Homework 3

Youhui Ye

9/13/2020

Problem 3

I learned at least 3 things from the R programming style. First, we should write necessary comments to help others follow our codes. Second, we should use indent to indicate specific code parts. Third, it is important to name variables and functions decently. Specifically, I used to name variables in "camelCase" style. And I decided to follow Hadley Wickham's instruction, who recommended to use underscore between words.

Problem 5

```
summary_dat <- function(dat) {
  ## Calculate means
  mu1 <- mean(dat[,1])
  mu2 <- mean(dat[,2])
  ## Calculate standard deviations
  sd1 <- sd(dat[,1])
  sd2 <- sd(dat[,2])
  ## Calculate the correlation
  correlation <- cor(dat[,1], dat[,2])
  result <- c(mu1, mu2, sd1, sd2, correlation)
  return(result)
}

url <- "https://raw.githubusercontent.com/rsettlage/STAT_5014_Fall_2020/master/homework/HW3_data.rds"
download.file(url,"HW3_data.rds", method="curl")
dat5 <-readRDS("HW3_data.rds")

## Calculate the number of obersers
num_observer <- length(unique(dat5$Observer))
summary_observers <- data.frame(matrix(NA, nrow = num_observer, ncol = 5))
colnames(summary_observers) <- c("mean_dev1", "mean_dev2", "sd_dev1", "sd_dev2", "correlation")
for (i in 1:13) {
  summary_observers[i, ] <- summary_dat(dat5[dat5$Observer == i, 2:3])
}
```

a.

```
knitr::kable(summary_observers, digits = 3)
```

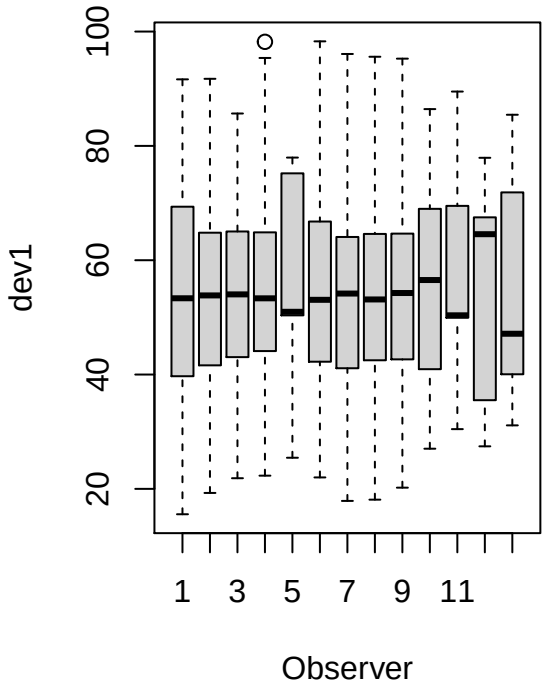
mean_dev1	mean_dev2	sd_dev1	sd_dev2	correlation
54.266	47.835	16.770	26.940	-0.064
54.269	47.831	16.769	26.936	-0.069
54.267	47.838	16.760	26.930	-0.068
54.263	47.832	16.765	26.935	-0.064
54.260	47.840	16.768	26.930	-0.060
54.261	47.830	16.766	26.940	-0.062
54.269	47.835	16.767	26.940	-0.069
54.268	47.836	16.767	26.936	-0.069
54.266	47.831	16.769	26.939	-0.069
54.267	47.840	16.769	26.930	-0.063
54.270	47.837	16.770	26.938	-0.069
54.267	47.832	16.770	26.938	-0.067
54.260	47.840	16.770	26.930	-0.066

