Homework 4

Due October 14, 2020

2020-10-14

For each assignment, turn in by the due date/time. Late assignments must be arranged prior to submission. In every case, assignments are to be typed neatly using proper English in Markdown.

The last couple of weeks, we spoke about vector/matrix operations in R, discussed how the apply family of functions can assist in row/column operations, and how parallel computing in R is enabled. Combining this with previous topics, we can write functions using our Good Programming Practices style and adhere to Reproducible Research principles to create fully functional, redable and reproducible code based analysis in R. In this homework, we will put this all together and actually analyze some data. Remember to adhere to both Reproducible Research and Good Programming Practices, ie describe what you are doing and comment/indent code where necessary.

R Vector/matrix manipulations and math, speed considerations R's Apply family of functions Parallel computing in R, foreach and dopar

Problem 1: Using the dual nature to our advantage

Sometimes using a mixture of true matrix math plus component operations cleans up our code giving better readibility. Suppose we wanted to form the following computation:

- $while(abs(\Theta_0^i - \Theta_0^{i-1})$ AND $abs(\Theta_1^i - \Theta_1^{i-1}) > tolerance)$ {

$$\begin{aligned}\Theta_0^i &= \Theta_0^{i-1} - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_0(x_i) - y_i) \\ \Theta_1^i &= \Theta_1^{i-1} - \alpha\frac{1}{m}\sum_{i=1}^{m}((h_0(x_i) - y_i)x_i)\end{aligned}$$

}

Where $h_0(x) = \Theta_0 + \Theta_1 x$.

Given $\mathbf{X}$ and $\vec{h}$ below, implement the above algorithm and compare the results with lm(h~0+$\mathbf{X}$). State the tolerance used and the step size, α.

```
    set.seed(1256)
    theta <- as.matrix(c(1,2),nrow=2)
    X <- cbind(1,rep(1:10,10))
    h <- X%*%theta+rnorm(100,0,0.2)
```

```
set.seed(1256)
theta <- as.matrix(c(1,2),nrow=2)
# True Parameter
theta
```

```
##      [,1]
## [1,]    1
## [2,]    2
```

```r
X <- cbind(1,rep(1:10,10))
h <- X%*%theta+rnorm(100,0,0.2)
h_0 <- X%*%theta


Dual <- function(theta,X,y,alpha,tol,maxiter){
  # theta is initial value
  up_theta <- theta
  up_theta[1] <- theta[1]-alpha*mean(X%*%theta-y)
  up_theta[2] <- theta[2]-alpha*t(X%*%theta-y)%*%X[,2]/nrow(X)
  i <- 1
  while (abs(up_theta[1]-theta[1])>tol & abs(up_theta[2]-theta[2])>tol & i<maxiter){
    theta <- up_theta
    up_theta[1] <- theta[1]-alpha*mean(X%*%theta-y)
    up_theta[2] <- theta[2]-alpha*t(X%*%theta-y)%*%X[,2]/nrow(X)
    i <- i+1
  }
  output <- c(i,up_theta)
  names(output)=c("iteration", "theta_0_hat", "theta_1_hat")
  return(output)
}

## Estimating parameter using regression
regfit <- lm(h~0+X)
regfit$coefficients
```

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

```r
# error
sum((regfit$coefficients-theta)^2)
```

```
## [1] 0.0009283884
```

```r
# Estimating parameter using regression
dualfit <- Dual(matrix(1,2,1),X,h,0.001,0.0001,10000)
dualfit
```

```
##   iteration theta_0_hat theta_1_hat
##   94.000000    1.133761    1.954940
```

```r
# error
sum((dualfit[2:3]-theta)^2)
```

```
## [1] 0.01992248
```

True model is $h = X\theta + \epsilon$, where $\theta = (1, 2)^T$ and $\epsilon_i \overset{iid}{\sim} N(0, 0.2^2)$. We want to estimate θ by two methods, where one is given algorithm in the problem and the other is linear regression. Both methods estimated

true θ well, but the performance of regression is better than that of our algorithm. I think if we set proper initial value, step size(α), and tolerance level in the dual algorithm, we can get similar performance as linear regression.

