r
## around 15 mins
system.time(y <- p + A %*% solve(B) %*% (q - r))
```

```
##    user  system elapsed
##  748.65   12.80  773.78
```

Part b

I think the inverse of matrix can be completed independently and in a different way. And the multiplication of A and inverse of B can be completed faster.

Part c

```r
## Use c++ code to speed up computing speed
require(RcppEigen)
require(inline)

## matrix multiplication by c++
txt <- "
using Eigen::Map;
using Eigen::MatrixXd;
using Rcpp::as;
NumericMatrix tm22(tm2);
NumericMatrix tmm(tm);
const MatrixXd ttm(as<MatrixXd>(tmm));
const MatrixXd ttm2(as<MatrixXd>(tm22));

MatrixXd prod = ttm*ttm2;
return(wrap(prod));
"

mul_cpp <- cxxfunction(signature(tm="NumericMatrix",
                                 tm2="NumericMatrix"),
                                 plugin="RcppEigen",
                                 body=txt)

## matrix inversion by c++
txt2 <- "
using namespace Rcpp;
using Eigen::Map;
using Eigen::VectorXd;
using Eigen::MatrixXd;
typedef Map<MatrixXd> MapMatd;
const MapMatd tmm(as<MapMatd>(tm));
const MatrixXd tmm_inv = tmm.inverse() ;
return( wrap(tmm_inv));"

solve_cpp <- cxxfunction(signature(tm="NumericMatrix"),
                 plugin="RcppEigen",
                 body=txt2)
```

```
## around 13.25 mins, slightly faster
system.time({
  B_inv <- solve_cpp(B)
  y <- p + mul_cpp( mul_cpp(A, B_inv), as.matrix(q - r))
})
```

Using R built-in function takes up 15 minutes to calculate the theta_hat. However, if I use RcppEigen library, it takes 13 minutes, which is faster.

Problem 5

a

```
compute_proportion <- function(vec) {
  sum(vec) / length(vec)
}
```

b

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

c

```
## calculate by row
apply(P4b_data, 1, compute_proportion)
```

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

```
## calculate by column
apply(P4b_data, 2, compute_proportion)
```

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

It is not generating 10 independent samples but replicate one sample 10 times.

d

```
generate_flips <- function(prob) {
  rbinom(10, 1, prob)
}
probabilities <- matrix((31:40) / 100, ncol = 10)
P4d_data <- apply(probabilities, 2, generate_flips)
## calculate by row
apply(P4d_data, 1, compute_proportion)
```

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

