Homework 3

Yixin Chen

2020-09-29

Problem 3

In the lecture, there were two links to programming style guides. What is your takeaway from this and what specifically are *you* going to do to improve your coding style?

Takeaway: We need to make our code readable and well documented in order to enable Reproducible Research.

I will start commenting more while I'm coding because I didn't comment at all before. Sometimes I have to revisit my code after a while and I still need to spend a lot of time to understand my own code.

Problem 5

```
ComputeSummary <- function(x){
  # A function to compute the geometric means, standard deviations and
  # correlation of the columns of a 2-column dataframe.
  if (ncol(x) != 2)  stop("The dataframe must contain only 2 columns")
  if (!is.numeric(x[,1]) || !is.numeric(x[,2]))  stop("The dataframe must be numeric")
  #Compute column means
  means <- colMeans(x)
  # Compute column standard deviations
  std<-numeric(2)
  for (i in 1: 2) {
    std[i] <- sd(x[,i])
  }
  # Compute correlation between 2 columns
  corr <- cor(x[,1],x[,2])
  SumStats <- c(means, std, corr)
  return(SumStats)
}
```

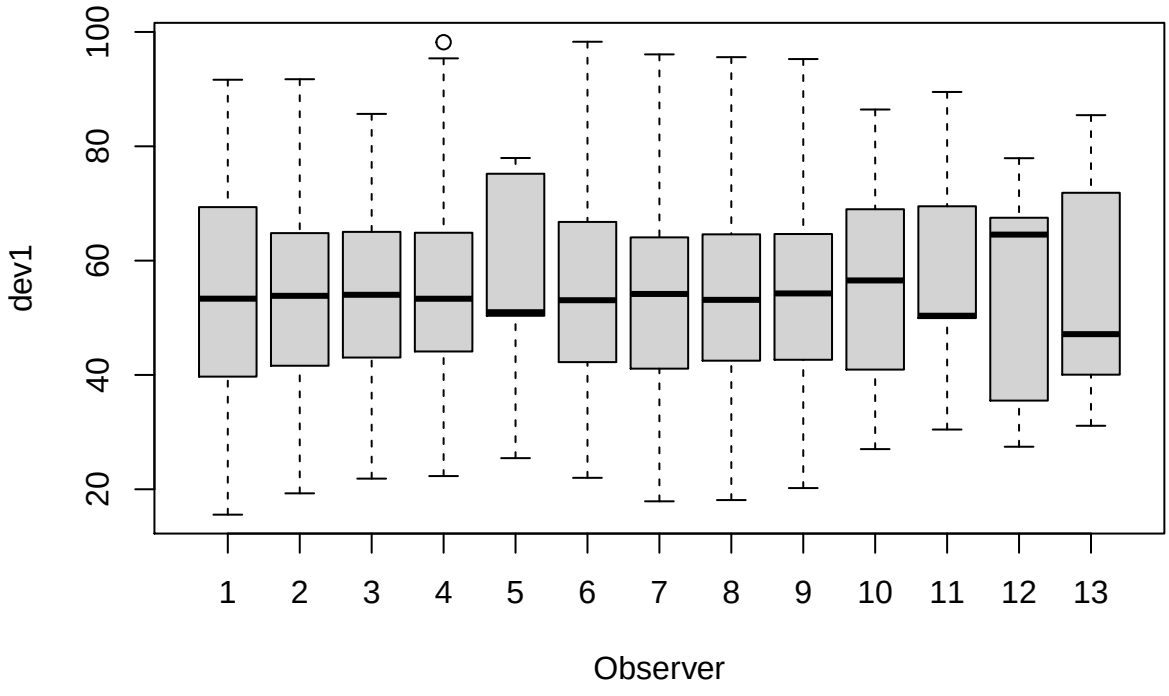
```
url <- "https://github.com/rsettlage/STAT_5014_Fall_2020/blob/master/homework/HW3_data.rds"
download.file(url,"HW3_data.rds")
#for some reason, I can't read the downloaded data, so I manually downloded the
#dataset from canvas announcement and loaded it.
HW3data <- readRDS("HW3_data_canvas.rds")
SumObs <- data.frame(matrix(NA,13,5))
colnames(SumObs) <- c("dev1_mean","dev2_mean","dev1_sd","dev2_sd","corr")
for (i in 1:13) {
   SumObs[i,] <- ComputeSummary(HW3data[HW3data$Observer == i, -1])
}
```