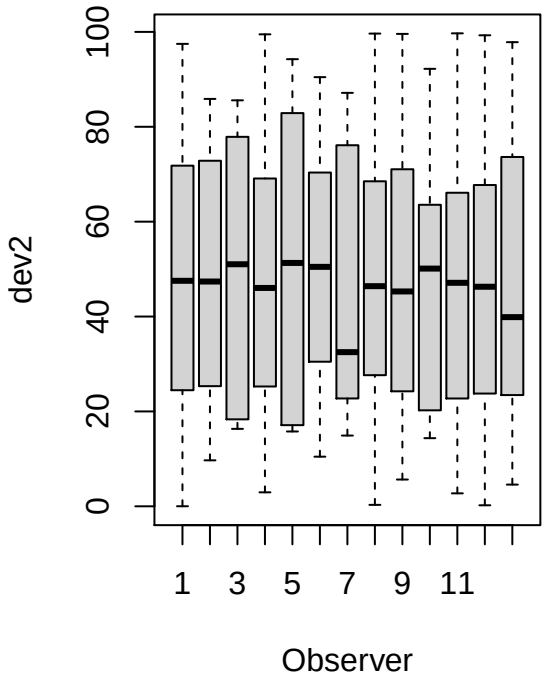
```
## Check the results using group_by and summarise
# dat5 %>% group_by(Observer) %>%
# summarise(mu1 = mean(dev1), mu2 = mean(dev2), sd1 = sd(dev1), sd2 = sd(dev2), correlation = cor(dev1,
```

From the summary table, we can say means, standard deviations and correlations are almost the same across all observers.

b.

```
par(mfrow = c(1,2))
boxplot(dev1 ~ Observer, data = dat5)
boxplot(dev2 ~ Observer, data = dat5)
```

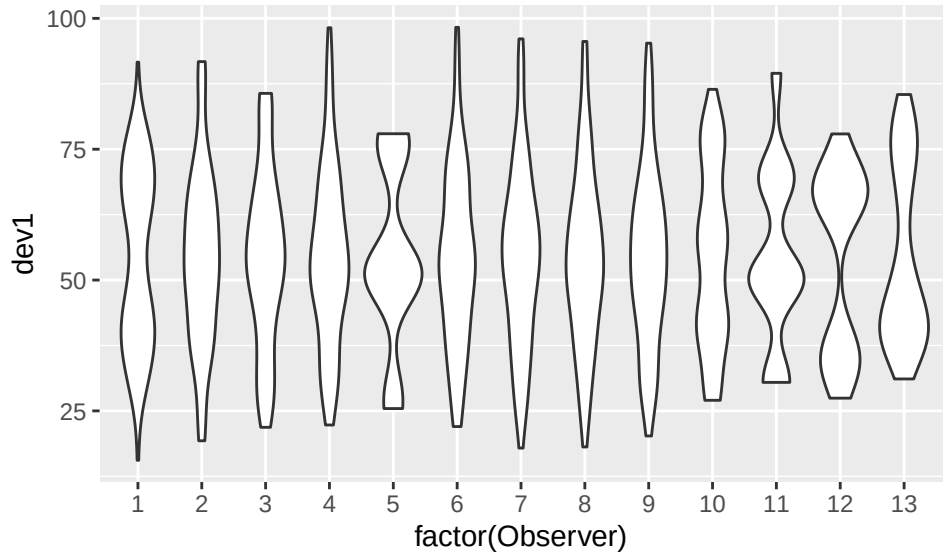




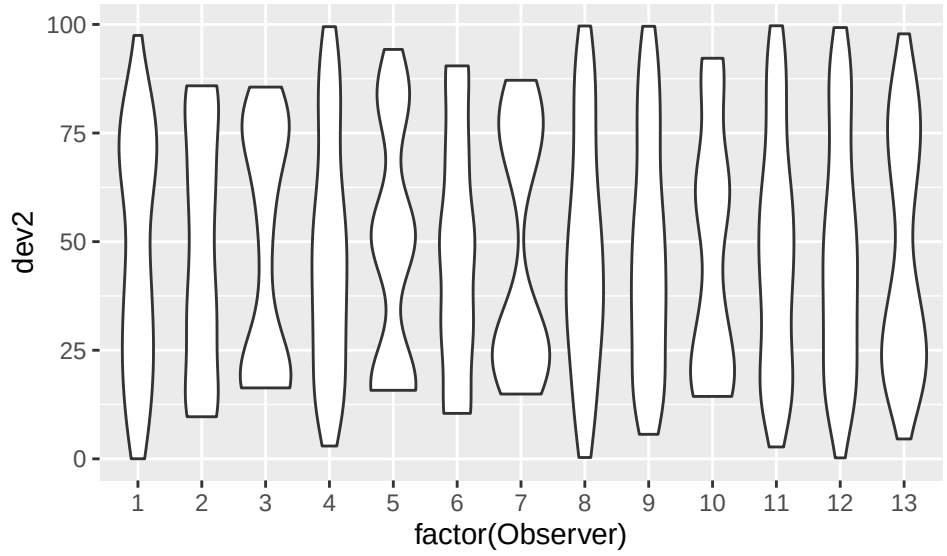
However, boxplots reveal more information about data distribution, from which we know observers recorded various observations.