```
## calculate by column
apply(P4d_data, 2, compute_proportion)
```

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

Problem 6

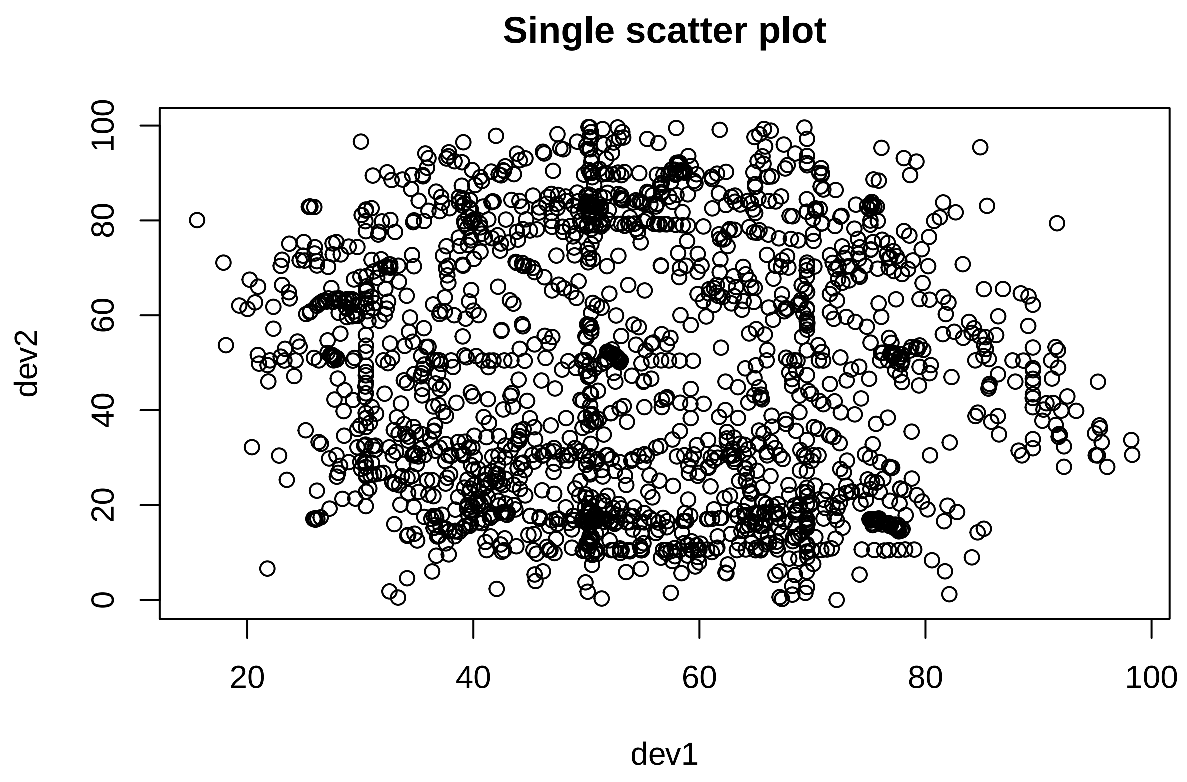
```r
dat6 <-readRDS("HW3_data.rds")
colnames(dat6) <- c("Observer", "x", "y")
```

1.

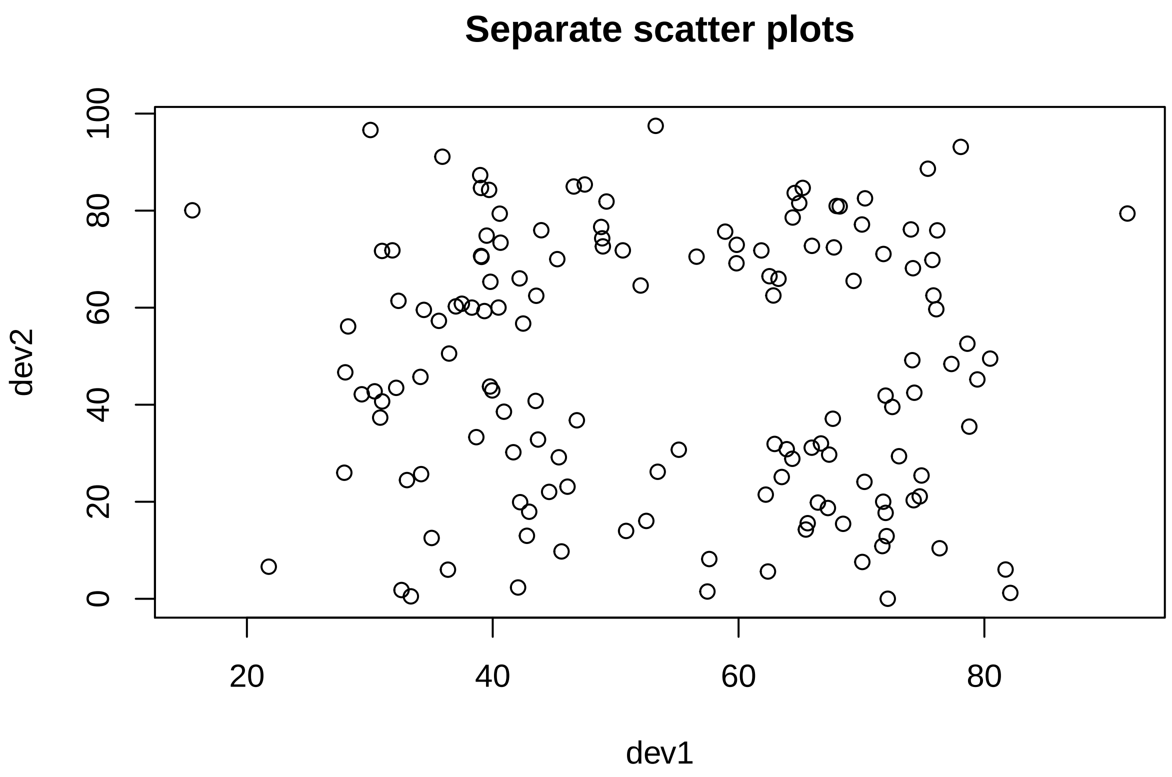
```r
## coordinates values must be names as "x" and "y"
my_plot <- function(dat, title, xlab, ylab) {
  plot(dat$x, dat$y, main = title, xlab = xlab, ylab = ylab)
}
```

2.

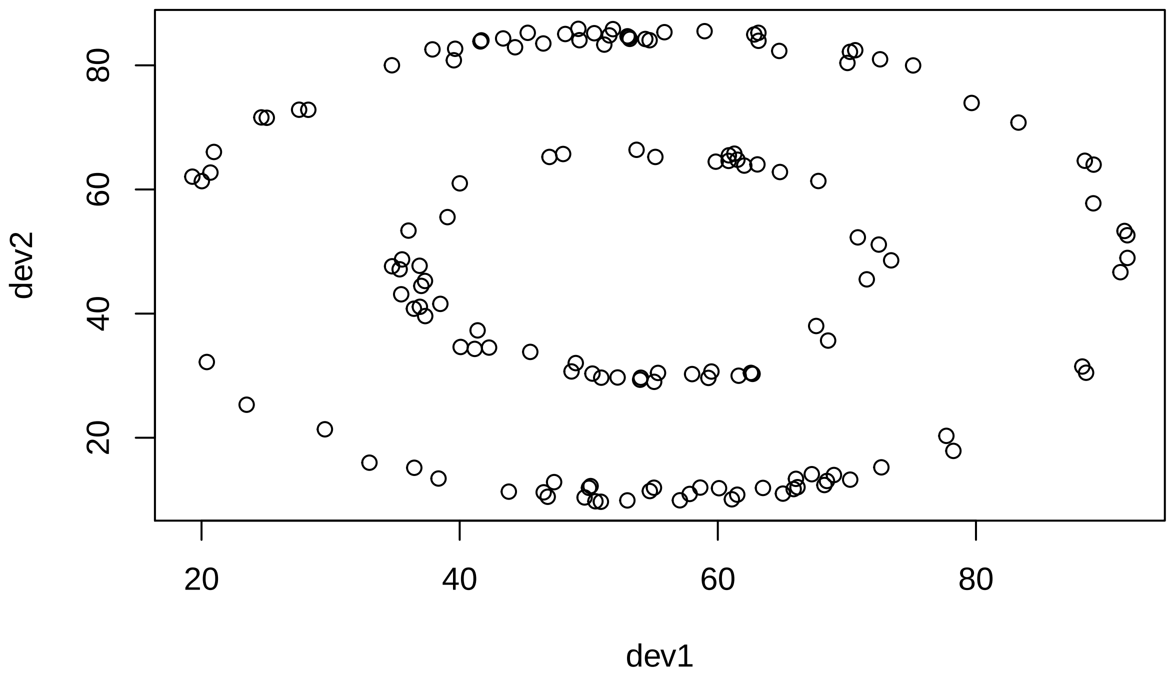
```r
## (a)
my_plot(dat6, "Single scatter plot", "dev1", "dev2")
```

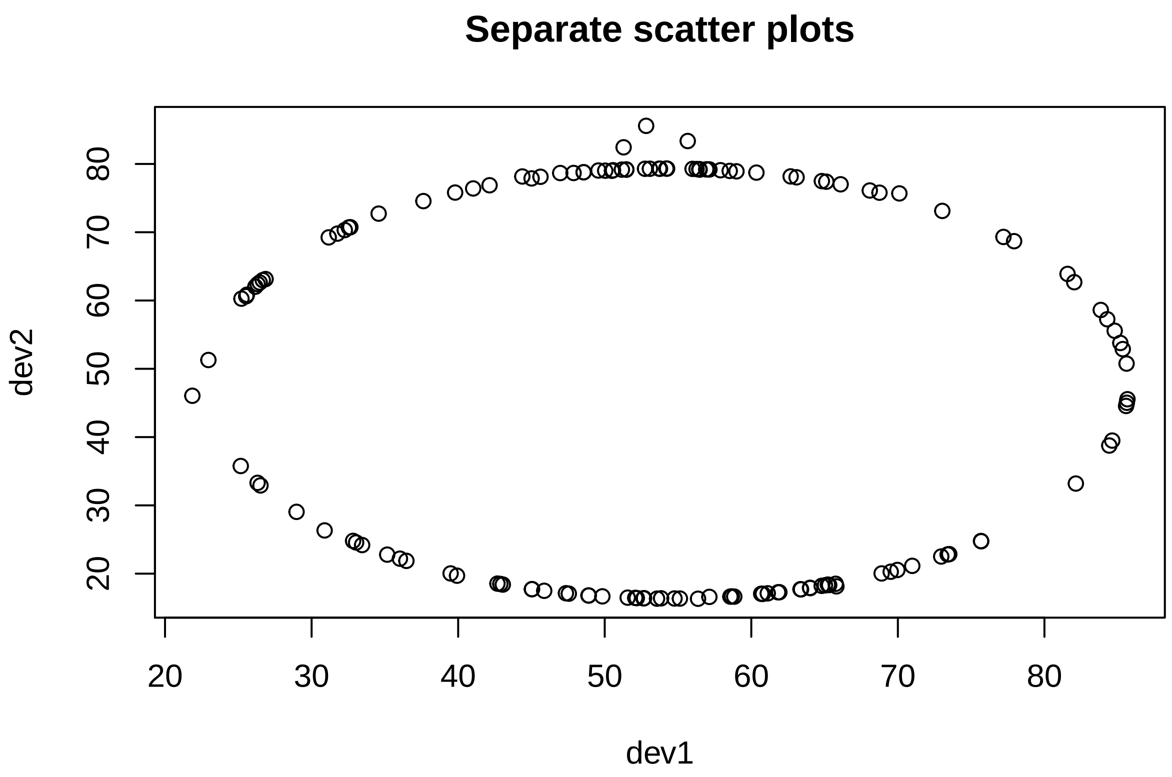