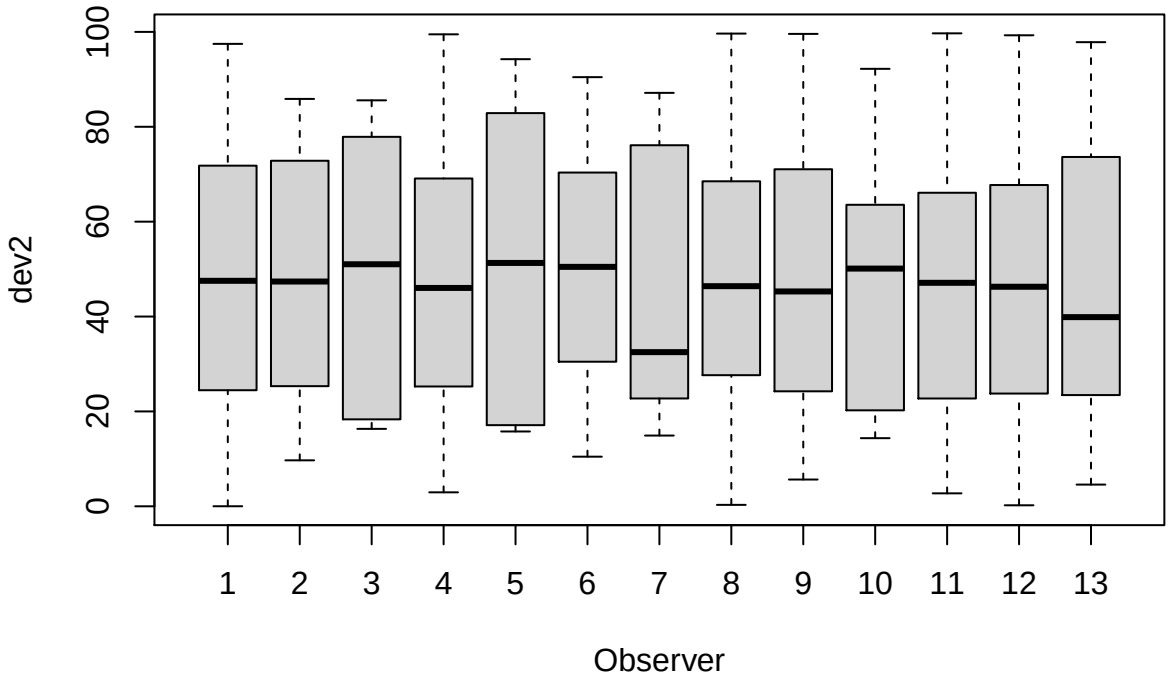
a. A single table of the means, sd, and correlation for each of the 13 Observers.

dev1_mean	dev2_mean	dev1_sd	dev2_sd	corr
54.26610	47.83472	16.76983	26.93974	-0.0641284
54.26873	47.83082	16.76924	26.93573	-0.0685864
54.26732	47.83772	16.76001	26.93004	-0.0683434
54.26327	47.83225	16.76514	26.93540	-0.0644719
54.26030	47.83983	16.76774	26.93019	-0.0603414
54.26144	47.83025	16.76590	26.93988	-0.0617148
54.26881	47.83545	16.76670	26.94000	-0.0685042
54.26785	47.83590	16.76676	26.93610	-0.0689797
54.26588	47.83150	16.76885	26.93861	-0.0686092
54.26734	47.83955	16.76896	26.93027	-0.0629611
54.26993	47.83699	16.76996	26.93768	-0.0694456
54.26692	47.83160	16.77000	26.93790	-0.0665752
54.26015	47.83972	16.76996	26.93000	-0.0655833

From the table, we can conclude that the means, standard deviations and correlations are basically the same for the same device across 13 observers.