Problem 2

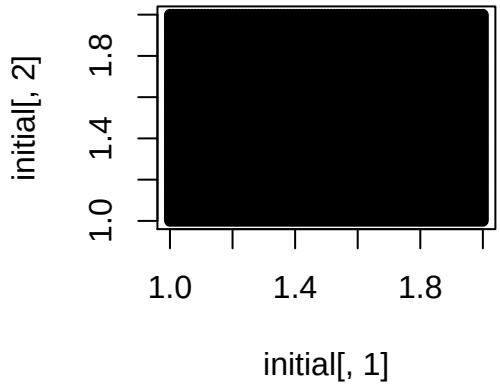
The above algorithm is called Gradient Descent. This algorithm, like Newton's method, has "hyperparameters" that are determined outside the algorithm and there are no set rules for determing what settings to use. For gradient descent, you need to set a start value, a step size and tolerance.

Part a. Using a step size of $1e^{-7}$ and tolerance of $1e^{-9}$, try 10000 different combinations of start values for β_0 and β_1 across the range of possible β's +/-1 from true determined in Problem 2, making sure to take advantages of parallel computing opportunities. In my try at this, I found starting close to true took 1.1M iterations, so set a stopping rule for 5M. Report the min and max number of iterations along with the starting values for those cases. Also report the average and stdev obtained across all 10000 β's.

```
# 10000 initial values
grid <- seq(1,2,length=100)
initial <- as.data.frame(rbind(cbind(grid,grid), t(combn(grid,2)), t(combn(grid,2))[,c(2,1)]))
initial[,3:5]=0
names(initial) <- c("Initial_theta_0", "Initial_theta_1", "iteration", "theta_0_hat", "theta_1_hat")
head(initial)
```

```
##   Initial_theta_0 Initial_theta_1 iteration theta_0_hat theta_1_hat
## 1        1.000000        1.000000         0           0           0
## 2        1.010101        1.010101         0           0           0
## 3        1.020202        1.020202         0           0           0
## 4        1.030303        1.030303         0           0           0
## 5        1.040404        1.040404         0           0           0
## 6        1.050505        1.050505         0           0           0
```

```
plot(initial[,1],initial[,2],pch=20)
```



I made 10000 initial values, which are grids for combination of $\theta_0^0 : 1 \sim 2$ and $\theta_1^0 : 1 \sim 2$.

```
# Parallel computing for initial values
library(foreach)
library(doParallel)

cores=detectCores()
cl <- makeCluster(cores[1]-1)
registerDoParallel(cl)
chunk=foreach(i = 1:nrow(initial), .combine= 'rbind') %dopar% {
  dualfit <- Dual(matrix(c(initial[i,1],initial[i,2]),2,1),X,h,0.1^7,0.1^9,5000000)
  dualfit
}
stopCluster(cl)
initial[,3:5]=chunk

save.image(file="Homework_4_gradient.RData")
```

