Homework 4

Steven Barnett

10/12/2020

Problem 2: Using the dual nature to our advantage

I'm going to implement the Gradient Descent algorithm in order to estimate parameters in a linear model. For this example, I will be setting the step size, $\alpha = 0.05$ and tolerance $= 0.00001$

```
set.seed(1256)
theta_old <- as.matrix(c(0, 1), nrow = 2)
theta_new <- as.matrix(c(1, 2), nrow = 2)
X <- cbind(1, rep(1:10, 10))
m <- length(X[,1])
h <- X %*% theta_new + rnorm(100, 0, 0.2)
step_size <- 0.0000001
tolerance <- 0.0000000001
count <- 0

while (abs(theta_new[1] - theta_old[1]) > tolerance &
        abs(theta_new[2] - theta_old[2]) > tolerance) {
  theta_old <- theta_new
  theta_new <- theta_new - step_size * (1 / m) * t(X) %*% (X %*% theta_new - h)
  count <- count + 1
}
```

Problem 3

Part A

We are going to use a step size of $1e^{-7}$ and a tolerance of $1e^{-9}$ and try 10000 different combinations of start values for β_0 and β_1. We are going to attempt to use parallel computing for this.

```
gradient_descent <- function(beta, h) {
  theta_new <- beta
  theta_old <- as.matrix(c(beta[1] + 1, beta[2] + 1), nrow = 2)

  X <- cbind(1, rep(1:10, 10))
  m <- length(X[,1])

  step_size <- 0.0000001
  tolerance <- 0.0000000001
  count <- 0

  while (abs(theta_new[1] - theta_old[1]) > tolerance &
          abs(theta_new[2] - theta_old[2]) > tolerance) {
    theta_old <- theta_new
    theta_new <- theta_new - step_size * (1 / m) * t(X) %*% (X %*% theta_new - h)
    count <- count + 1
    if (count > 5000000) {
      break
    }
  }

  return(c("theta" = theta_new, "num_iter" = count))
}

theta <- as.matrix(c(1, 2), nrow = 2)
X <- cbind(1, rep(1:10, 10))
m <- length(X[,1])
h <- X %*% theta + rnorm(100, 0, 0.2)
myobj <- cbind(runif(10000, min = 0, max = 2), runif(10000, min = 1, max = 3))

# cores <- max(1, detectCores() - 1)
# cl <- makeCluster(cores)
# makeForkCluster(cores)
# registerDoParallel(cl)
#
# results <- foreach(index = 1:100, .combine = rbind) %dopar% {
#   gradient_descent(myobj[index,], h)
# }

# stopCluster(cl)

# min(results[,3])
# max(results[,3])
# mean(results[,3])
# sd(results[,3])
```

I attempted to run 10,000 different combinations, but my computer couldn't handle that many in a reasonable time. Instead, I scaled my algorithm down to 100 different combinations. Below are the results:

Minimum number of iterations: 389,454

Maximum number of iterations: 5,000,000

Mean number of iterations: 1,250,595

Standard deviation for number of iterations: 679,738.4

Part B

We could add a stopping rule to the algorithm given that we know the true value. This concept would definitely cut down on execution time as various cycles could be eliminated. However, this algorithm does not do a good job of landing on exact values. We would still need to implement some tolerance between the true value and our stopping criteria. I think this would be similar to our current stopping criteria.

Part C

I think this algorithm is a very clever way to approximate parameters. One application I can think of is to double check one's manual work to estimate parameters. Similar to Monte Carlo simulations, Gradient Descent allows us to harness the power of computing to verify results. In some cases, our manual work cannot estimate parameters as well as using computing power to do so.

Problem 4: Inverting matrices

John Cook does make some good points about the computationally expensive process of inverting large matrices. If I were tasked with computing the given equation, I would take note of a couple things. First, X^tX is an a symmetric and invertible matrix. I would look into doing a kind of factorization or matrix decomposition.

Problem 5: Need for speed challenge

```r
set.seed(12456)

G <- matrix(sample(c(0, 0.5, 1), size = 6400, replace = TRUE), ncol = 10)
R <- cor(G) # R: 10 * 10 correlation matrix of G
C <- kronecker(R, diag(640)) # C is a 16000 * 16000 block diagonal matrix
id <- sample(1:6400, size = 372, replace = FALSE)
q <- sample(c(0, 0.5, 1), size = 6028, replace = TRUE) # vector of length 15068
A <- C[id, -id ] # matrix of dimension 932 * 15068
B <- C[-id, -id] # matrix of dimension 15068 * 15068
p <- runif(372, 0, 1)
r <- runif(6028, 0, 1)
C <- NULL # save some memory space

object.size(A)
```

```
## 17939544 bytes
```

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

```
## 290694488 bytes
```

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

#system.time(y_chol <- p + A %*% chol2inv(chol(B)) %*% (q - r))

#system.time(y_pd <- p + A %*% pd.solve(B) %*% (q - r))

#system.time(chol2inv(chol(B)))
```

Part A

In order to run this on my machine, I reduced the dimensions of A and B. A is now a 372x6028 matrix and B is a 6028x6028 matrix. Given those modifications, A is about 17.9 MB. B is about 290 MB. Without any optimization tricks, it takes around two minutes to calculate y.

Part B

The two steps that are taking the longest are 1) taking the inverse of B and 2) multiplying A and B^{-1} together.

If A and B were sparse matrices, then we could do some optimization both in storage and multiplication of the matrices.

Part C

I used a couple packages to speed up the calculation of y. First, I used the function chol2inv. I then used the mnormt package and the pd.solve function within that package. Neither of these two modifications sped up the calculation as I was hoping. In fact, they almost seemed to calculate y slower.

My next steps would be to explore some other linear algebra libraries as well as rely on C++ to speed up the code.

Problem 3

Part A + B