c.

```
ggplot(data = dat5) + geom_violin(mapping = aes(factor(Observer), dev1))
```



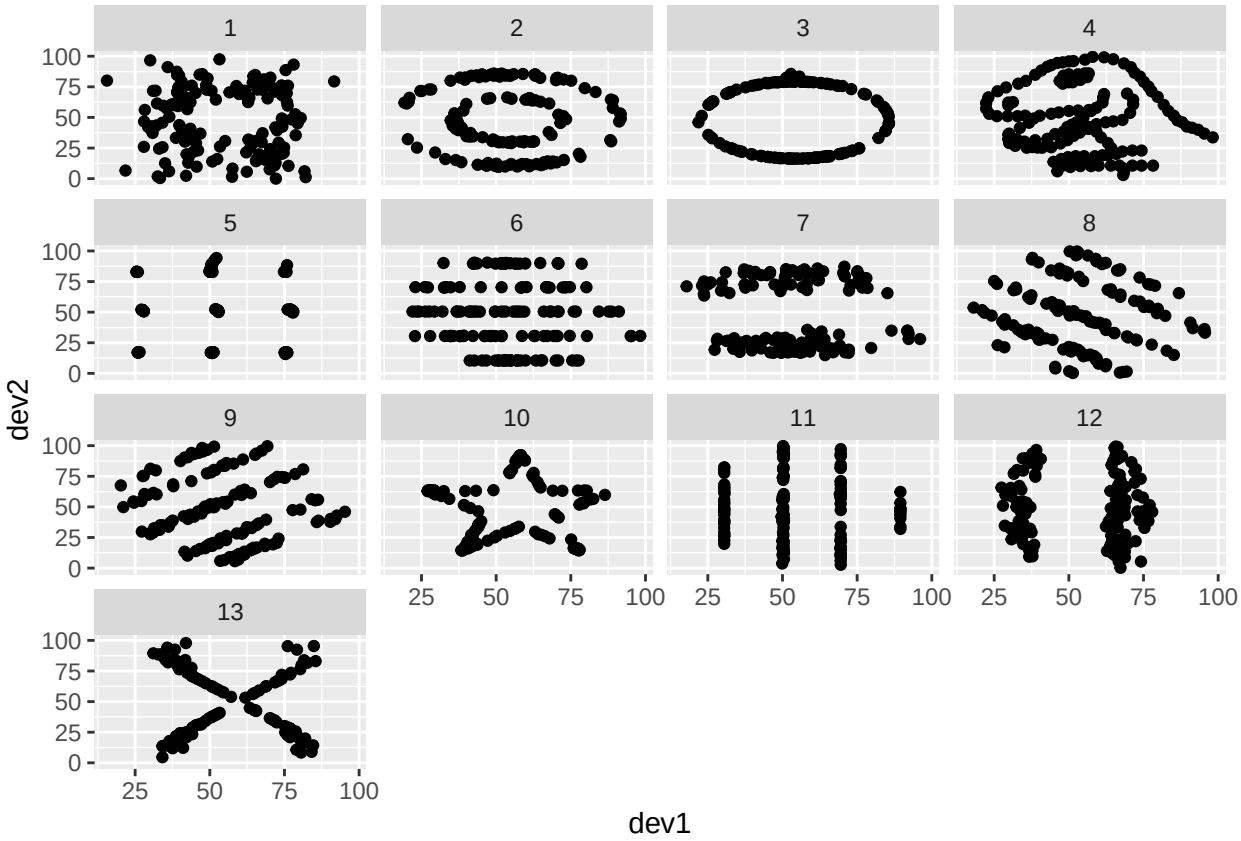
```
ggplot(data = dat5) + geom_violin(mapping = aes(factor(Observer), dev2))
```