I performed gradient descent algorithm for 10000 initial values using parallel computing. Since the computation time is so long, I implemented above code in my R (It took many hours) and saved the result in Rdata file. I bring the Rdata file in the RMarkdown.

```
setwd("~/")
load("~/Homework_4_gradient.RData")

# Minimum iterations
initial[initial$iteration==1,]
```

```
##      Initial_theta_0 Initial_theta_1 iteration theta_0_hat theta_1_hat
## 489         1.030303        1.989899         1    1.030303    1.989899
## 584         1.040404        1.989899         1    1.040404    1.989899
## 678         1.050505        1.989899         1    1.050505    1.989899
## 953         1.080808        1.979798         1    1.080808    1.979798
## 1043        1.090909        1.979798         1    1.090909    1.979798
## 1307        1.121212        1.979798         1    1.121212    1.979798
## 1477        1.141414        1.969697         1    1.141414    1.969697
## 1561        1.151515        1.969697         1    1.151515    1.969697
## 1886        1.191919        1.959596         1    1.191919    1.959596
## 1887        1.191919        1.969697         1    1.191919    1.969697
## 1965        1.202020        1.959596         1    1.202020    1.959596
## 2344        1.252525        1.949495         1    1.252525    1.949495
## 2417        1.262626        1.949495         1    1.262626    1.949495
## 2418        1.262626        1.959596         1    1.262626    1.959596
## 2698        1.303030        1.939394         1    1.303030    1.939394
## 2766        1.313131        1.939394         1    1.313131    1.939394
## 2900        1.333333        1.949495         1    1.333333    1.949495
## 3090        1.363636        1.929293         1    1.363636    1.929293
## 3152        1.373737        1.929293         1    1.373737    1.929293
## 3333        1.404040        1.939394         1    1.404040    1.939394
## 3389        1.414141        1.919192         1    1.414141    1.919192
## 3446        1.424242        1.919192         1    1.424242    1.919192
## 3715        1.474747        1.909091         1    1.474747    1.909091
## 3717        1.474747        1.929293         1    1.474747    1.929293
```

```
## 3766        1.484848        1.909091         1    1.484848    1.909091
## 3959        1.525253        1.898990         1    1.525253    1.898990
## 4005        1.535354        1.898990         1    1.535354    1.898990
## 4052        1.545455        1.919192         1    1.545455    1.919192
## 4219        1.585859        1.888889         1    1.585859    1.888889
## 4259        1.595960        1.888889         1    1.595960    1.888889
## 4338        1.616162        1.909091         1    1.616162    1.909091
## 4408        1.636364        1.878788         1    1.636364    1.878788
## 4443        1.646465        1.878788         1    1.646465    1.878788
## 4575        1.686869        1.898990         1    1.686869    1.898990
## 4602        1.696970        1.868687         1    1.696970    1.868687
## 4631        1.707071        1.868687         1    1.707071    1.868687
## 4736        1.747475        1.858586         1    1.747475    1.858586
## 4760        1.757576        1.858586         1    1.757576    1.858586
## 4763        1.757576        1.888889         1    1.757576    1.888889
## 4864        1.808081        1.848485         1    1.808081    1.848485
## 4882        1.818182        1.848485         1    1.818182    1.848485
## 4902        1.828283        1.878788         1    1.828283    1.878788
## 9844        1.969697        1.818182         1    1.969697    1.818182
## 9845        1.979798        1.818182         1    1.979798    1.818182
## 9856        1.919192        1.828283         1    1.919192    1.828283
## 9857        1.929293        1.828283         1    1.929293    1.828283
## 9866        1.858586        1.838384         1    1.858586    1.838384
## 9867        1.868687        1.838384         1    1.868687    1.838384
## 9906        1.969697        1.858586         1    1.969697    1.858586
## 9912        1.898990        1.868687         1    1.898990    1.868687
```

```
# Maximum iterations
initial[initial$iteration==5000000,]
```

```
##      Initial_theta_0 Initial_theta_1 iteration theta_0_hat theta_1_hat
## 85          1.848485        1.848485     5e+06    1.764287    1.887409
## 86          1.858586        1.858586     5e+06    1.771918    1.886313
## 87          1.868687        1.868687     5e+06    1.779549    1.885217
## 2964        1.343434        1.939394     5e+06    1.307246    1.953059
## 3028        1.353535        1.939394     5e+06    1.316156    1.951779
## 3091        1.363636        1.939394     5e+06    1.325067    1.950499
## 3153        1.373737        1.939394     5e+06    1.333977    1.949219
## 3213        1.383838        1.929293     5e+06    1.344168    1.947756
## 3214        1.383838        1.939394     5e+06    1.342888    1.947940
## 3273        1.393939        1.929293     5e+06    1.353078    1.946476
## 3274        1.393939        1.939394     5e+06    1.351798    1.946660
## 3332        1.404040        1.929293     5e+06    1.361989    1.945196
## 3390        1.414141        1.929293     5e+06    1.370899    1.943916
## 3447        1.424242        1.929293     5e+06    1.379810    1.942636
## 3502        1.434343        1.919192     5e+06    1.390000    1.941172
## 3503        1.434343        1.929293     5e+06    1.388720    1.941356
## 3557        1.444444        1.919192     5e+06    1.398910    1.939892
## 3558        1.444444        1.929293     5e+06    1.397630    1.940076
## 3611        1.454545        1.919192     5e+06    1.407821    1.938613
## 3612        1.454545        1.929293     5e+06    1.406541    1.938796
## 3664        1.464646        1.919192     5e+06    1.416731    1.937333
## 3665        1.464646        1.929293     5e+06    1.415451    1.937516
## 3716        1.474747        1.919192     5e+06    1.425642    1.936053
```