```
get_proportion_of_success <- function(results) {
  ## Successes represented by a 1. Failures represented by a 0.
  ## Summing the vector will return the number of successes
  return(sum(results) / length(results))
}

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

Part C

The proportion of success is the same for all columns. This is due to the fact that or matrix was created with one call of rbinom() repeated 10 times.

```
prop_success_by_row <- apply(P4b_data, 1, get_proportion_of_success)
prop_success_by_column <- apply(P4b_data, 2, get_proportion_of_success)
prop_success_by_row
```

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

```
prop_success_by_column
```

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

```
get_prob_vector <- function(p) {
  prob_vector <- rbinom(10, 1, p)
}

desired_probabilities <- (31:40) / 100
probabilities_matrix <- sapply(desired_probabilities, get_prob_vector)

prop_success_by_row <- apply(probabilities_matrix, 1, get_proportion_of_success)
prop_success_by_column <- apply(probabilities_matrix, 2,
                                get_proportion_of_success)
prop_success_by_row
```

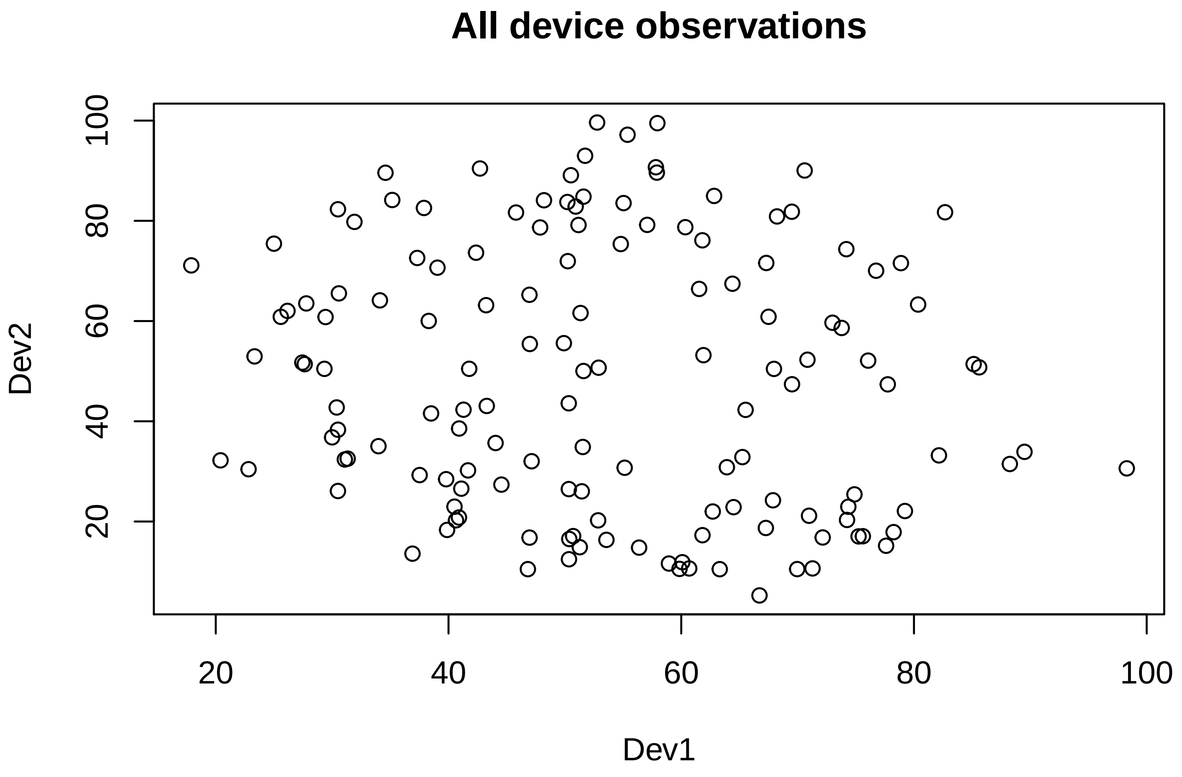
```
##  [1] 0.7 0.3 0.5 0.5 0.3 0.1 0.8 0.4 0.1 0.2
```

```
prop_success_by_column
```

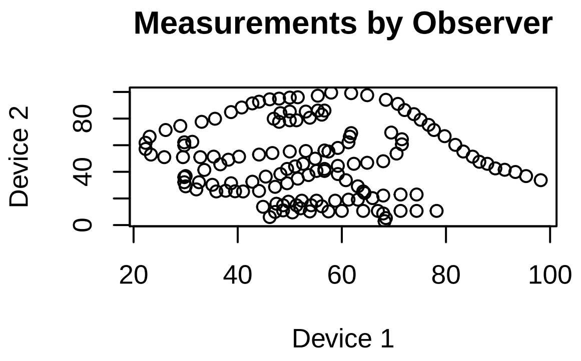
```
##  [1] 0.2 0.3 0.4 0.3 0.4 0.6 0.3 0.3 0.5 0.6
```