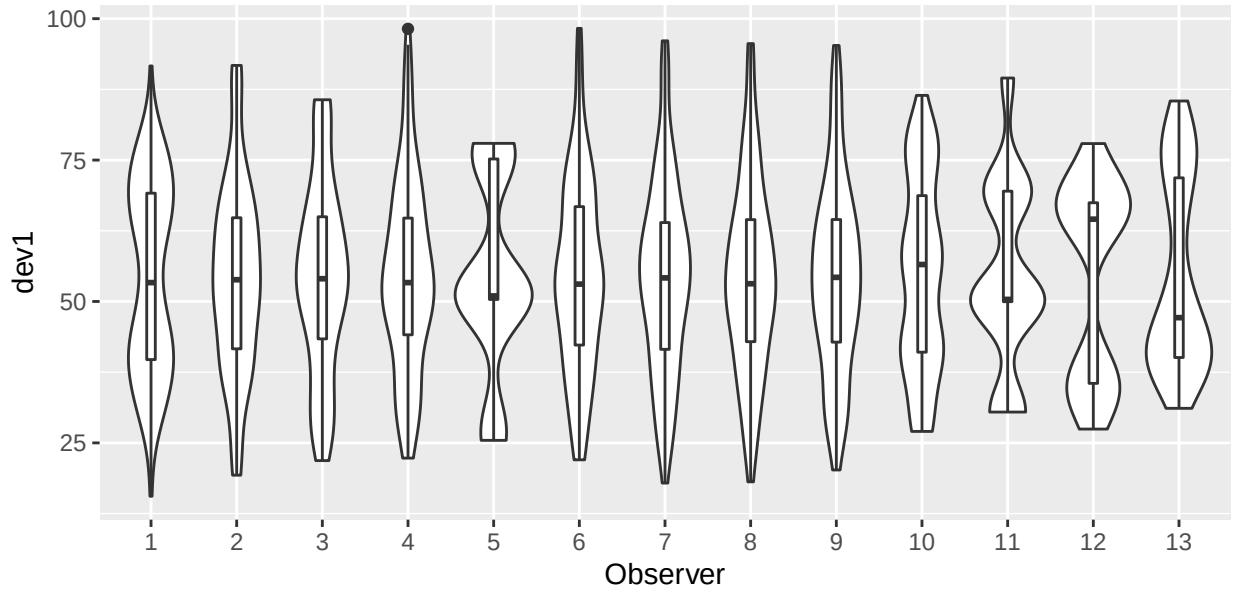
b. A box plot of dev, by Observer.