```
## 3767       1.484848       1.919192    5e+06    1.434552    1.934773
## 3816       1.494949       1.909091    5e+06    1.444742    1.933309
## 3817       1.494949       1.919192    5e+06    1.443463    1.933493
## 3865       1.505051       1.909091    5e+06    1.453653    1.932029
## 3866       1.505051       1.919192    5e+06    1.452373    1.932213
## 3913       1.515152       1.909091    5e+06    1.462563    1.930749
## 3914       1.515152       1.919192    5e+06    1.461283    1.930933
## 3960       1.525253       1.909091    5e+06    1.471474    1.929469
## 3961       1.525253       1.919192    5e+06    1.470194    1.929653
## 4006       1.535354       1.909091    5e+06    1.480384    1.928189
## 4007       1.535354       1.919192    5e+06    1.479104    1.928373
## 4050       1.545455       1.898990    5e+06    1.490575    1.926726
## 4051       1.545455       1.909091    5e+06    1.489295    1.926910
## 4094       1.555556       1.898990    5e+06    1.499485    1.925446
## 4095       1.555556       1.909091    5e+06    1.498205    1.925630
## 4137       1.565657       1.898990    5e+06    1.508395    1.924166
## 4138       1.565657       1.909091    5e+06    1.507116    1.924350
## 4179       1.575758       1.898990    5e+06    1.517306    1.922886
## 4180       1.575758       1.909091    5e+06    1.516026    1.923070
## 4220       1.585859       1.898990    5e+06    1.526216    1.921606
## 4221       1.585859       1.909091    5e+06    1.524936    1.921790
## 4260       1.595960       1.898990    5e+06    1.535127    1.920326
## 4261       1.595960       1.909091    5e+06    1.533847    1.920510
## 4298       1.606061       1.888889    5e+06    1.545317    1.918862
## 4299       1.606061       1.898990    5e+06    1.544037    1.919046
## 4300       1.606061       1.909091    5e+06    1.542757    1.919230
## 4336       1.616162       1.888889    5e+06    1.554228    1.917583
## 4337       1.616162       1.898990    5e+06    1.552948    1.917766
## 4373       1.626263       1.888889    5e+06    1.563138    1.916303
## 4374       1.626263       1.898990    5e+06    1.561858    1.916487
## 4409       1.636364       1.888889    5e+06    1.572048    1.915023
## 4410       1.636364       1.898990    5e+06    1.570769    1.915207
## 4444       1.646465       1.888889    5e+06    1.580959    1.913743
## 4445       1.646465       1.898990    5e+06    1.579679    1.913927
## 4477       1.656566       1.878788    5e+06    1.591149    1.912279
## 4478       1.656566       1.888889    5e+06    1.589869    1.912463
## 4479       1.656566       1.898990    5e+06    1.588589    1.912647
## 4510       1.666667       1.878788    5e+06    1.600060    1.910999
## 4511       1.666667       1.888889    5e+06    1.598780    1.911183
## 4512       1.666667       1.898990    5e+06    1.597500    1.911367
## 4542       1.676768       1.878788    5e+06    1.608970    1.909719
## 4543       1.676768       1.888889    5e+06    1.607690    1.909903
## 4544       1.676768       1.898990    5e+06    1.606410    1.910087
## 4573       1.686869       1.878788    5e+06    1.617881    1.908439
## 4574       1.686869       1.888889    5e+06    1.616601    1.908623
## 4603       1.696970       1.878788    5e+06    1.626791    1.907160
## 4604       1.696970       1.888889    5e+06    1.625511    1.907343
## 4632       1.707071       1.878788    5e+06    1.635702    1.905880
## 4633       1.707071       1.888889    5e+06    1.634422    1.906063
## 4659       1.717172       1.868687    5e+06    1.645892    1.904416
## 4660       1.717172       1.878788    5e+06    1.644612    1.904600
## 4661       1.717172       1.888889    5e+06    1.643332    1.904784
## 4686       1.727273       1.868687    5e+06    1.654802    1.903136
## 4687       1.727273       1.878788    5e+06    1.653522    1.903320
```

```
## 4688       1.727273       1.888889    5e+06    1.652243    1.903504
## 4712       1.737374       1.868687    5e+06    1.663713    1.901856
## 4713       1.737374       1.878788    5e+06    1.662433    1.902040
## 4714       1.737374       1.888889    5e+06    1.661153    1.902224
## 4737       1.747475       1.868687    5e+06    1.672623    1.900576
## 4738       1.747475       1.878788    5e+06    1.671343    1.900760
## 4739       1.747475       1.888889    5e+06    1.670063    1.900944
## 4761       1.757576       1.868687    5e+06    1.681534    1.899296
## 4762       1.757576       1.878788    5e+06    1.680254    1.899480
## 4783       1.767677       1.858586    5e+06    1.691724    1.897833
## 4784       1.767677       1.868687    5e+06    1.690444    1.898016
## 4785       1.767677       