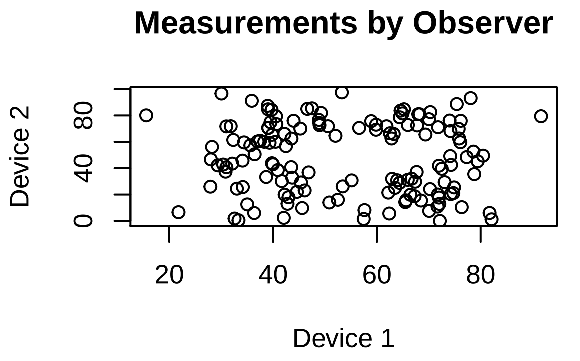
Problem 4

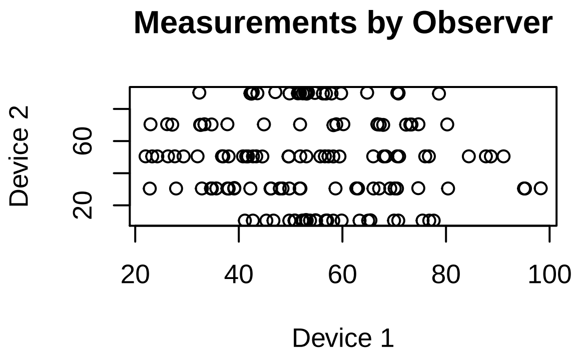
```
my_data <- readRDS("./dwnldd_data/HW3_data.rds")
colnames(my_data) <- c("Observer", "x", "y")
observers <- unique(my_data$Observer)

create_scatter_plot <- function(values = my_data, index ,title, xlabel, ylabel) {
  plot_values <- subset(values, Observer == index)
  plot(x = dplyr::pull(plot_values, x), y = dplyr::pull(plot_values, y),
       main = title, xlab = xlabel, ylab = ylabel)
}

create_scatter_plot(values = my_data, index = observers,
                    title = "All device observations", xlabel = "Dev1",
                    ylabel = "Dev2")
```

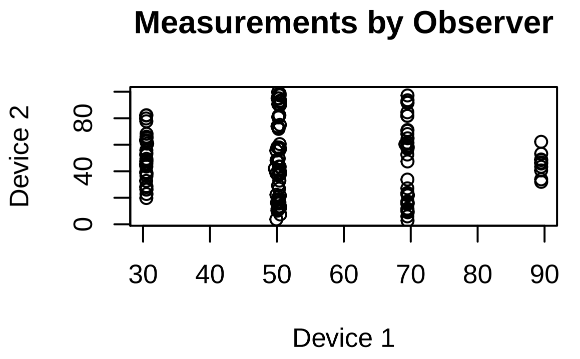


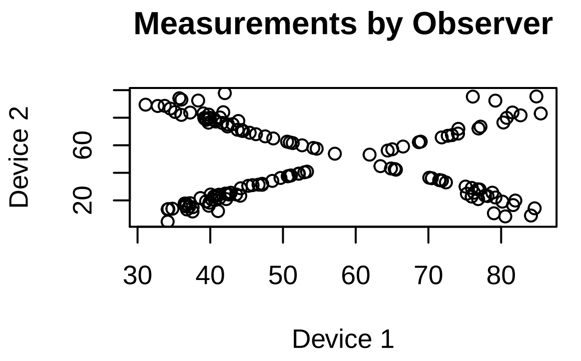
```
par(mfrow = c(2, 2))
results <- sapply(observers, create_scatter_plot, values = my_data,
       title = "Measurements by Observer", xlabel = "Device 1",
       ylabel = "Device 2")
```