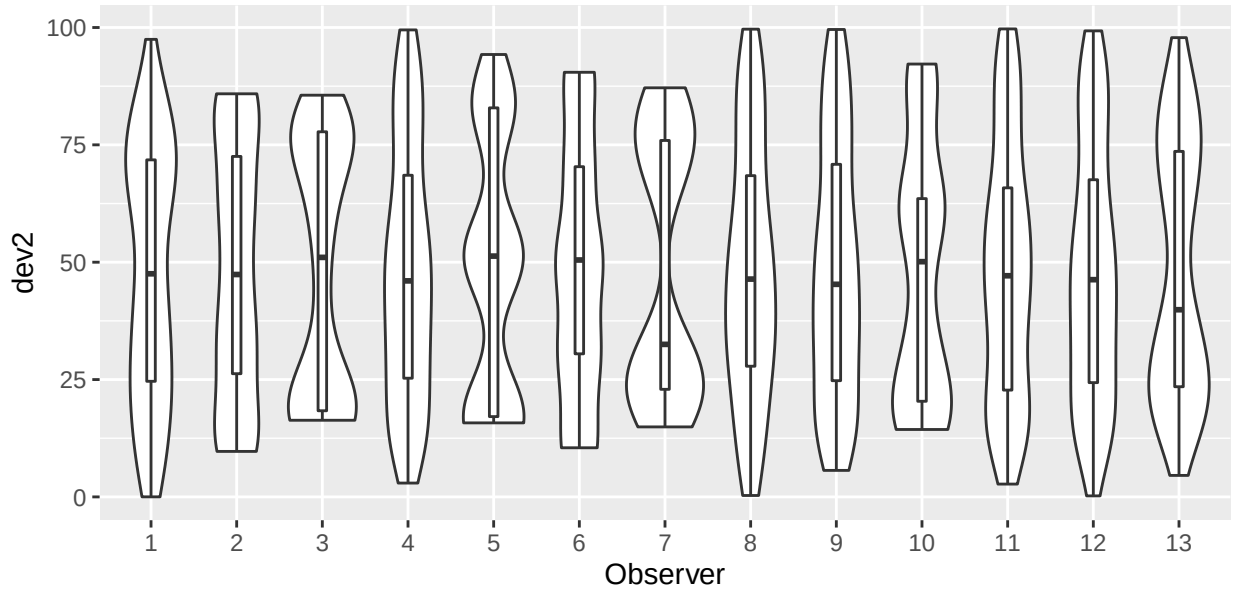




From the boxplot, we can conclude that the maximum, minimum, median and skewness are quite different for the same device accross 13 observers.

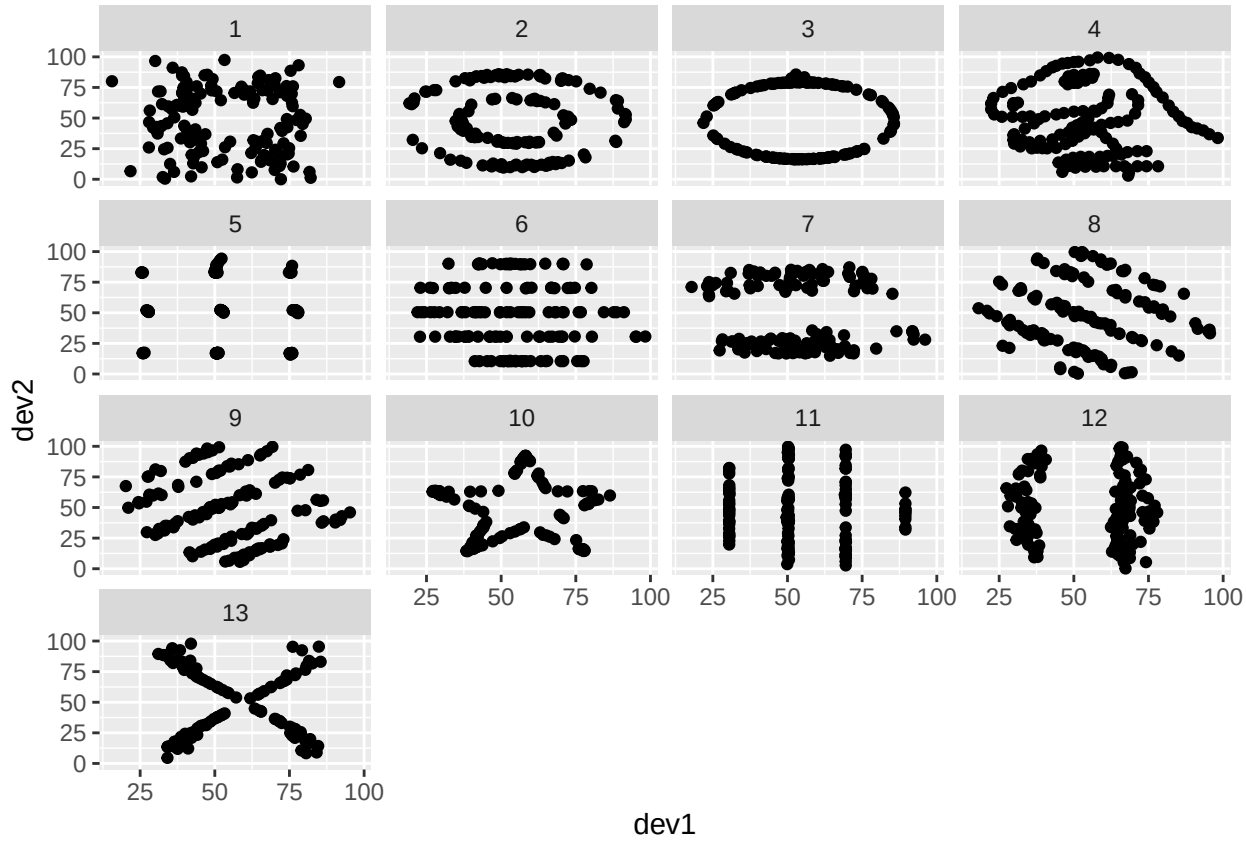
c. A violin plot of dev by Observer.





