Homework 7

Hwasoo Shin

2019 10 12

Problem 2

(a)

The code won't work since the object "Boot" was never assigned. He used Boot_times to assign number of bootstraps. However, when he was making loops he sued "Boot" instead. To make the code work he should change "Boot" to "Boot_times", then it will work.

(b)

We can get bootstraps from the sample. When bootstrapping, we have to make sure that the size of bootstrap sample is same as the sample size.

```
Sensory<-read_table("https://www2.isye.gatech.edu/~jeffwu/wuhamadabook/data/Sensory.dat",skip=1)
```

```
## Parsed with column specification:
## cols(
##   `Item 1 2 3 4 5` = col_character()
## )
```

```
#we will use read_table with skip=1 option.
Sensory<-as.data.frame(Sensory)
#Change the data type into data.frame
SensoryName<-word(colnames(Sensory), sep=" ", 1:6)
#We will fetch the names of data. We use word function to get the variables.
Sensory<-Sensory %>% separate(colnames(Sensory), SensoryName,sep=" ")
```

```
## Warning: Expected 6 pieces. Missing pieces filled with `NA` in 20 rows [2,
## 3, 5, 6, 8, 9, 11, 12, 14, 15, 17, 18, 20, 21, 23, 24, 26, 27, 29, 30].
```

```
#Pipe operators (%>%). separate function will distribute the items in one column into
#multiple columns.
idx<-c(1:30)[-seq(1, 30, by = 3)]
#Since there are values where there isn't a item number, we will get rows without item numbers
Sensoryidx<-Sensory[idx, ]
#Getting rows without item numbers
Sensory[,1]<-rep(1:10, each = 3)
#Put item numbers on every row
Sensory[idx, 2:6]<-Sensoryidx
```

```
## Warning in `[<-.data.frame`(`*tmp*`, idx, 2:6, value = structure(list(Item
## = c("4.3", : provided 6 variables to replace 5 variables
```

```r
#This is our modified data
#We can also make a data which columns are order of items.

Sensory<-apply(Sensory,2,as.numeric) #Make all elements numeric
Sensory<-as.data.frame(Sensory) #Turn it into a data frame

parmat<-matrix(0,1000,6)
system.time({
for(i in 1:1000){
  idx<-sample(1:30,30,replace=TRUE)
  SenBoot<-Sensory[idx,]
  colnames(SenBoot)<-c("Item","Op1","Op2","Op3","Op4","Op5")
  parmat[i,]<-lm(data=SenBoot, Item~.)$coefficients
}
})
```

```
##    user  system elapsed
##    0.80    0.01    0.81
```

```r
#We can also use apply function to get this.

problem2<-function(x){
  x<-NULL
  idx<-sample(1:30,30,replace=TRUE)
  SenBoot<-Sensory[idx,]
  colnames(SenBoot)<-c("Item","Op1","Op2","Op3","Op4","Op5")
  parmat[i,]<-lm(data=SenBoot, Item~.)$coefficients
}

system.time({
sapply(1:1000,problem2)
})
```

```
##    user  system elapsed
##    0.81    0.00    0.82
```

```r
apply(parmat,2,mean) #Mean of parameters
```

```
## [1]  7.0203223  0.2468646 -0.8138915  0.9599156 -1.3560362  1.0376213
```

```r
apply(parmat,2,var) #Variance of parameters
```

```
## [1] 2.6042009 0.6286753 0.6891342 0.9090767 0.6892989 0.8909860
```

(c)

We will use "parallel" package to make clusters. Theoretically, it will boost up our code to run faster. We can do parallel computing since there are more than 1 core in the computer to operate programs.

```r
detectCores() #there are 8 cores in my laptop
```

```
## [1] 8
```

```r
Soocores<-6 #Therefore will use only 6 of it

problem2<-function(x){
  idx<-sample(1:30,30,replace=TRUE)
  SenBoot<-Sensory[idx,]
  colnames(SenBoot)<-c("Item","Op1","Op2","Op3","Op4","Op5")
  k<-parmat[i,]<-lm(data=SenBoot, Item~.)$coefficients
  return(k)
}


system.time({
  cl<-makeCluster(8)
  clusterExport(cl,"Sensory")
  parSapply(cl,1:1000,function(x){
    x<-NULL
  idx<-sample(1:30,30,replace=TRUE)
  SenBoot<-Sensory[idx,]
  colnames(SenBoot)<-c("Item","Op1","Op2","Op3","Op4","Op5")
  k<-lm(data=SenBoot, Item~.)$coefficients
})
  stopCluster(cl)
})
```

```
##    user  system elapsed
##    0.03    0.03    2.22
```

It took far less time than using the for loop alone.