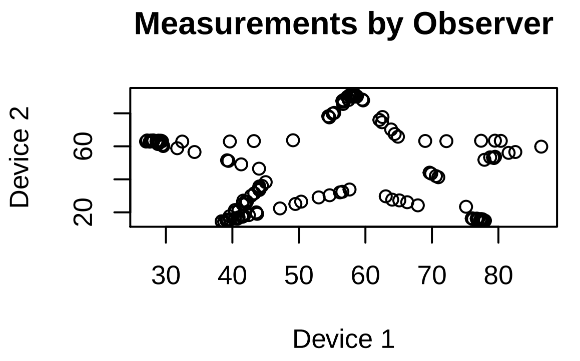

Measurements by Observer
Device 2
0
40
80
Device 1
20
40
60
80
100

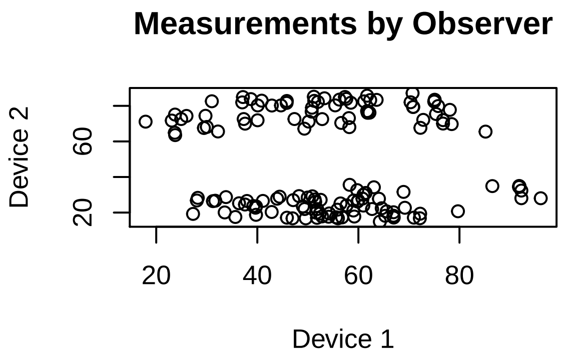

Measurements by Observer
Device 2
0
40
80
Device 1
20
40
60
80

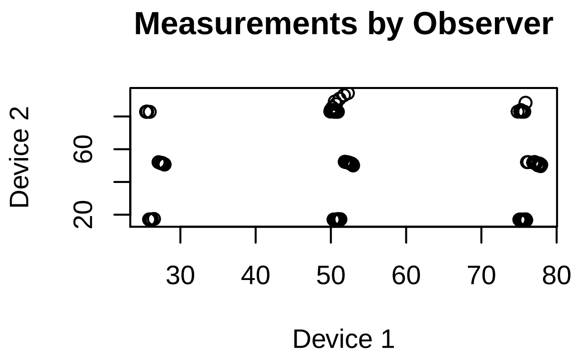

Measurements by Observer
Device 2
20
60
Device 1
20
40
60
80
100

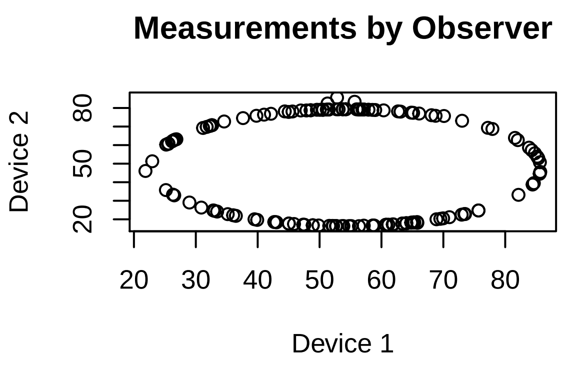

Measurements by Observer
Device 2
0
40
80
Device 1
30
40
50
60
70
80
90

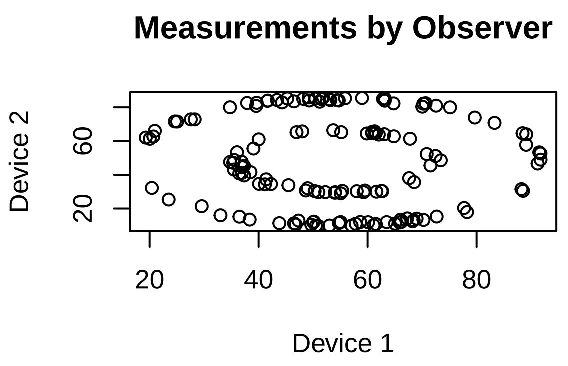

Measurements by Observer
Device 2
20
60
Device 1
30
40
50
60
70
80

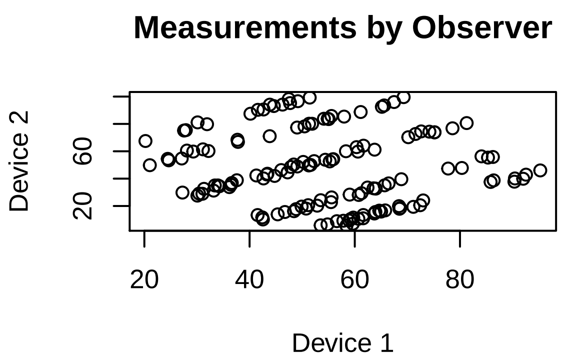

Measurements by Observer
Device 2
20
60
Device 1
30
40
50
60
70
80

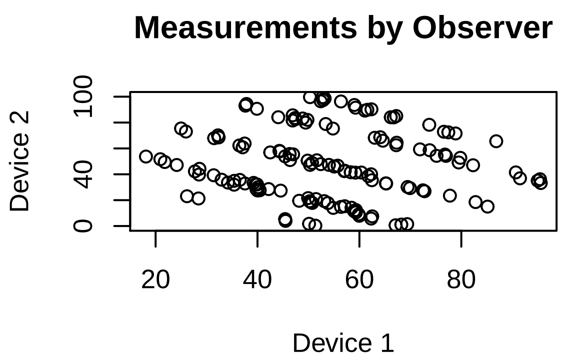

Measurements by Observer
Device 2
20
60
Device 1
20
40
60
80

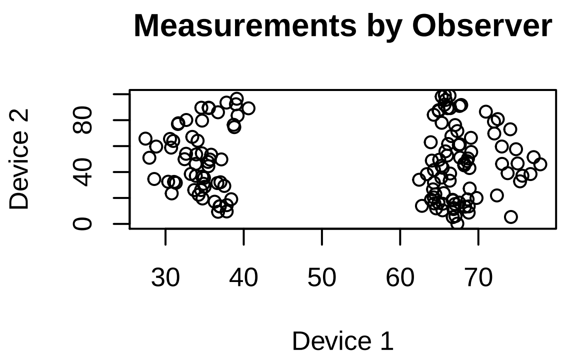

Measurements by Observer
Device 2
20
60
Device 1
30
40
50
60
70
80











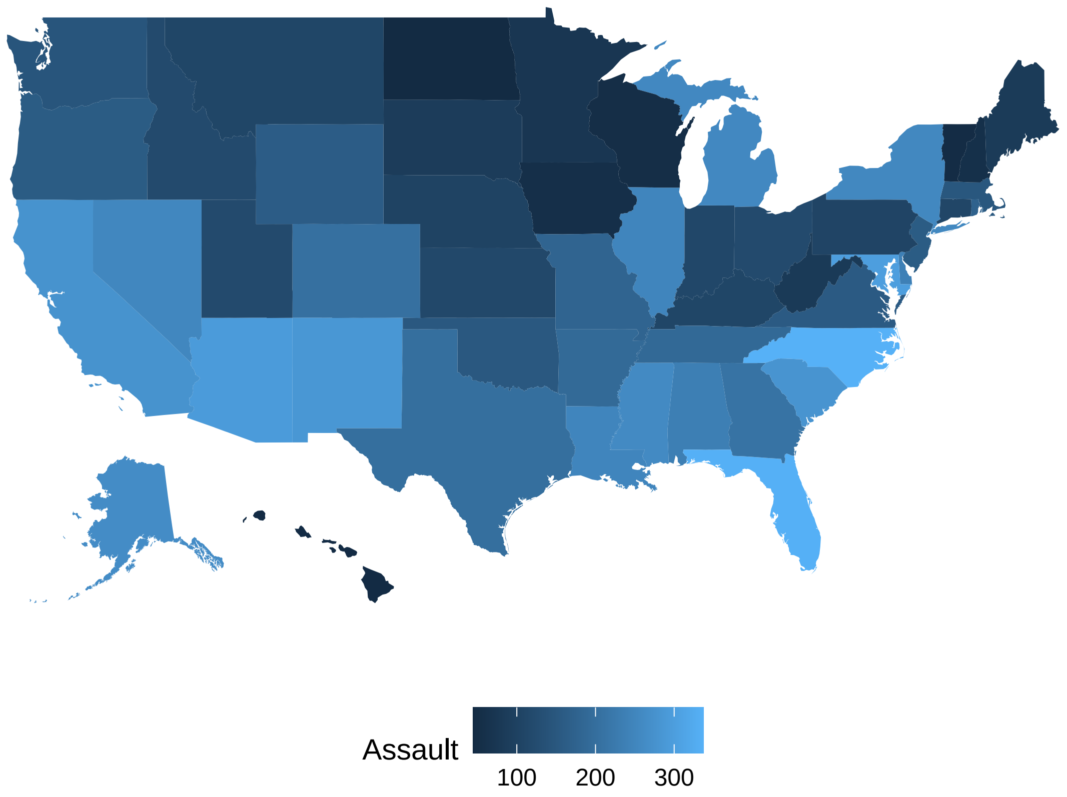
Problem 5

```