From the violin plots, we can conclude that the distributions of the same device across 13 observers are quite different despite that they have the same means, standard deviations and correlations. We will have a better understanding to the datasets by looking at the violin plots compared to the summary statistics in part (a).

d. a scatter plot of the data using ggplot, geom_points, and add facet_wrap on Observer.



Despite that the correlations between `dev1` and `dev2` are basically the same for each observer, I saw 13 totally different scatter plots of `dev1` vs. `dev2`. We can learn from this problem that we can not get a good understanding of a dataset by just looking at the mean, standard deiviation and correlation. There are much more important information about a dataset that can not be well presented by those three important yet simple statistics.

Problem 6

```
# A function that uses Reimann sums to approximate the integral:
# f(x) = \int_0^1 e^{-\frac{x^2}{2}}
# w: width
ApproxIntegral <- function(w){
  if(w <= 0 || w > 1) stop("the width must be within 0 and 1")
  # Here I'm using Right sum to approximate, by definition, we need to start from
  # the second element of the sequence
  approx_int <- sum(exp(- seq(0, 1, by = w)[-1] ^2 / 2)) * w
  return(approx_int)
}
```

Now use a looping construct (for or while) to loop through possible slice widths. Report the various slice widths used, the sum calculated, and the slice width necessary to obtain an answer within $1e^{-6}$ of the analytical solution.

Analytical solution:

$$\int_0^1 \exp\left(-\frac{x^2}{2}\right) dx = \sqrt{\frac{\pi}{2}}\,\mathrm{erf}\left(\frac{1}{\sqrt{2}}\right) \approx 0.8556244$$

```
# start from width = 0.5
w <- 0.5
# analytical solution obtained above
AnalSol<-0.8556244
# calculate the difference between approximation and analytical solution
error<-abs(ApproxIntegral(w)-AnalSol)
while (error > 1e-6) {
  w <- w/5
  approx_int <- ApproxIntegral(w)
  error<-abs(approx_int - AnalSol)
  cat("width:", w, "Integral Approximation:", approx_int, "\n")
}
```

```
## width: 0.1 Integral Approximation: 0.8354453
## width: 0.02 Integral Approximation: 0.8516695
## width: 0.004 Integral Approximation: 0.8548366
## width: 8e-04 Integral Approximation: 0.855467
## width: 0.00016 Integral Approximation: 0.8555929
## width: 3.2e-05 Integral Approximation: 0.8556181
## width: 6.4e-06 Integral Approximation: 0.8556231
## width: 1.28e-06 Integral Approximation: 0.8556241
```

```
cat("slice width necessary to obtain an answer within 1e-6 of the analytical solution:",w)
```