Distributions of "dev" variables vary with each observer. As you can see, some observers' data are nearly normal while others are bimodal. Compared to boxplots and summary statistics, violin plots does give us a better portrait of data distribution.

d.

```
ggplot(dat5, aes(x=dev1,y=dev2)) + geom_point() + facet_wrap(Observer~.)
```



I saw 10 patterns (chaos, concentric circles, star, dinosaur, etc.). What I learned is never draw a conclusion before you have a full understanding of the data you are analyzing. Means and standard deviations are just brief summary of our data and they have abandoned most of information. Therefore, they are not trustable under this kind of scenarios.

Problem 6

```
calculate_integral <- function(width) {
  x <- seq(0, 1, width)
  ## discard the first one
  right_sum <- width * sum(exp(-x[-1]^2/2))

  ## discard the last one
  #left_sum <- width * sum(exp(-x[-length(x)]^2/2))
  return(right_sum)
}

## Initiate value of slice width
width <- 0.2

## left_sum - right_sum = width * (largest_function_value - smallest_function_value)
while (width * (exp(0) - exp(-1/2)) >= 1e-6) {
  width <- width / 2
  result <- calculate_integral(width)
  cat("Integral: ", result, "Slice Width Used: ", width, "\n")
```

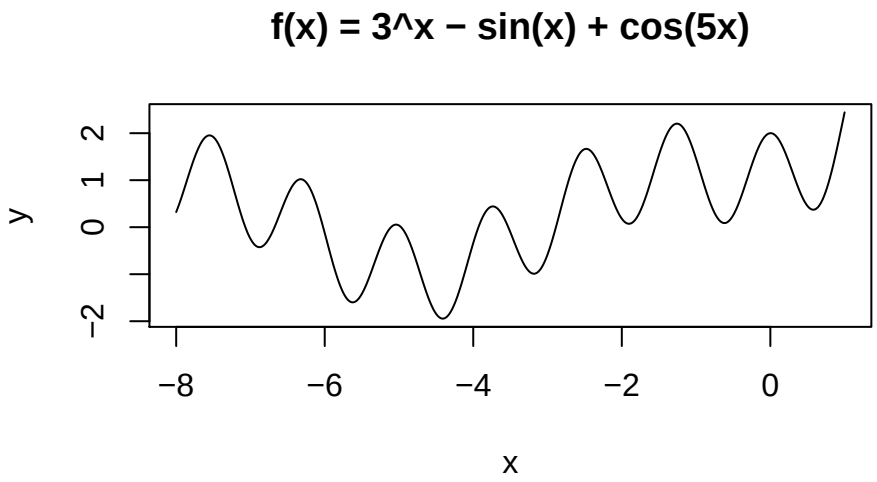
```
}
```

```
## Integral:  0.8354453 Slice Width Used:  0.1
## Integral:  0.8456613 Slice Width Used:  0.05
## Integral:  0.8506744 Slice Width Used:  0.025
## Integral:  0.8531573 Slice Width Used:  0.0125
## Integral:  0.8543928 Slice Width Used:  0.00625
## Integral:  0.8550091 Slice Width Used:  0.003125
## Integral:  0.8553169 Slice Width Used:  0.0015625
## Integral:  0.8554707 Slice Width Used:  0.00078125
## Integral:  0.8555475 Slice Width Used:  0.000390625
## Integral:  0.855586 Slice Width Used:  0.0001953125
## Integral:  0.8556052 Slice Width Used:  9.765625e-05
## Integral:  0.8556148 Slice Width Used:  4.882813e-05
## Integral:  0.8556196 Slice Width Used:  2.441406e-05
## Integral:  0.855622 Slice Width Used:  1.220703e-05
## Integral:  0.8556232 Slice Width Used:  6.103516e-06
## Integral:  0.8556238 Slice Width Used:  3.051758e-06
## Integral:  0.8556241 Slice Width Used:  1.525879e-06
```

Slice width 1.525879e-06 is necessary to get desired answer.