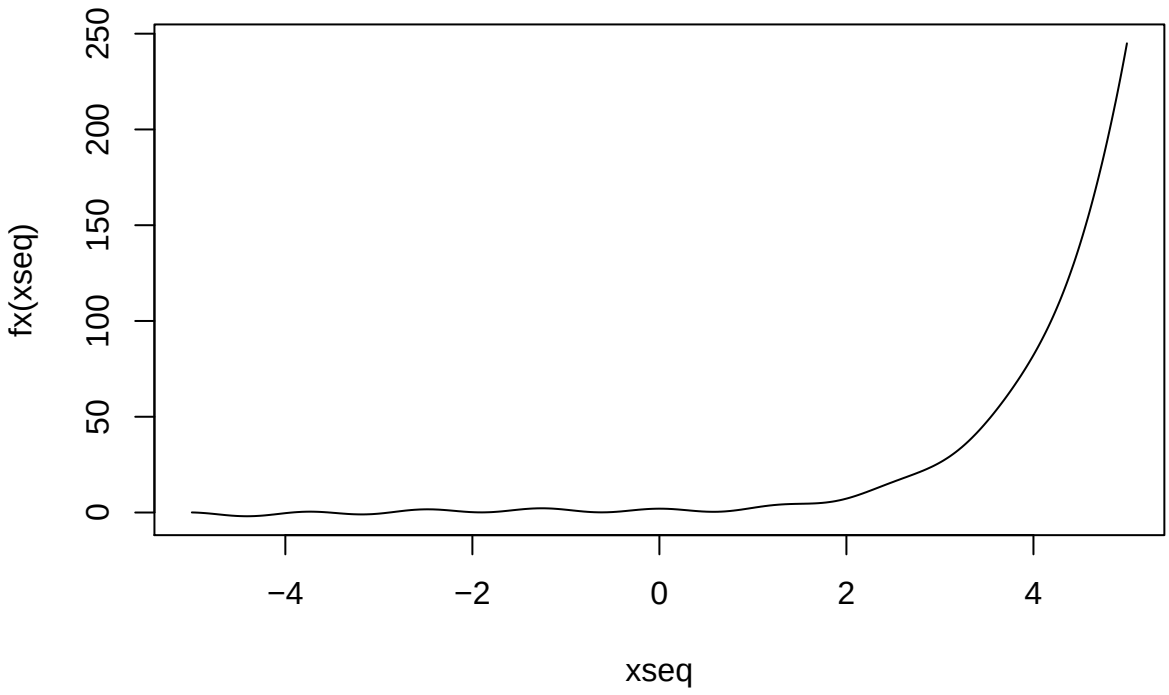
Problem 3

We will use Newton's method to get the solutions, but with parallel computing for the speed.

(a)

First we start with only the apply function.

```r
fx<-function(x){3^(x)-sin(x)+cos(5*x)} #The original function for the problem
xseq<-seq(-5,5,by=0.01)
plot(xseq,fx(xseq),type='l')
```




```
#This is the graph of f(x). We can assume that it will have solutions when x<0.
#Therefore, we can set the interval from -20 to 10.


problem3<-function(k){
  for(i in 1:100){
    k<-k-(3^(k)-sin(k)+cos(5*k))/(log(3)*3^(k)-cos(k)-5*sin(5*k))
  }
return(k)
} #Using Newton Method to get the solution.
#We will loop 100 times to find the solutions.

possol<-seq(-20,10,length=1000)
system.time({possol2<-sapply(possol,problem3)})
```

```
##    user  system elapsed
##    0.13    0.00    0.12
```

(b)