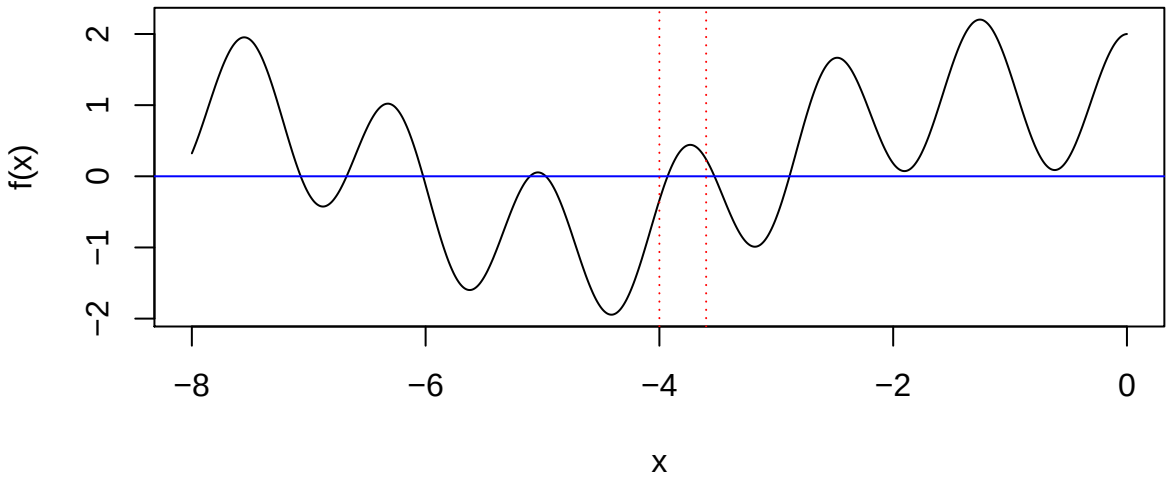
```
## slice width necessary to obtain an answer within 1e-6 of the analytical solution: 1.28e-06
```

Problem 7

Implement Newton's method

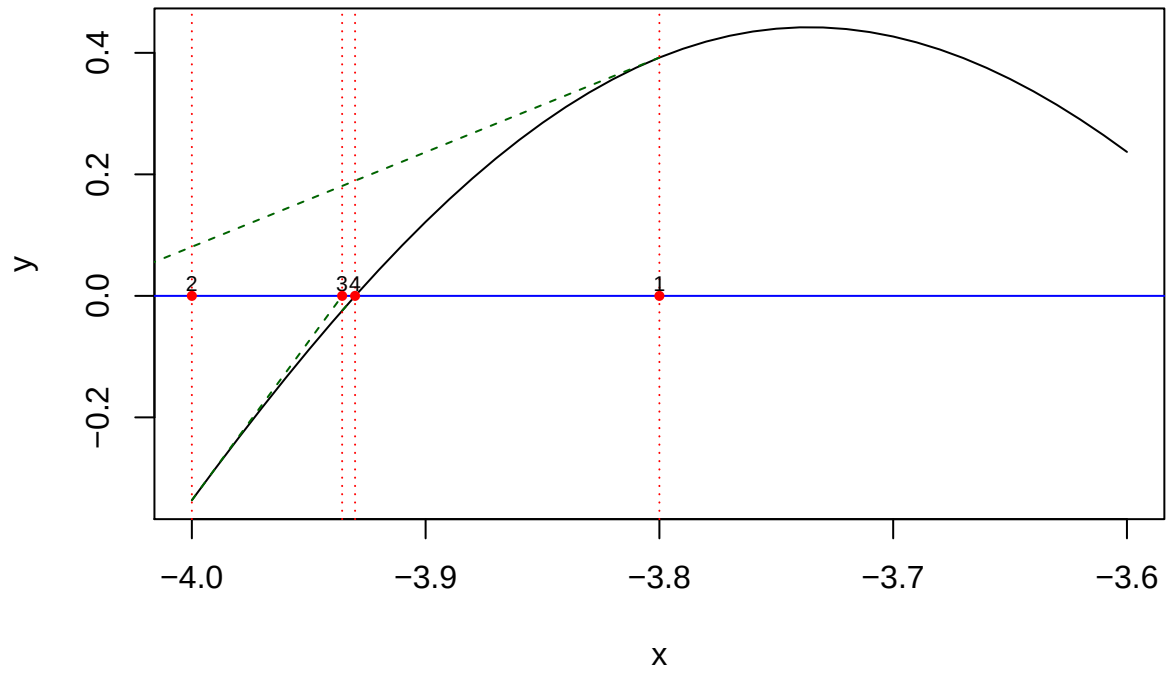
```
# function (1):
f <- function(x){
  func<-3^x-sin(x)+cos(5*x)
  return(func)
}
# First derivative:
df <- function(x){
  3^x*log(3) - cos(x) - 5*sin(5*x)
}
# Root finding using Newton's method
newton_root <- function(tol, a, b){
  # tol: tolerance used to terminate the loop
  # a,b: the ending points of the chosen interval [a,b]

  # Table that contains x_0,x_1,...,x_n+1. Initial guess of the root: (a+b)/2
  # The 1st row of the table is x_n in nth loop, 2nd row is x_n+1 in nth loop
  root<-matrix(c((a+b)/2,NA))
  # set maximum iteration times to prevent crash
  maxit <- 10000
  # plot the function
  x<-seq(a,b,0.01);y<-f(x)
  plot(x,y,type = "l", xlim = c(a,b))
  abline(h = 0, col="blue")
  #iteration times:
  it<-0

  for (i in 1:maxit) {
    it<-it+1
    # calculate x_n+1.
    temp <- root[1,i]-f(root[1,i])/df(root[1,i])
    # if x_n+1 < a, let x_n+1 = a
    # if x_n+1 > b, let x_n+1 = b
    if (temp < a) {root[2,i] <- a}
    else if (temp > b) {root[2,i] <- b}
    else {root[2,i] <- temp}

    # add lines and points to the plot
    segments(root[1,i],f(root[1,i]),temp,0,lty = "dashed",col = "darkgreen")
    abline(v = root[1,i],col="red",lty="dotted")
    text(root[1,i],0.02, labels = it,cex = 0.7)
    points(root[1,i], 0, pch = 16, col="red",cex = 0.7)

    # break when x_n+1 -x_n < tolerance
    if(abs(root[2,i] - root[1,i]) < tol) break
    # append the new roots
    root<-cbind(root,c(root[2,i],NA))
  }
  cat("Root=",root[2,i],"tolerance=",tol,"interval=[",a,",",b,"]","\n","root table")
  return(root)
}
```