```r
## (b)
dat6_by_observer = split(dat6, f = dat6$Observer)
trash_can <- lapply(dat6_by_observer, my_plot, "Separate scatter plots", "dev1", "dev2")
```

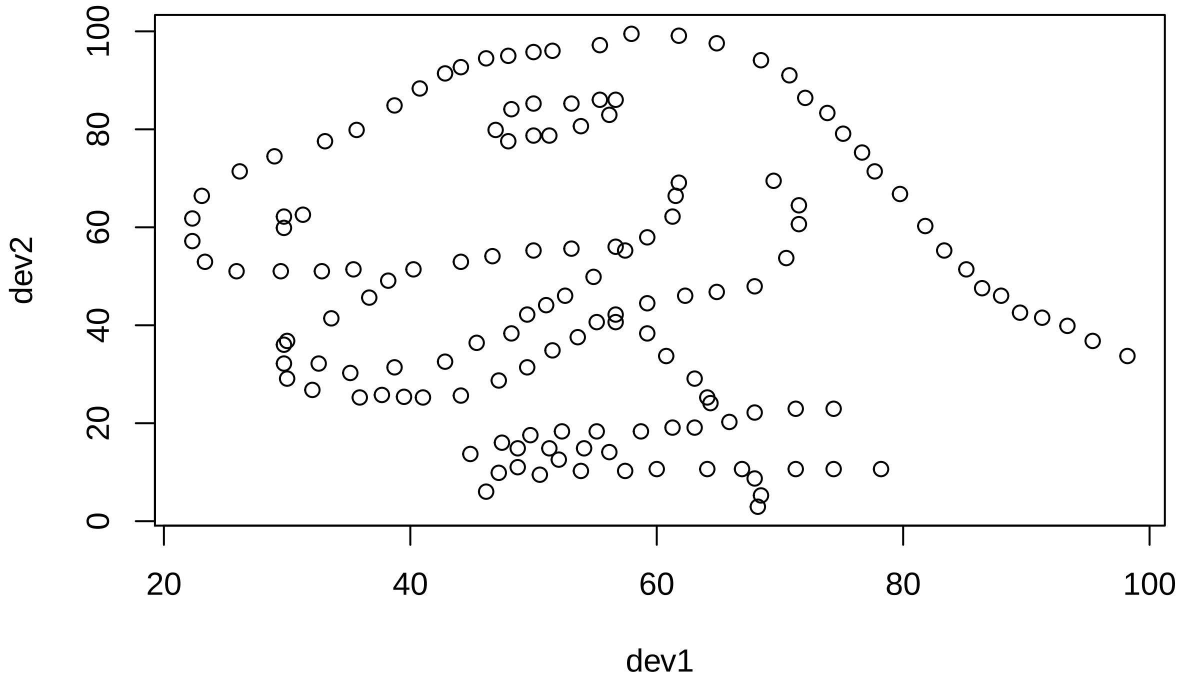

Separate scatter plots
dev2
0
20
40
60
80
100
20
40
60
80
dev1

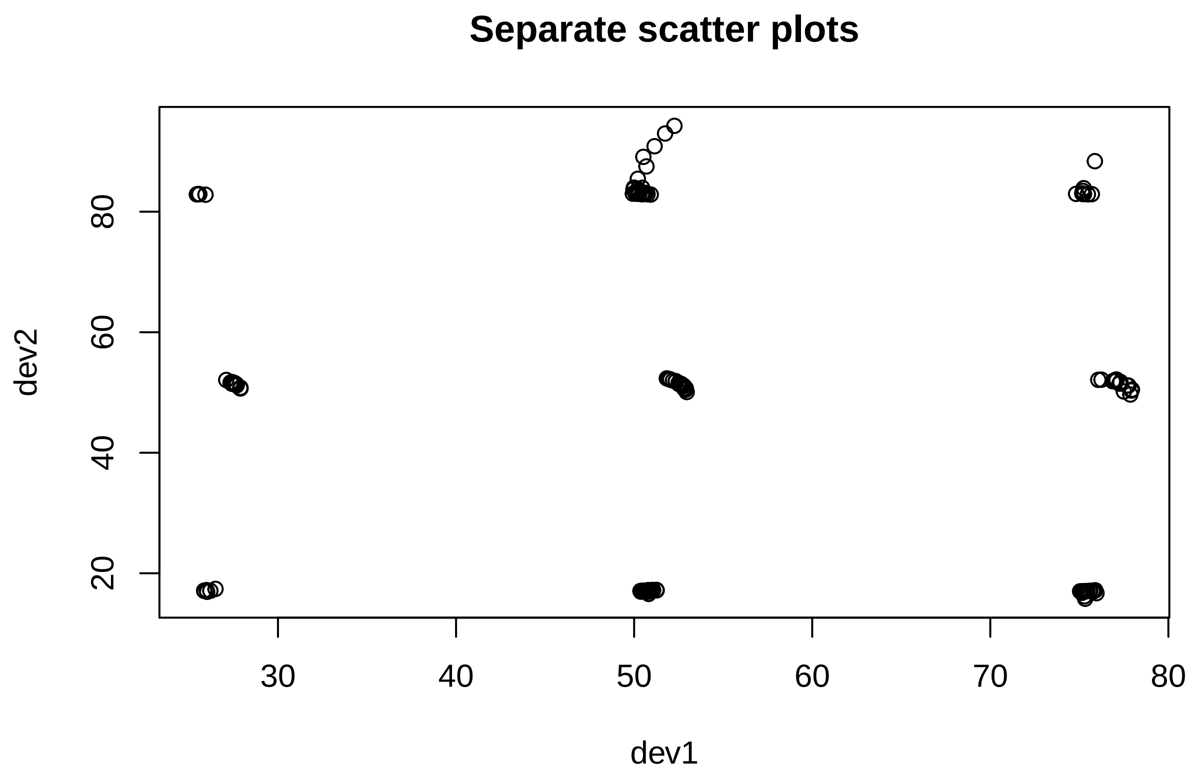
Separate scatter plots