download("http://www.farinspace.com/wp-content/uploads/us_cities_and_states.zip",
         dest = "./dwnldd_data/us_cities_states.zip")
unzip("./dwnldd_data/us_cities_states.zip", exdir = "./dwnldd_data")


states <- fread(input = "./dwnldd_data/us_cities_and_states/states.sql",
                skip = 23, sep = "'", sep2 = ",", header = FALSE,
                select = c(2, 4))

cities <- fread(input = "./dwnldd_data/us_cities_and_states/cities_extended.sql",
                skip = 23, sep = "'", sep2 = ",", header = FALSE,
                select = c(2, 4))

cities <- filter(cities, V4 != "PR")
```

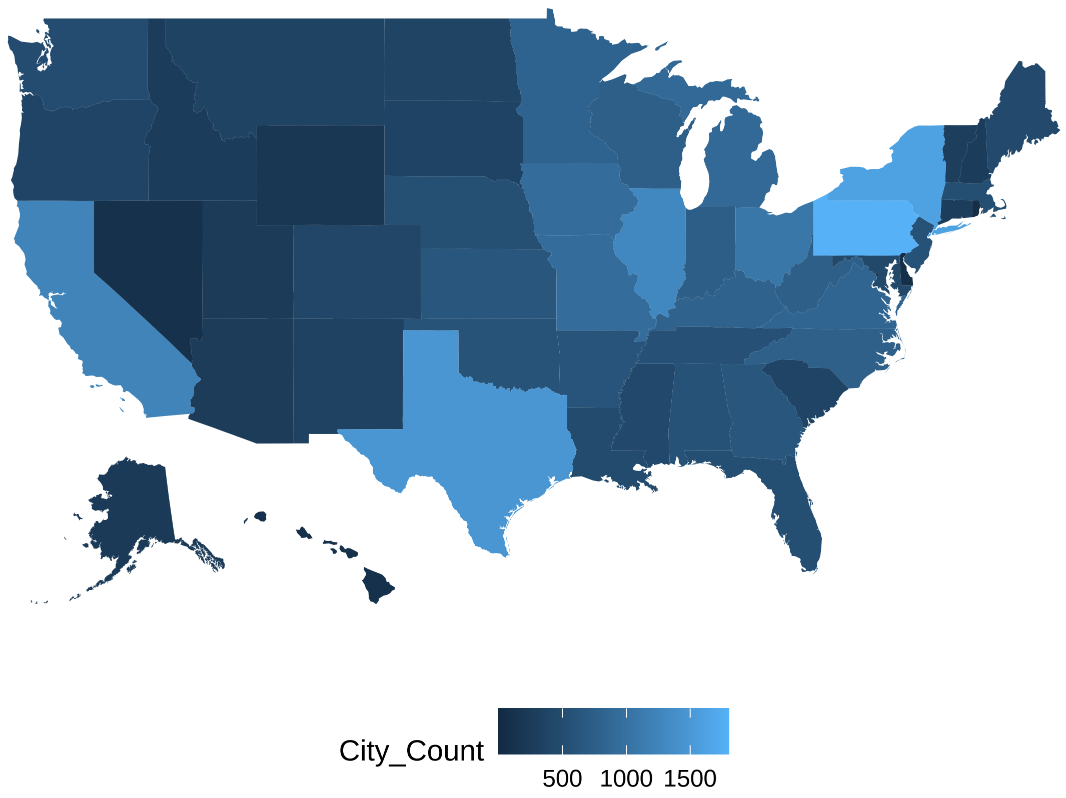
```r
city_count_by_state <- summarise(group_by(cities, V4), count = n_distinct(V2))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

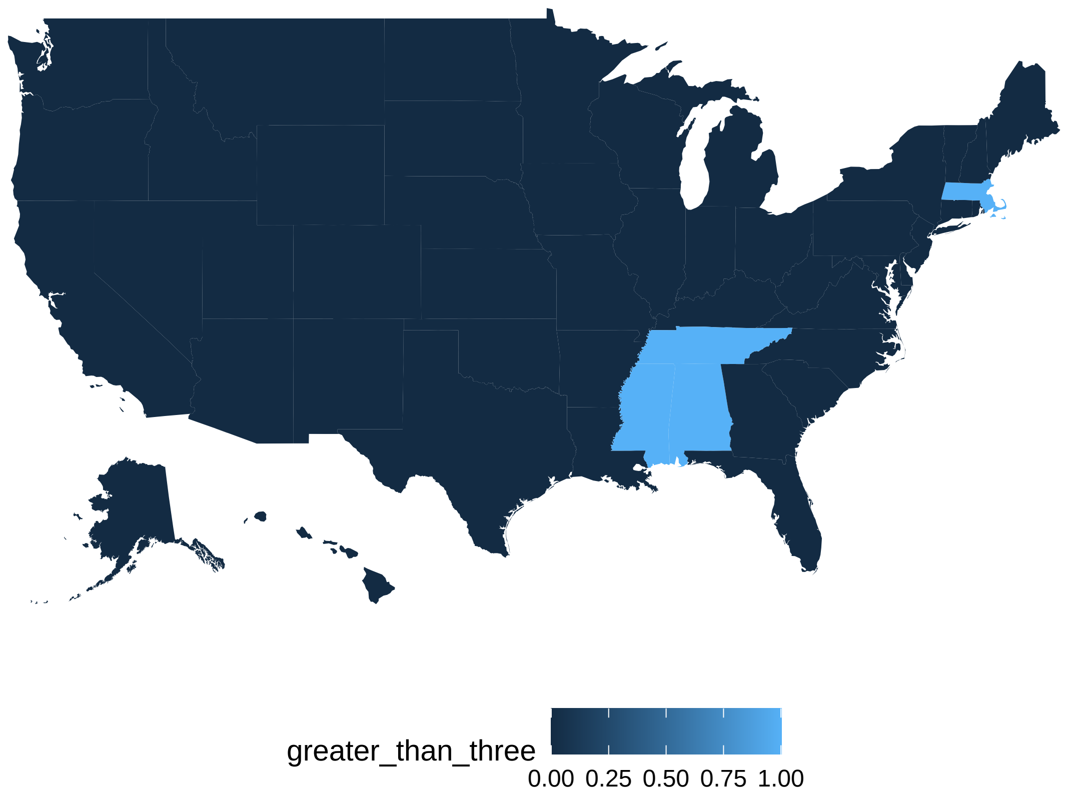
```r
city_count_by_state <- cbind(city_count_by_state, tolower(states$V2))
colnames(city_count_by_state) <- c("State", "City_Count", "State_Long")