1.878788    5e+06    1.689164    1.898200
## 4805       1.777778       1.858586    5e+06    1.700634    1.896553
## 4806       1.777778       1.868687    5e+06    1.699355    1.896736
## 4807       1.777778       1.878788    5e+06    1.698075    1.896920
## 4826       1.787879       1.858586    5e+06    1.709545    1.895273
## 4827       1.787879       1.868687    5e+06    1.708265    1.895457
## 4828       1.787879       1.878788    5e+06    1.706985    1.895640
## 4846       1.797980       1.858586    5e+06    1.718455    1.893993
## 4847       1.797980       1.868687    5e+06    1.717175    1.894177
## 4848       1.797980       1.878788    5e+06    1.715896    1.894361
## 4865       1.808081       1.858586    5e+06    1.727366    1.892713
## 4866       1.808081       1.868687    5e+06    1.726086    1.892897
## 4867       1.808081       1.878788    5e+06    1.724806    1.893081
## 4883       1.818182       1.858586    5e+06    1.736276    1.891433
## 4884       1.818182       1.868687    5e+06    1.734996    1.891617
## 4885       1.818182       1.878788    5e+06    1.733716    1.891801
## 4899       1.828283       1.848485    5e+06    1.746467    1.889969
## 4900       1.828283       1.858586    5e+06    1.745187    1.890153
## 4901       1.828283       1.868687    5e+06    1.743907    1.890337
## 4915       1.838384       1.848485    5e+06    1.755377    1.888689
## 4916       1.838384       1.858586    5e+06    1.754097    1.888873
## 4917       1.838384       1.868687    5e+06    1.752817    1.889057
## 4931       1.848485       1.858586    5e+06    1.763008    1.887593
## 4932       1.848485       1.868687    5e+06    1.761728    1.887777
## 4946       1.858586       1.868687    5e+06    1.770638    1.886497
## 9846       1.989899       1.818182    5e+06    1.892873    1.868939
## 9847       2.000000       1.818182    5e+06    1.901784    1.867659
## 9858       1.939394       1.828283    5e+06    1.847041    1.875523
## 9859       1.949495       1.828283    5e+06    1.855952    1.874243
## 9860       1.959596       1.828283    5e+06    1.864862    1.872963
## 9861       1.969697       1.828283    5e+06    1.873773    1.871683
## 9862       1.979798       1.828283    5e+06    1.882683    1.870403
## 9863       1.989899       1.828283    5e+06    1.891594    1.869123
## 9864       2.000000       1.828283    5e+06    1.900504    1.867843
## 9868       1.878788       1.838384    5e+06    1.792299    1.883386
## 9869       1.888889       1.838384    5e+06    1.801209    1.882106
## 9870       1.898990       1.838384    5e+06    1.810120    1.880826
## 9871       1.909091       1.838384    5e+06    1.819030    1.879546
## 9872       1.919192       1.838384    5e+06    1.827940    1.878266
## 9873       1.929293       1.838384    5e+06    1.836851    1.876986
## 9874       1.939394       1.838384    5e+06    1.845761    1.875707
## 9875       1.949495       1.838384    5e+06    1.854672    1.874427
## 9876       1.959596       1.838384    5e+06    1.863582    1.873147
```

```
## 9877        1.969697        1.838384     5e+06    1.872493    1.871867
## 9878        1.979798        1.838384     5e+06    1.881403    1.870587
## 9879        1.989899        1.838384     5e+06    1.890314    1.869307
## 9880        2.000000        1.838384     5e+06    1.899224    1.868027
## 9881        1.858586        1.848485     5e+06    1.773198    1.886130
## 9882        1.868687        1.848485     5e+06    1.782108    1.884850
## 9883        1.878788        1.848485     5e+06    1.791019    1.883570
## 9884        1.888889        1.848485     5e+06    1.799929    1.882290
## 9885        1.898990        1.848485     5e+06    1.808840    1.881010
## 9886        1.909091        1.848485     5e+06    1.817750    1.879730
## 9887        1.919192        1.848485     5e+06    1.826661    1.878450
## 9888        1.929293        1.848485     5e+06    1.835571    1.877170
## 9889        1.939394        1.848485     5e+06    1.844481    1.875890
## 9890        1.949495        1.848485     5e+06    1.853392    1.874610
## 9891        1.959596        1.848485     5e+06    1.862302    1.873331
## 9892        1.969697        1.848485     5e+06    1.871213    1.872051
## 9893        1.979798        1.848485     5e+06    1.880123    1.870771
## 9894        1.989899        1.848485     5e+06    1.889034    1.869491
## 9895        2.000000        1.848485     5e+06    1.897944    1.868211
## 9896        1.868687        1.858586     5e+06    1.780828    1.885034
## 9897        1.878788        1.858586     5e+06    1.789739    1.883754
## 9898        1.888889        1.858586     5e+06    1.798649    1.882474
## 9899        1.898990        1.858586     5e+06    1.807560    1.881194
## 9900        1.909091        1.858586     5e+06    1.816470    1.879914
## 9901        1.919192        1.858586     5e+06    1.825381    1.878634
## 9902        1.929293        1.858586     5e+06    1.834291    1.877354
## 9903        1.939394        1.858586     5e+06    1.843202    1.876074
## 9904        1.949495        1.858586     5e+06    1.852112    1.874794
## 9905        1.959596        1.858586     5e+06    1.861022    1.873514
## 9910        1.878788        1.868687     5e+06    1.788459    1.883937
## 9911        1.888889        1.868687     5e+06    1.797369    1.882658
```