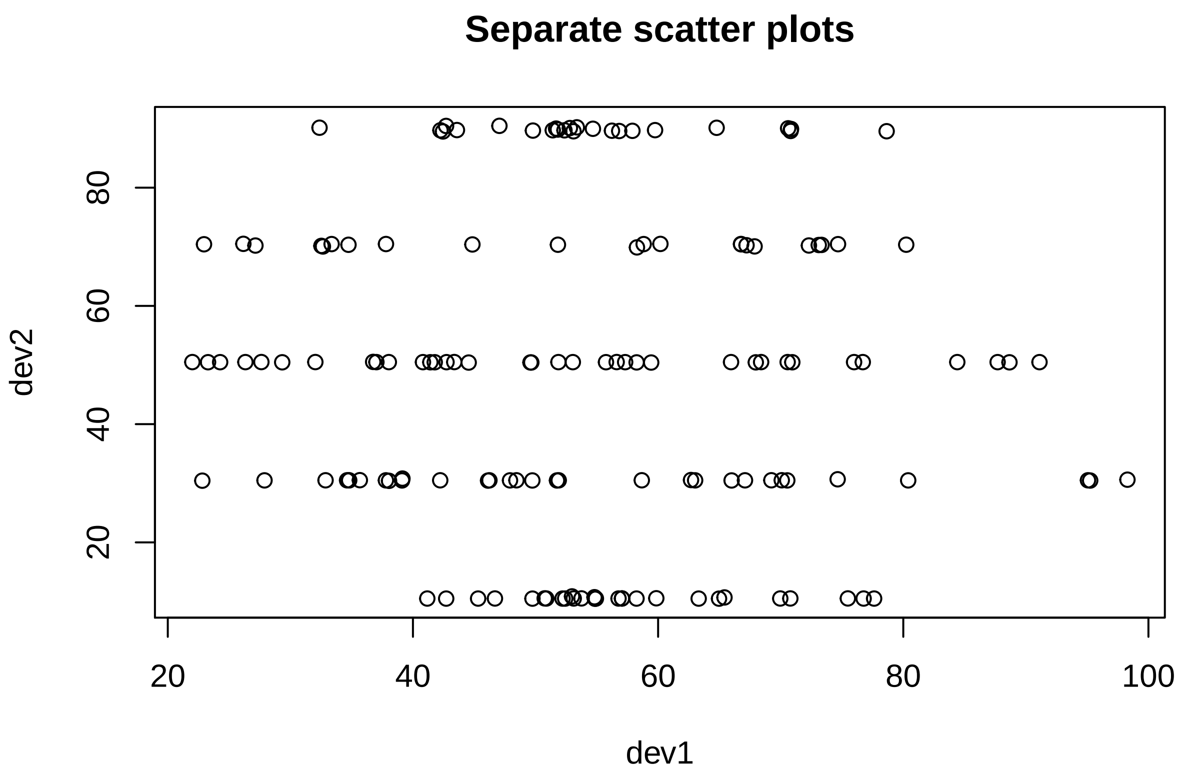


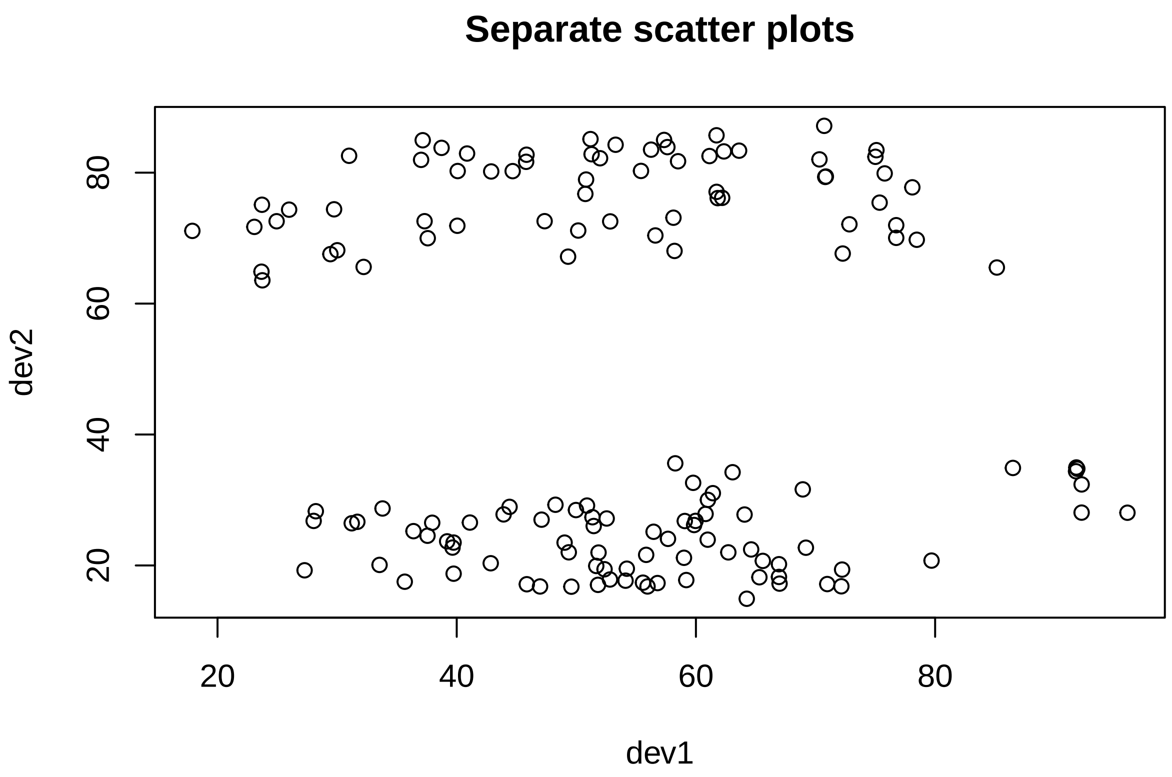

Separate scatter plots
dev2
dev1
20
30
40
50
60
70
80

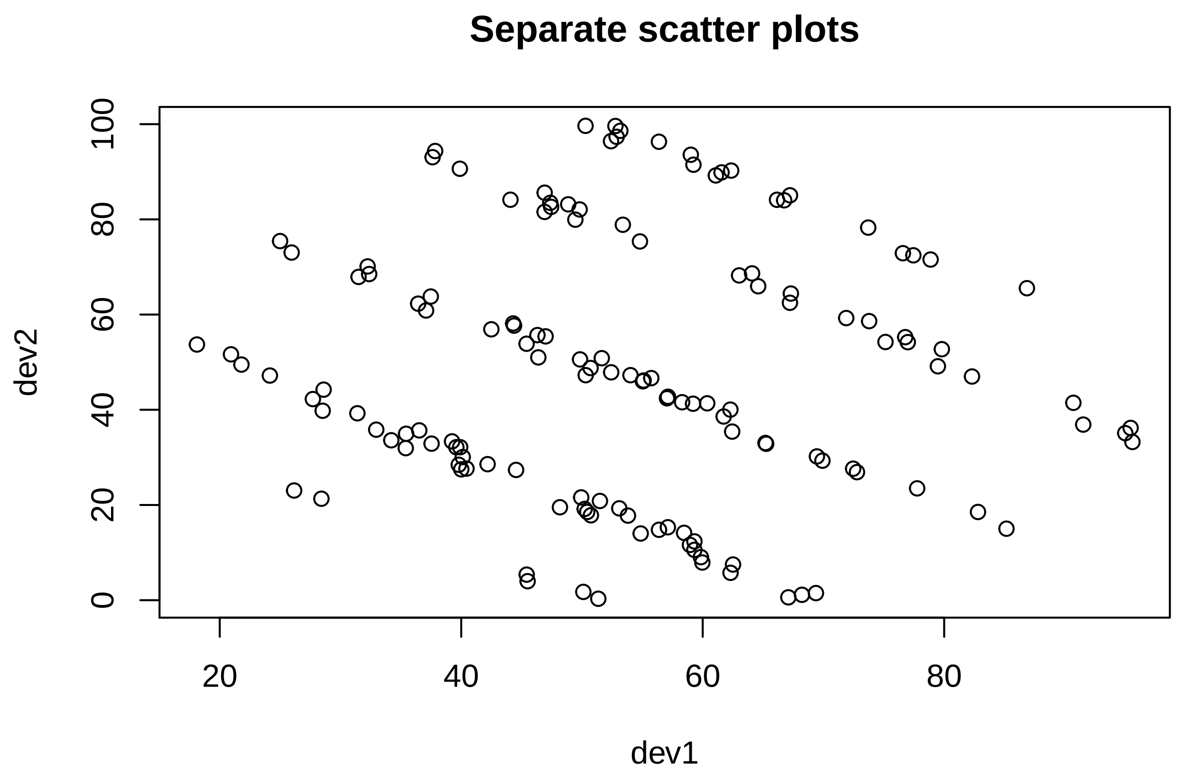
Separate scatter plots

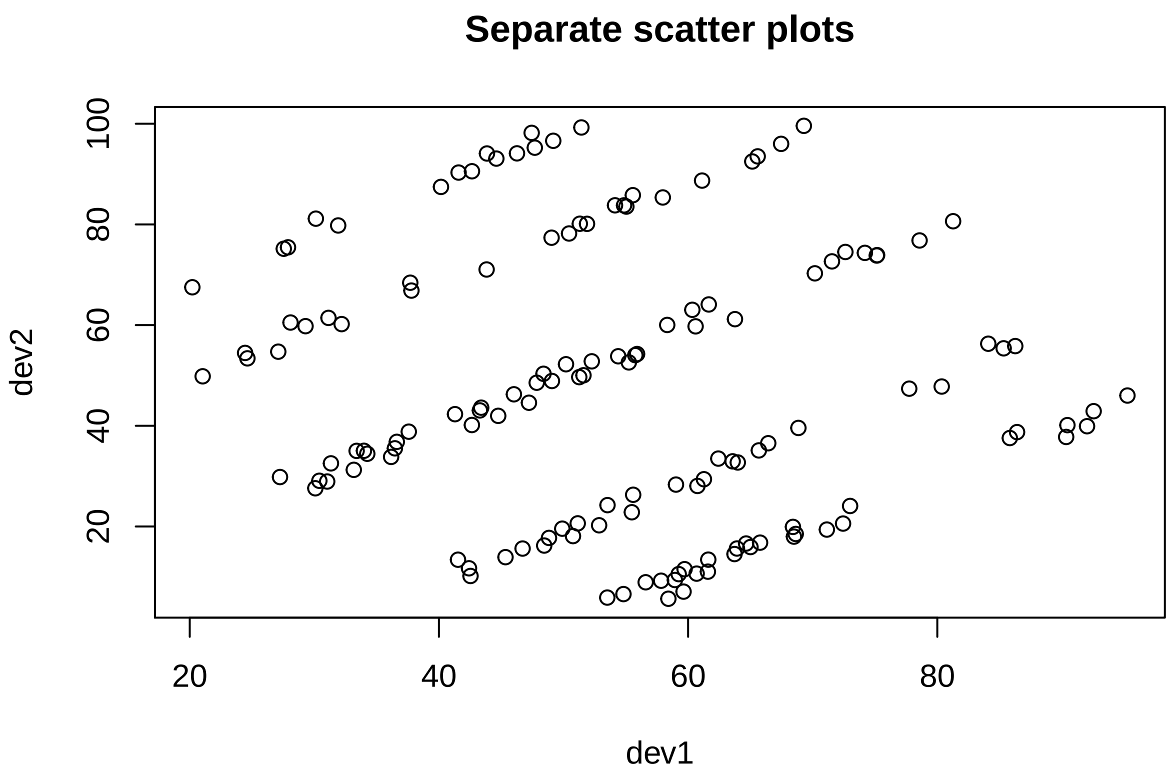


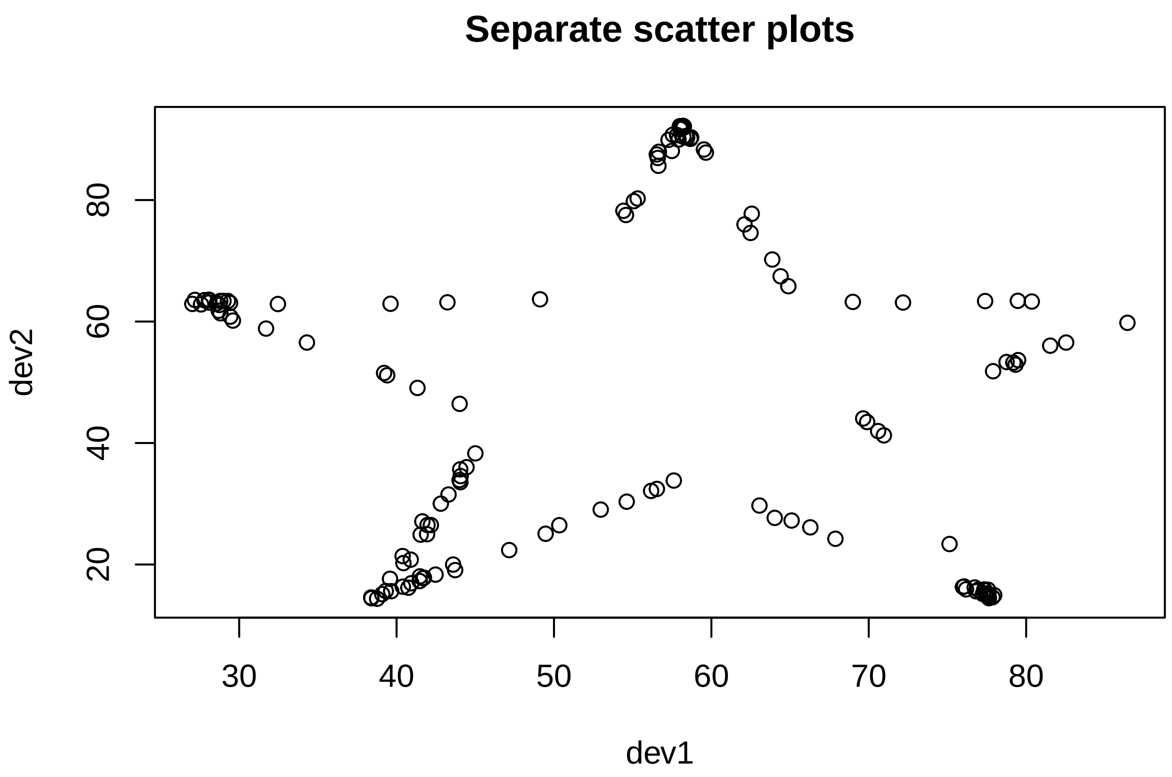

Separate scatter plots