get_letter_count <- function(letter, state_name) {
  letter_count <- str_count(state_name, letter)
  return(letter_count)
}
letter_count <- data.frame(matrix(NA, nrow = 51, ncol = 26))

for (i in 1:51) {
  current_state <- toupper(states$V2[i])
  letter_count[i,] <- lapply(toupper(letters), get_letter_count, state_name = current_state)
}

greater_than_three <- apply(letter_count, 1, function(x) if(any(x > 3)) { return(1)} else {return(0)})
city_count_by_state <- cbind(city_count_by_state, greater_than_three)

data("fifty_states")
crimes <- data.frame(state = tolower(rownames(USArrests)), USArrests)

ggplot(crimes, aes(map_id = state)) +
  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())
```



```
ggplot(city_count_by_state, aes(map_id = State_Long)) +
  geom_map(aes(fill = City_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())
```



```
ggplot(city_count_by_state, aes(map_id = State_Long)) +
  geom_map(aes(fill = greater_than_three), 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())
```



Problem 2

Part A

First, the question asked in the StackExchange was why is the supplied code not working. This question was actually never answered. What is the problem with the code?

The StackExchange user is getting the same results because he is passing in static values to the lm() function and not column names.

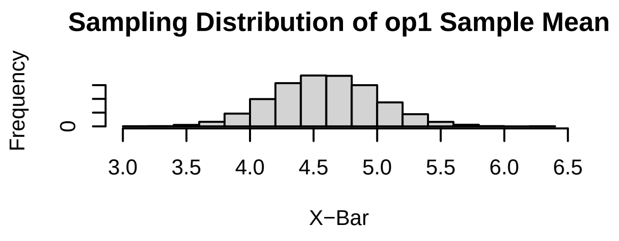
Part B

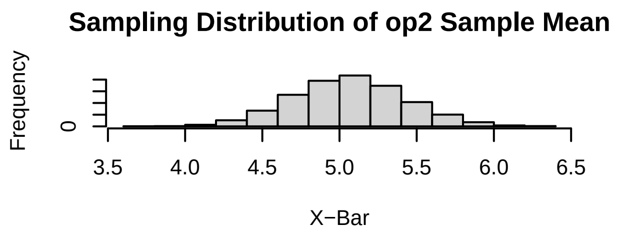
Bootstrap the analysis to get the parameter estimates using 100 bootstrapped samples. Make sure to use system.time to get total time for the analysis. You should probably make sure the samples are balanced across the operators, i.e. each sample draws for each operator.

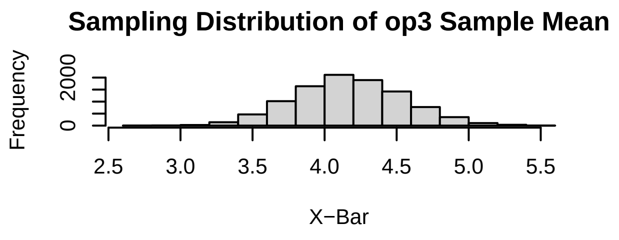
I will be using bootstrapping to estimate the mean of each operator. I will do this by creating 100 bootstrapped samples for each operator and taking the sample mean. This can also work in tandem with the Central Limit Theorem and allow us to make inferences using an approximated normal distribution.

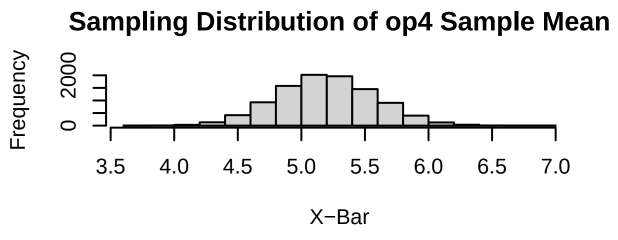
```
sensory_data <- readRDS("./dwnldd_data/sensory_data_raw.RDS")
jagged_data <- select(filter(sensory_data, !is.na(`5`)), !Item)
unjagged_data <- select(filter(sensory_data, is.na(`5`)), !`5`)
colnames(unjagged_data) <- colnames(sensory_data)[2:6]
sensory_data <- union(jagged_data, unjagged_data)
colnames(sensory_data) <- c("op1", "op2", "op3", "op4", "op5")