Problem 7

```
x <- seq(-8, 1, 0.01)
y <- 3^x - sin(x) + cos(5*x)
plot(x, y, type = 'l', main = "f(x) = 3^x - sin(x) + cos(5x)")
```



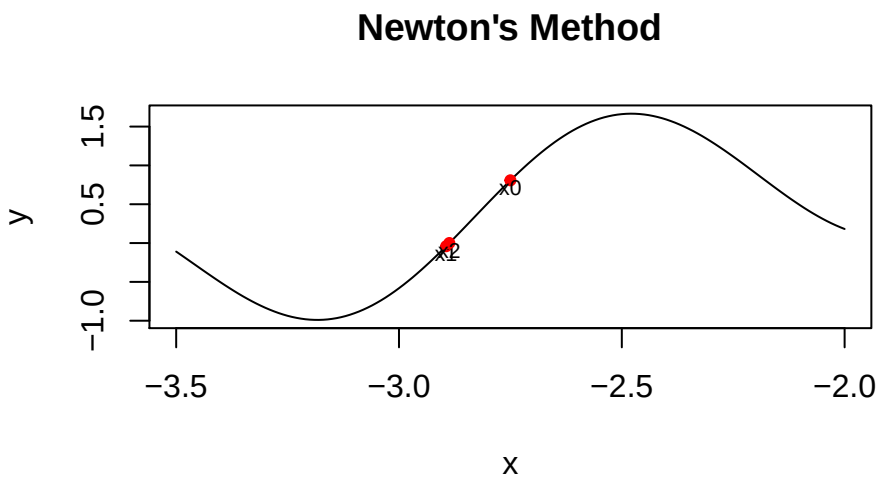
Apparently, this function has no more than one root. The choice of start point and interval would have big influence on the result, so the function would take interval as one parameter and start the loop at mid of the interval. Also, tolerance should be treated as an input.

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

```r
  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(tol, interval) {
  ## tolerance is a single value while interval is a vector of dimension 2
  left <- interval[1]
  right <- interval[2]
  initial_point <- (left + right) / 2
  x <- seq(left, right, 0.01)
  y <- 3^x - sin(x) + cos(5*x)
  plot(x, y, type = 'l', main = "Newton's Method")
  points(initial_point, f(initial_point), pch = 19, col = 'red', cex = 0.7)
  text(initial_point, f(initial_point) - 0.1, labels="x0", cex = 0.7)
  ## use while loop to solve the problem
  prev <- initial_point
  i <- 1
  while (abs(f(prev)) > tol ) {
    ## calculate a new point
    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
    points(curr, f(curr), pch = 19, col = 'red', cex = 0.7)
    label <- paste0("x",i)
    text(curr, f(curr)-0.1, labels=label, cex = 0.7)
    i <- i + 1
  }
  cat("Tolerance Used:", tol, "\n")
  cat("Interval Used:", "(", interval[1], ",", interval[2], ")", "\n")
  cat("Final Answer:", curr, "  Function value:", f(curr))
}
find_the_root(0.001, c(-3.5, -2))
```




```
## Tolerance Used: 0.001
## Interval Used: ( -3.5 , -2 )
## Final Answer: -2.887025   Function value: 0.0001917383
```

Problem 8

```
set.seed(12345)
X <- cbind(rep(1,100),rep.int(1:10,time=10))
beta <- c(4,5)
y <- X%*%beta + rnorm(100)

mu <- mean(y)
ones <- rep(1, 100)
loop_sum <- function() {
  summation <- 0
  for (i in 1:length(y)) {
    summation <- summation + (y[i] - mu)^2
  }
  return(summation)
}

matrix_sum <- function() {
  summation <- t((y - mu) * (y - mu)) %*% ones
  return(summation)
}
loop_sum()
```

```
## [1] 20467.85
```

```
matrix_sum()
```

```
##          [,1]
## [1,] 20467.85
```

```
set.seed(123456)
microbenchmark(loop_sum,matrix_sum,  times = 100)
```

```
## Unit: nanoseconds
##        expr min lq  mean median uq  max neval
##    loop_sum  31 32 45.16     33 33 1284   100
##  matrix_sum  34 35 66.08     36 37 3021   100
```