dev2
80
60
40
20
30
40
50
60
70
80
dev1

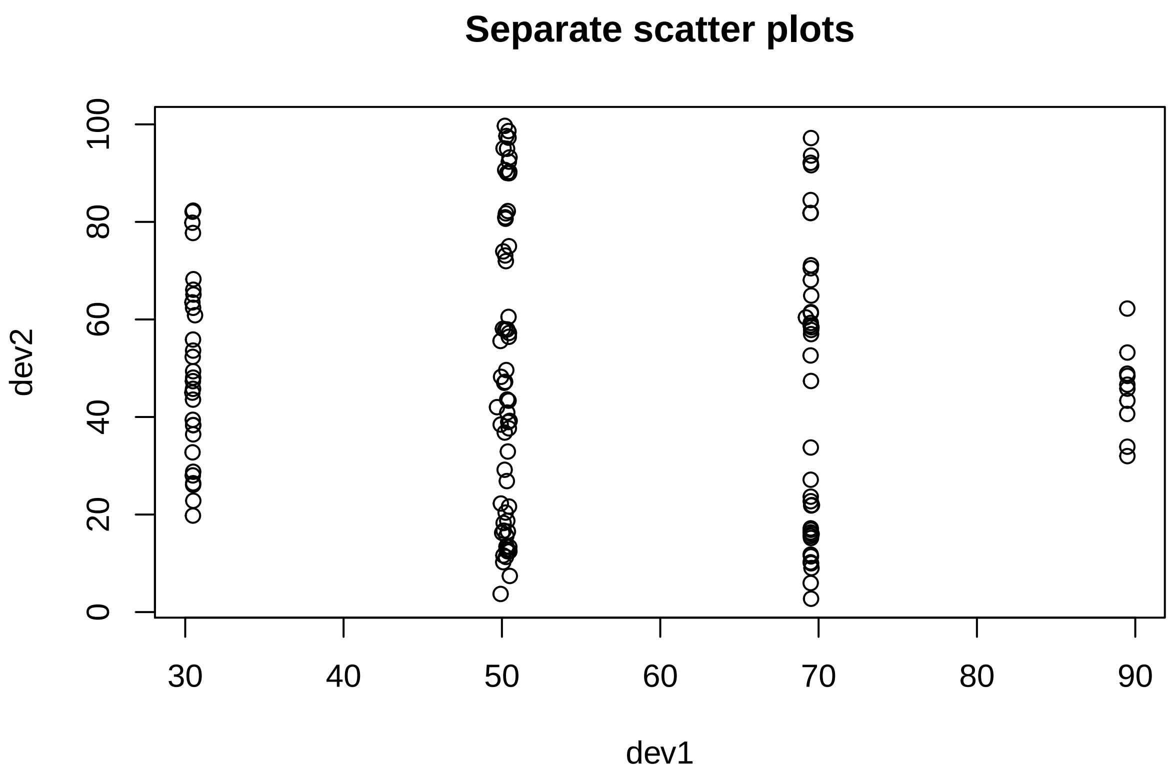

Separate scatter plots
dev2
80
60
40
20
20
40
60
80
100
dev1

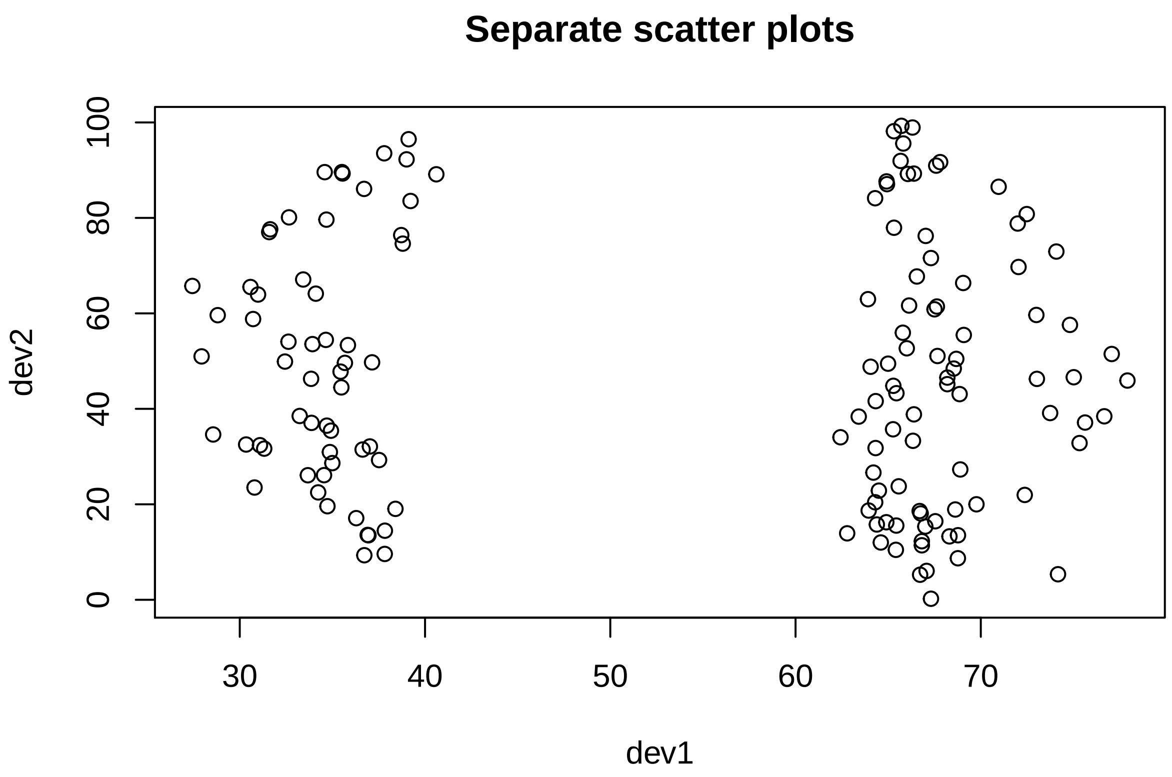

Separate scatter plots
dev2
dev1
20
40
60
80
20
40
60
80

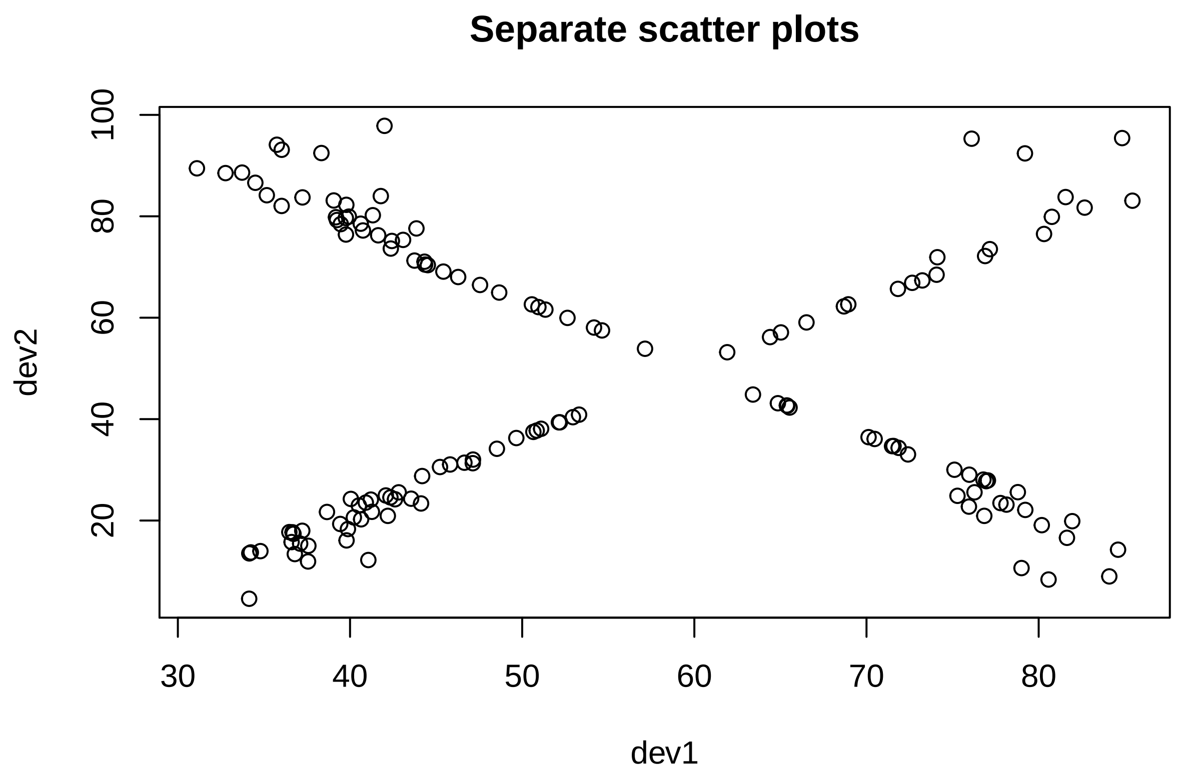

Separate scatter plots
dev2
dev1
0
20
40
60
80
100
20
40
60
80


Separate scatter plots
dev2
dev1
20
40
60
80
100


Separate scatter plots
dev2
dev1
20
40
60
80
30
40
50
60
70
80


Separate scatter plots
dev2
dev1
0
20
40
60
80
100
30
40
50
60
70
80
90


Separate scatter plots
dev2
dev1
0
20
40
60
80
100
30
40
50
60
70



Problem 7

Part a