operator_means_orig <- apply(sensory_data, 2, mean)
bootstrap_means_non_par <- data.frame()

start_time <- proc.time()
```

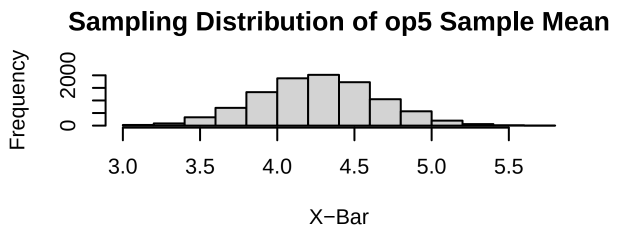
```r
## Bootstrap
for (i in 1:10000) {
  bootstrapped_data <-apply(sensory_data, 2, sample, size = 30, replace = TRUE)
  means <- apply(bootstrapped_data, 2, mean)
  bootstrap_means_non_par <- rbind(bootstrap_means_non_par, means)
}
non_parallel_time <- proc.time() - start_time
colnames(bootstrap_means_non_par) <- colnames(sensory_data)
sum_means_non_par <- apply(bootstrap_means_non_par, 2, mean)

par(mfrow = c(3,2))
hist(bootstrap_means_non_par[,1], main = "Sampling Distribution of op1 Sample Mean",
     xlab = "X-Bar")
hist(bootstrap_means_non_par[,2], main = "Sampling Distribution of op2 Sample Mean",
     xlab = "X-Bar")
hist(bootstrap_means_non_par[,3], main = "Sampling Distribution of op3 Sample Mean",
     xlab = "X-Bar")
hist(bootstrap_means_non_par[,4], main = "Sampling Distribution of op4 Sample Mean",
     xlab = "X-Bar")
hist(bootstrap_means_non_par[,5], main = "Sampling Distribution of op5 Sample Mean",
     xlab = "X-Bar")
```











Part C

Redo the last problem but run the bootstraps in parallel (cl <- makeCluster(8)), don't forget to stopCluster(cl)). Why can you do this? Make sure to use system.time to get total time for the analysis.

```r
## Bootstrap
cores <- max(1, detectCores() - 1)
```

```r
cl <- makeCluster(cores)
makeForkCluster(cores)
```

```
## socket cluster with 3 nodes on host 'localhost'
```

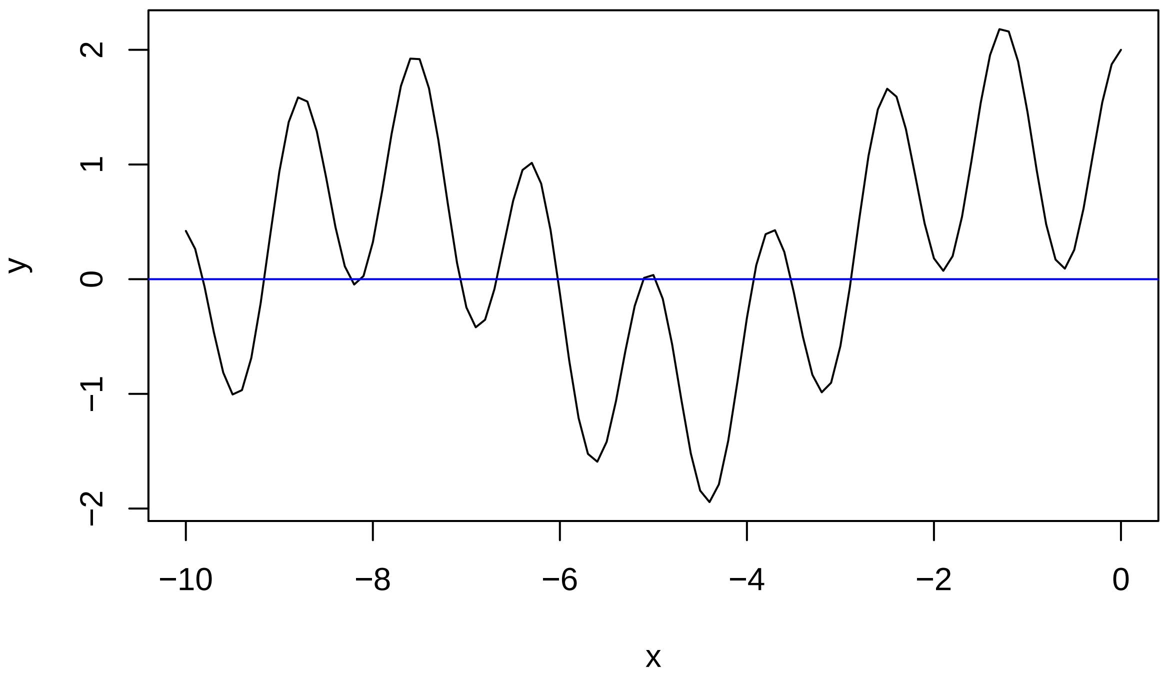
```r
registerDoParallel(cl)