This time, we use the same thing but will use parallel computing.

```
problem3<-function(k){
  for(i in 1:100){
```

```
    k<-k-(3^(k)-sin(k)+cos(5*k))/(log(3)*3^(k)-cos(k)-5*sin(5*k))
  }
return(k)
}

system.time({
cl<-makeCluster(8)
clusterExport(cl,"problem3")
parSapply(cl,seq(-20,10,length=1000),problem3)
stopCluster(cl)
})
```

```
##    user  system elapsed
##    0.03    0.02    1.82
```

We can see the time has slightly reduced to run the code. However, the elasped time has actually increased. Apparantely the overall time to process the code took more on the second step. This is because CPU has to spend more time to express the value. In my opinion, the overall speed will be faster on second method than the first one when the syntax we use is more sophisticated.

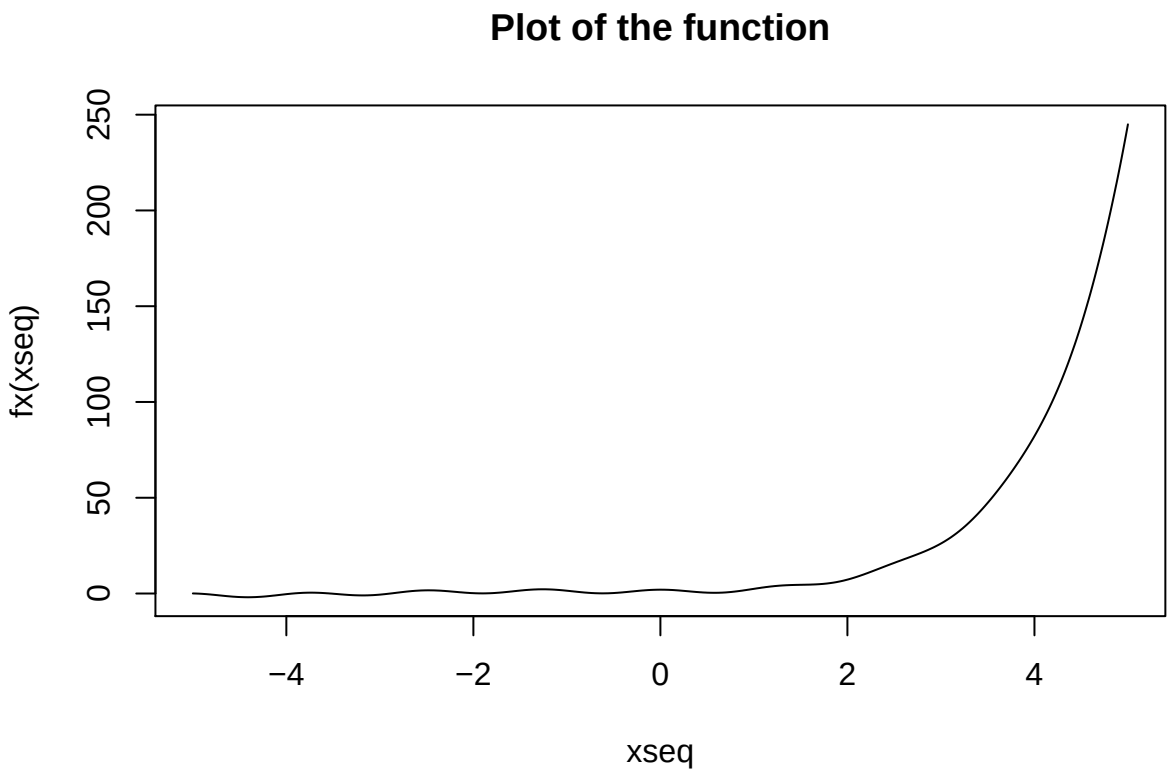
Problem 4

Using descent gradient method, we will start with various initial points and step sizes. The tolerance is 1e-9, which is, if the difference between previous value and current value is smaller than 1e-9, our iteration to find the solution will stop.

```
posx<-seq(-20,-2,length=10000)

problem4<-function(k){
  kpre<-0
  i<-0
  while(abs(k-kpre)>1e-9){
    kpre<-k
    k<-k-1e-7*(log(3)*3^(k)-cos(k)-5*sin(5*k))
    i<-i+1
    if(i>1e8){break} #Avoid too much loops
  }
return(c(k,i))
}

plot(xseq,fx(xseq),type='l',main="Plot of the function")
```




```
system.time({
cl<-makeCluster(7)
clusterExport(cl,varlist=c("problem4","posx"))
xsol<-parSapply(cl,posx,problem4)
stopCluster(cl)
})
```

```
##    user  system elapsed
##    0.05    0.02 8694.08
```

```
summary(xsol[1,]) #Summary of solutions
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -19.445 -15.748 -10.694 -10.994  -6.878  -1.904
```

```
summary(xsol[2,]) #Summary of loops
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##       1 2425518 2776504 2774386 3127602 5562656
```

```
sol5m<-sum(xsol[2,]<5000000) #Number of solutions that took less than 5M loops
sol5m
```

```
## [1] 9982
```

(a)

We can see that it takes a long time to run this whole syntax. One way to avoid is to change the stopping rule, for example, we can change the tolerance to 1e-7. Then the function will loop less and will help us to get results faster. However, the results might be incorrect since the tolerance is too big. We can also increase the step. This can help us avoid the function only finding the local minimum, but also prevents the function from getting enough close to any minimum.

(b)