```
#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
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 = F
states <- states[-8,]
cities <- fread(input = "./us_cities_and_states/cities_extended.sql",sep = "'", sep2 = ",", header = FA

### YOU do the CITIES
### I suggest the cities_extended.sql may have everything you need
### can you figure out how to limit this to the 50?
```

Part b

```
## remove DC and PR
city_counts <- data.frame(table(cities$V4)[c(-8, -40)])
```

```
state_city_counts <- data.frame(city_counts, state = tolower(states$V2))
state_city_counts
```

```
##    Var1 Freq          state
## 1    AK  273         alaska
## 2    AL  838        alabama
## 3    AR  709       arkansas
## 4    AZ  532        arizona
## 5    CA 2651     california
## 6    CO  659       colorado
## 7    CT  438    connecticut
## 8    DE   98       delaware
## 9    FL 1487        florida
## 10   GA  972        georgia
## 11   HI  139         hawaii
## 12   IA 1060           iowa
## 13   ID  325          idaho
## 14   IL 1587       illinois
## 15   IN  989        indiana
## 16   KS  756         kansas
## 17   KY  961       kentucky
## 18   LA  725      louisiana
## 19   MA  703  massachusetts
## 20   MD  619       maryland
## 21   ME  489          maine
## 22   MI 1170       michigan
## 23   MN 1031      minnesota
## 24   MO 1170       missouri
## 25   MS  533    mississippi
## 26   MT  405        montana
## 27   NC 1090 north carolina
## 28   ND  407   north dakota
## 29   NE  620       nebraska
## 30   NH  284  new hampshire
## 31   NJ  733     new jersey
## 32   NM  426     new mexico
## 33   NV  253         nevada
## 34   NY 2207       new york
## 35   OH 1446           ohio
## 36   OK  774       oklahoma
## 37   OR  484         oregon
## 38   PA 2208   pennsylvania
## 39   RI   91   rhode island
## 40   SC  539 south carolina
## 41   SD  394   south dakota
## 42   TN  795      tennessee
## 43   TX 2650          texas
## 44   UT  344           utah
## 45   VA 1238       virginia
## 46   VT  309        vermont
## 47   WA  732     washington
## 48   WI  898      wisconsin
## 49   WV  859  west virginia
## 50   WY  195        wyoming
```

Part c

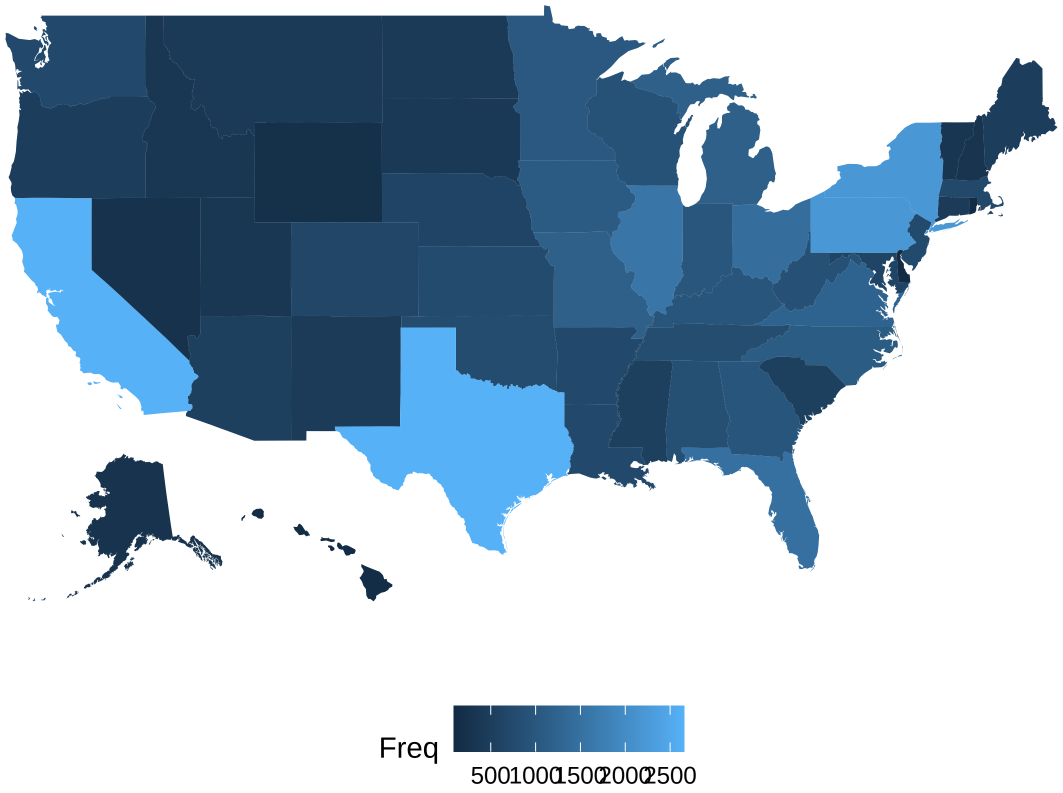
```r
letter_count <- data.frame(matrix(NA,nrow=50, ncol=26))
getCount <- function(letter, state_name){
  temp <- strsplit(tolower(state_name), "")
  count <- sum(temp[[1]] == letter)
  # how to count??
  return(count)
}
for(i in 1:50){ letter_count[i,] <- lapply(letters, getCount, states[i, 2]) }
```

Part d

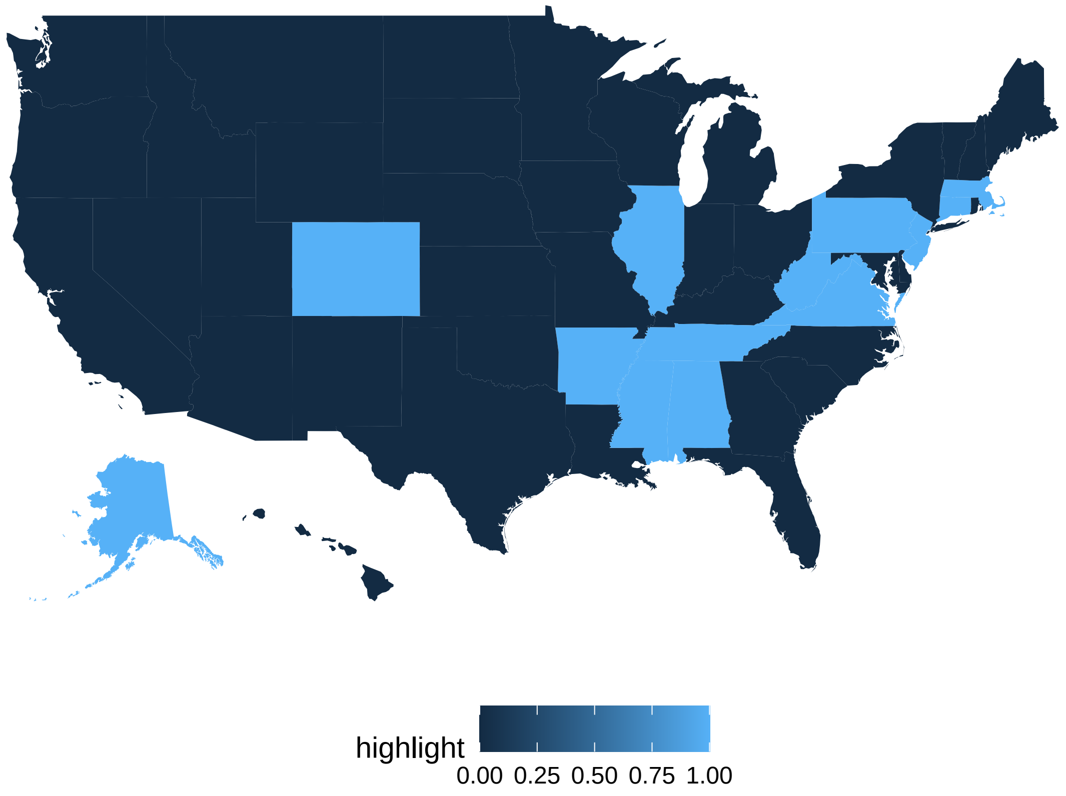
```r
#https://cran.r-project.org/web/packages/fiftystater/vignettes/fiftystater.html
library(ggplot2)
library(mapproj)
```

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

```r
# crimes <- data.frame(state = tolower(rownames(USArrests)), USArrests)
# map_id creates the aesthetic mapping to the state name column in your data
load("fifty_states.rda")
p <- ggplot(state_city_counts, aes(map_id = state)) +
  # map points to the fifty_states shape data
  geom_map(aes(fill = Freq), 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
```