A first skim at the function plot:



I will work on the root that lies between $[-4, -3.6]$.

```
newton_root(1e-3,-4,-3.6)
```



```
## Root= -3.930114 tolerance= 0.001 interval=[ -4 , -3.6 ]
##  root table
```

```
##      [,1]      [,2]      [,3]      [,4]
## [1,] -3.8 -4.000000 -3.935707 -3.930172
## [2,] -4.0 -3.935707 -3.930172 -3.930114
```

Problem 8

In this problem, we want to compare use of a for loop to using matrix operations in calculating these sums of squares. We will use data simulated using:

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

Without going too far into the Linear Regression material, we want to calculate SST =

$$\Sigma_{i=1}^{100}(y_i - \bar{y})^2$$

Please calculate this using:

a. accumulating values in a for loop

```
ybar<-mean(y)
LoopSST <- function(y){
  SST<-0
  for (i in 1:length(y)){
   SST <- SST + (y[i] - ybar)^2
  }
  return(SST)
}
LoopSST(y)
```

```
## [1] 20580.76
```

b. matrix operations only

```
# SST = (Y-Ybar)'(Y-Ybar)
MatSST <- function(y){
  SST<-t(y-ybar) %*% as.matrix(y-ybar)
  return(SST)
}
MatSST(y)
```

```
##          [,1]
## [1,] 20580.76
```

```
microbenchmark(result1<-LoopSST(y), result2<-MatSST(y),times = 100,unit = "ms")
```

```
## Unit: milliseconds
##                   expr      min        lq       mean    median        uq
##  result1 <- LoopSST(y) 0.007603 0.0077765 0.00958093 0.0079135 0.0119820
##   result2 <- MatSST(y) 0.005705 0.0060110 0.05631232 0.0076195 0.0101235
##       max neval
##  0.021978   100
##  4.801312   100
```

Theoretically, matrix operation takes less time than loop calculation. But I got opposite results: matrix operation takes moret time than calculation. I didn't figure out why this happened.