start_time <- proc.time()
bootstrap_means_par <- foreach(index = 1:1000, .combine = rbind) %dopar% {
  apply(apply(sensory_data, 2, sample, size = 30, replace = TRUE), 2, mean)
}
parallel_time <- proc.time() - start_time
stopCluster(cl)
colnames(bootstrap_means_par) <- colnames(sensory_data)
sum_means_par <- apply(bootstrap_means_par, 2, mean)


sensory_df <- data.frame(non_parallel_time["elapsed"])
sensory_df <- rbind(sensory_df, parallel_time["elapsed"])

means_matrix <- matrix(c(sum_means_non_par, sum_means_par), ncol = 5, byrow = TRUE)
sensory_df <- cbind(sensory_df, means_matrix)
colnames(sensory_df) <- c("Elapsed Time", colnames(sensory_data))
row.names(sensory_df) <- c("Non-Parallel", "Parallel")

knitr::kable(sensory_df)
```

	Elapsed Time	op1	op2	op3	op4	op5
Non-Parallel	16.907	4.592452	5.065167	4.169800	5.195335	4.264641
Parallel	1.721	4.599197	5.068240	4.178983	5.200127	4.276100

As expected, the parallel computation completes the bootstrapping much quicker than the non-parallel computation. I expect as the number of bootstraps I run, the wider the gap between parallel and non-parallel will be.

Problem 3

```r
x_vals <- seq(-10, 0, by = 0.1)
y_vals <- 3^x_vals - sin(x_vals) + cos(5 * x_vals)
plot(x = x_vals, y = y_vals, type = "l", xlab = "x", ylab = "y")
abline(h = 0, col = "blue")
```



The equation $y = 3^x - sin(x) + cos(5x)$ contains an infinite number of roots as $x \to \infty$. For this problem, I will focus on the range $x \in [-10, 0]$. From the plot above, this equation appears to have 12 roots for $x \in [-10, 0]$.

Part A.

Using one of the apply functions, find the roots noting the time it takes to run the apply function.

```
get_newton_method <- function(x_0, tolerance) {
  attempt <- 0
  new_estimate <- x_0
  should_continue <- TRUE
  while (should_continue) {
    prev_estimate <- new_estimate
    func_result <- 3^prev_estimate - sin(prev_estimate) + cos(5 * prev_estimate)
    derivative_result <- 3^prev_estimate * log(3) - cos(prev_estimate) -
                          5 * sin(5 * prev_estimate)
    new_estimate <- prev_estimate - (func_result / derivative_result)
    attempt <- attempt + 1
    ## Evaluating if we have reached the tolerance level to be satisfied with
    ## our result
    should_continue <- abs(prev_estimate - new_estimate) > tolerance
  }
  return(new_estimate)
}

tol <- 0.00001
estimate_vector <- seq(-20, 0, by = 0.1)
non_par_newton_time <- system.time(roots <- sapply(estimate_vector,
                                                   get_newton_method,
                                                   tolerance = tol))
non_par_newton_time
```

```
##    user  system elapsed
##   0.026   0.000   0.029
```

```r
non_par_unique_roots <- unique(round(roots[roots > -10 & roots < 0], 4))
```

Part B

Repeat the apply command using the equivalent parApply command. Use 8 workers.

My machine only has 5 cores, so I will be using 4 workers.

```r
cores <- max(1, detectCores() - 1)
cl <- makeCluster(cores)
registerDoParallel(cl)
par_newton_time <- system.time(roots <- parSapply(cl, estimate_vector,
                                                  get_newton_method,
                                                  tolerance = tol))
par_unique_roots <- unique(round(roots[roots > -10 & roots < 0], 4))
stopCluster(cl)

newton_df <- data.frame(paste(par_unique_roots, collapse = ","),
                        round(par_newton_time["elapsed"], 10))
newton_df <- rbind(newton_df, c(paste(non_par_unique_roots, collapse = ","),
                                round(non_par_newton_time["elapsed"], 10)))

row.names(newton_df) <- c("Parallel", "Non-Parallel")
colnames(newton_df) <- c("Unique Roots", "Time Elapsed")

knitr::kable(newton_df)
```

	Unique Roots	Time Elapsed
Parallel	-3.5287,-9.8175,-9.163,-8.1159,-8.2466,-6.0212,-7.0685,-6.6761,-5.1074,-2.8871,-4.9715,-3.9301	0.008
Non-Parallel	-3.5287,-9.8175,-9.163,-8.1159,-8.2466,-6.0212,-7.0685,-6.6761,-5.1074,-2.8871,-4.9715,-3.9301	0.029

This table shows results that one would not expect. Parallel computing should be faster than non-parallel computing. However, because the computation time is so low, I believe the start up cost for parallel computing factored into the length of time, making it take longer than non-parallel.

For my grid, I set the values to be spaced apart by 0.1. Once I decreased that value, both non-parallel and parallel computing could not complete it within a reasonable time. I am assuming this is due to the low computing power of my machine.