Number of solutions that took less than 5,000,000 loops is 9982 when setting step size 1e-7 and tolerance 1e-9. We can slightly change the stopping rule to check how many loops it took to get the solution. To check how many iterations we needed, we will use a function to calculate loops with different starting values.

```
startb<-c(-10,-8,-6,-4,-2)

problem4b<-function(k){
  initk<-k
  kpre<-0
  i<-0
  m<-c(1e-10,1e-9,1e-8,1e-7)
  s<-c(1e-8,1e-7,1e-6)
  kM<-matrix(0,4,3)
  rownames(kM)<-c("tol-10","tol-9","tol-8","tol-7")
  colnames(kM)<-c("step-8","step-7","step-6")
  for(si in 1:3){
  for(j in 1:4){
  while(abs(k-kpre)>m[j]){
    kpre<-k
    k<-k-s[si]*(log(3)*3^(k)-cos(k)-5*sin(5*k))
    i<-i+1
    if(i>1e8){break} #Avoid too much loops
  }
    kM[j,si]=i
    i<-0
    k<-initk
    kpre<-0
  }
  }
return(kM)
}

lapply(startb,problem4b)
```

```
## [[1]]
##          step-8  step-7 step-6
## tol-10 40342614 4972860 591130
## tol-9  30942967 4034258 497284
## tol-8         1 3094297 403425
## tol-7         1       1 309430
##
## [[2]]
##          step-8  step-7 step-6
```

```
## tol-10 25490019 3508302 446761
## tol-9  15901017 2548997 350827
## tol-8   6311156 1590101 254898
## tol-7         1  631116 159010
##
## [[3]]
##          step-8  step-7 step-6
## tol-10 27832775 3693162 460311
## tol-9  18742951 2783274 369313
## tol-8   9709862 1874294 278326
## tol-7         1  970987 187429
##
## [[4]]
##          step-8  step-7 step-6
## tol-10 28292536 3717645 460604
## tol-9  19411663 2829251 371762
## tol-8  10531078 1941166 282924
## tol-7         1 1053108 194116
##
## [[5]]
##          step-8  step-7 step-6
## tol-10 22795453 3235171 419070
## tol-9  13232105 2279544 323515
## tol-8   3553720 1323210 227953
## tol-7         1  355373 132321
```

Each list represents the number of loops that took when the starting value is -10, -8, -6, -4 and -2 respectively. The rows represent tolerance, from 1e-10 to 1e-7. On the other hand, columns represent step size, from 1e-8 to 1e-6. From the results, we can see that when the step size is smaller than tolerance, the loop won't work. Also, when tolerance or step size are smaller it will cause the function to go through more loops. For example, when the initial value is -2, number of cases that had less than 5M loops is 10. The only cases that had more than 5M loops were when tolerance and step size are too small. We can conclude that when the step size and tolerance are small, gradient descent method will have to be repeated more than usual.