```
letter_highlight <- data.frame(state = tolower(states$V2),
                               highlight = apply(letter_count >=3, 1,
                                           function(x){ifelse(sum(x) > 0, 1, 0)}) )
p2 <- ggplot(letter_highlight, aes(map_id = state)) +
  # map points to the fifty_states shape data
  geom_map(aes(fill = highlight), map = fifty_states) +
  expand_limits(x = fifty_states$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())
p2
```



Problem 8

```r
library(tidyverse)
sensory_data_raw <- readRDS("sensory_data_raw.RDS")
sensory_data_tv <- sensory_data_raw %>%
  separate(col = "Operator",into = c("Item",as.character(1:5)), sep=" ",fill = "left") %>%
  fill("Item") %>%
  slice(2:n()) %>%
  gather(key = "Operator", value = "value", -Item)
```

Part a

When he was “sd.boot[i]= coef(summary(lm(logapple08~logrm08, data = bootdata)))[2,2]”. He used logapple08 and logrm08, which are not the variable names of bootdata but existing vector.

Part b

```r
set.seed(3456)
Boot <- 100
coef_boot <- rep(0,Boot)
ind_box <- matrix(1:150, ncol = 5)
system.time({
  for(i in 1:Boot){
    # nonparametric bootstrap
```

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.	time
apply	-0.329	-0.116	-0.035	-0.040	0.032	0.309	0.09
parApply	-0.298	-0.138	-0.052	-0.037	0.057	0.25	0.08

```r
    ind <- c(apply(ind_box, 2, function(x){sample(x, 30, replace = TRUE)}))
    bootdata <- sensory_data_tv[ind,]
    coef_boot[i]= coef(lm(as.numeric(value) ~ as.numeric(Operator), data = bootdata))[2]
  }
})
```

```
##    user  system elapsed
##    0.08    0.00    0.07
```

Part c

```r
library(foreach)
library(doParallel)


#setup parallel backend to use many processors
cores <- detectCores()
cl <- makeCluster(cores-1) #not to overload your computer
registerDoParallel(cl)
set.seed(3456)
system.time({
  coef_boot_par <- foreach(i=1:100, .combine=cbind) %dopar% {
    ind <- c(apply(ind_box, 2, function(x){sample(x, 30, replace = TRUE)}))
    bootdata <- sensory_data_tv[ind,]
    coef(lm(as.numeric(value) ~ as.numeric(Operator), data = bootdata))[2]
  }
})
```

```
##    user  system elapsed
##    0.04    0.02    0.08
```

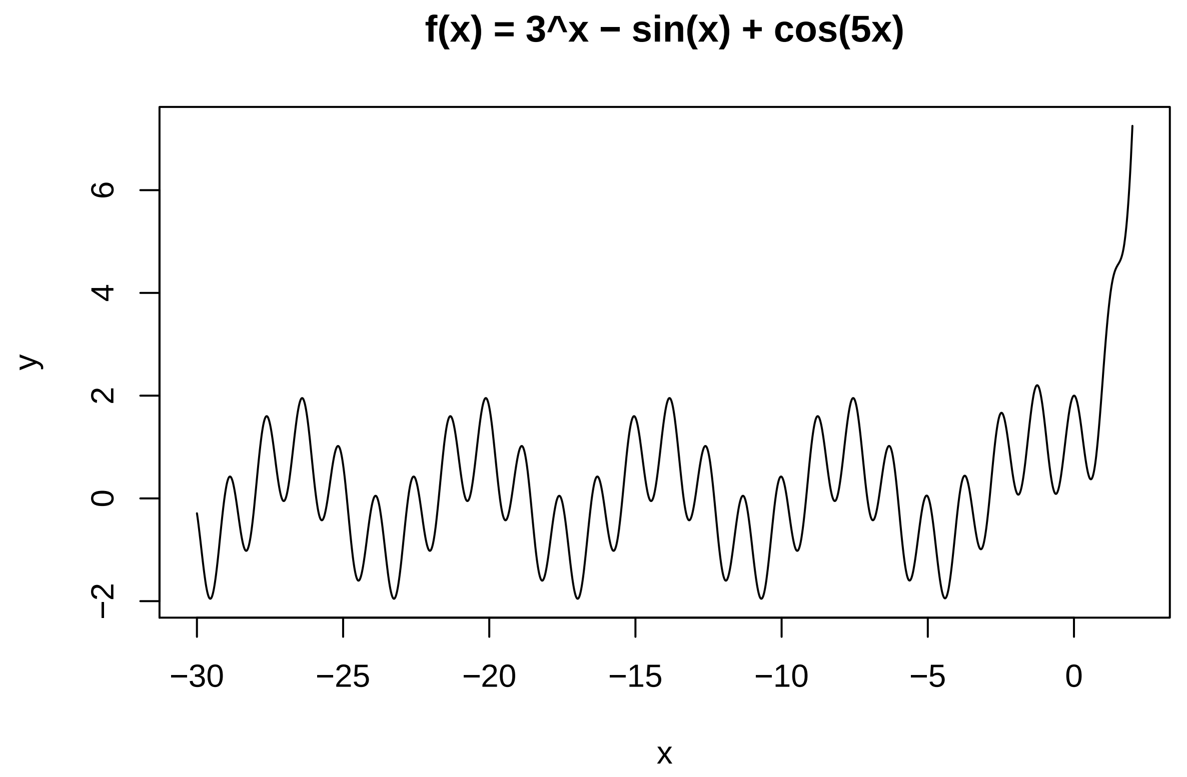
```r
#stop cluster
stopCluster(cl)
```

Since bootstraping samples are independent from each other, it can be completed on different computing node. That is why we can utilize parallel computing method.

I think in this particular question, they give similar result and their running time do not differ too much because it is a relatively small loop.

Problem 9

```r
x <- seq(-30, 2, 0.01)
y <- 3^x - sin(x) + cos(5*x)
plot(x, y, type = 'l', main = "f(x) = 3^x - sin(x) + cos(5x)")
```



This function has countless roots.

Part a

```r
## target function
f <- function(x) {
  value <- 3^x - sin(x) + cos(5*x)
  return(value)
}

## derivative of target function
grad <- function(x) {
  value <- log(3) * 3^x - cos(x) - 5*sin(5*x)
  return(value)
}

find_the_root <- function(interval, tol) {
  ## tolerance is a single value while interval is a vector of dimension 2
  left <- interval[1]
  right <- interval[2]
  initial_point <- (left + right) / 2
  ## use while loop to solve the problem
  prev <- initial_point
  i <- 1
  while (abs(f(prev)) > tol ) {
    ## calculate a new point
```

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.	time
apply	-39.66	-31.15	-22.38	-22.51	-13.35	-4.97	129.64
parApply	-39.66	-31.15	-22.38	-22.51	-13.35	-4.97	20.64

```
    curr <- prev - f(prev) / grad(prev)
    ## if the update goes beyond the given interval, assign limit to it
    if (curr < left) {
      curr <- left
    } else if (curr > right) {
      curr <- right
    }
    ## update the previous point and draw it on the plot
    prev <- curr
    i <- i + 1
    if(i > 1e06) break
  }
  ## track progress
  cat("Yes\n",file="log.txt", append = TRUE)
  return(prev)
}

## try to find all the roots from -40 to -5
start_points <- seq(-40, -5, length.out = 1000)

start_intervals <- matrix(c(start_points - 1, start_points + 1), ncol = 2)

system.time({ roots <- apply(start_intervals, 1, find_the_root, 1e-06)})
```

Part b

```
#setup parallel backend to use many processors
library(parallel)
cl <- makeCluster(cores -1)
clusterExport(cl, "f")
clusterExport(cl, "grad")
system.time(roots_par <- parApply(cl, start_intervals, 1, find_the_root, 1e-06))
stopCluster(cl)
```

As we can see, the results both methods got are almost the same. And in this case, parallel computing is substantially faster than regular ones.