We can get initial values with the minimum and the maximum iterations. The minimum iterations is 1 and the maximum iterations is 5M. In the table, Initial_theta_0 and Initial_theta_1 columns are for initial values (θ_0^0, θ_1^0) and iteration column is for the number of iterations in the algorithm, and theta_0_hat and theta_1_hat columns are for estimate $(\hat{\theta}_0, \hat{\theta}_1)$.

```
# Mean and Sd for Theta_0_hat
mean(initial$theta_0_hat)
```

```
## [1] 1.547135
```

```
sd(initial$theta_0_hat)
```

```
## [1] 0.2832148
```

```
# Mean and Sd for Theta_1_hat
mean(initial$theta_1_hat)
```

```
## [1] 1.897423
```

```r
sd(initial$theta_1_hat)
```

```
## [1] 0.05071434
```

I got mean and standard deviation for estimates. estimates for θ_1, which is the slope, are more accurate and have less variations than those of θ_0, which is intercept.

Part b. What if you were to change the stopping rule to include our knowledge of the true value? Is this a good way to run this algorithm? What is a potential problem?

I think it is inappropriate. The potential problem is we do not know the true value of parameters. Therefore we cannot use it in the stopping criteria of the algorithm. Of course, if we know true values of parameters and use it in the stopping criteria, it will enhance the result.

Part c. What are your thoughts on this algorithm?

I think this algorithm is good but we should choose initial values, step size, and tolerance carefully.

Problem 3: Inverting matrices

Ok, so John Cook makes some good points, but if you want to do:

$$\hat{\beta} = (X'X)^{-1}X'\underline{y}$$

what are you to do?? Can you explain what is going on?

I will use solve function on $X'X$ to get $(X'X)^{-1}$. I will also use %*% to do multiplication between matrices. The r code is like below.

```r
solve(t(X)%*%X)%*%t(X)%*%y
```

Problem 4: Need for speed challenge

In this problem, we are looking to compute the following:

$$y = p + AB^{-1}(q - r) \tag{1}$$

Where A, B, p, q and r are formed by:

```r
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. How large (bytes) are A and B? Without any optimization tricks, how long does the it take to calculate y?

```
# Size of the matrices
object.size(A)
```

```
## 112347224 bytes
```

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

```
## 1816357208 bytes
```

Size of A is 112347224 bytes and size of B is 1816357208 bytes. Time is calculated like this.

Part b. How would you break apart this compute, i.e., what order of operations would make sense? Are there any mathmatical simplifications you can make? Is there anything about the vectors or matrices we might take advantage of?

Since B is positive definite, We can use Cholesky decomposition and Cholesky inverse to substitute solve function. We can also use qr inverse function. It is known that Cholesky decomposition and cholesky inverse is faster than solve function in high dimensional matrices.

Part c. Use ANY means (ANY package, ANY trick, etc) necessary to compute the above, fast. Wrap your code in "system.time({})", everything you do past assignment "C <- NULL".

```
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

# Calculation time
system.time_solve <- system.time(y <- p+A%*%solve(B)%*%(q-r))
system.time_qr <- system.time(y <- p+A%*%qr.solve(B)%*%(q-r))
system.time_chol <- system.time(y_new <- p+A%*%chol2inv(chol(B))%*%(q-r))

save.image(file="Homework_4_inverse.RData")
```

Since the computation time is so long, I implemented above code in my R (It took many hours) and saved the result in Rdata file. I bring the Rdata file in the RMarkdown.

```r
setwd("~/")
load("~/Homework_4_inverse.RData")

# Time Comparison between methods
system.time_solve
```

```
##    user  system elapsed
## 1228.99    1.08 1234.84
```

```r
system.time_qr
```

```
##    user  system elapsed
## 4346.37   18.44 4373.64
```

```r
system.time_chol
```

```
##    user  system elapsed
## 1379.24    0.91 1380.84
```

We can see that solve function is faster than cholsky inverse or qr inverse function. I think solve function is the best.

Problem 5

a. Create a function that computes the proportion of successes in a vector. Use good programming practices.

```r
compute_success_prob <- function(x){
  success_prob <- mean(x) # mean of binary data is proportion of success
  return(success_prob)
}
```

b. Create a matrix to simulate 10 flips of a coin with varying degrees of "fairness" (columns = probability) as follows:

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

c. Use your function in conjunction with apply to compute the proportion of success in P4b_data by column and then by row. What do you observe? What is going on?

```r
# Success by column
apply(P4b_data, 2, compute_success_prob)
```

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

```
# Success by row
apply(P4b_data, 1, compute_success_prob)
```

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

I found that row-wise proportion is 1 or 0 and column-wise plot is 0.6. The original data duplicates one column in every other columns.

d. You are to fix the above matrix by creating a function whose input is a probability and output is a vector whose elements are the outcomes of 10 flips of a coin. Now create a vector of the desired probabilities. Using the appropriate apply family function, create the matrix we really wanted above. Prove this has worked by using the function created in part a to compute and tabulate the appropriate marginal successes.

```
binom_random <- function(given_prob){
  rbinom(10, 1, prob = given_prob)
}

P4b_data_new <- sapply(30:40/100,binom_random)
P4b_data_new
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
##  [1,]    0    0    1    1    1    1    1    1    1     0     1
##  [2,]    0    0    0    0    1    0    0    0    1     1     0
##  [3,]    1    1    0    1    0    1    0    0    0     1     1
##  [4,]    0    1    1    1    0    1    0    0    0     1     0
##  [5,]    0    0    0    0    1    1    0    0    1     0     1
##  [6,]    0    0    0    0    0    0    0    0    0     1     1
##  [7,]    0    1    1    0    1    1    1    1    1     1     1
##  [8,]    0    0    1    0    0    0    1    0    1     1     0
##  [9,]    0    0    0    0    0    0    0    1    0     0     0
## [10,]    1    0    0    0    0    1    0    0    0     0     0
```

I made an appropriate matrix.

```
# Success by column
apply(P4b_data_new, 2, compute_success_prob)
```

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

We can see that proportion of columns are similar to (0.30,0.31,0.32,0.33,0.34,0.35,0.36,0.37,0.38,0.39,0.40) in order, which is the parameter for each column data.

```
# Success by row
apply(P4b_data_new, 1, compute_success_prob)
```

```
##  [1] 0.72727273 0.27272727 0.54545455 0.45454545 0.36363636 0.18181818
##  [7] 0.81818182 0.36363636 0.09090909 0.18181818
```

We can see that proportion of rows have no pattern, which is appropriate.

Problem 6

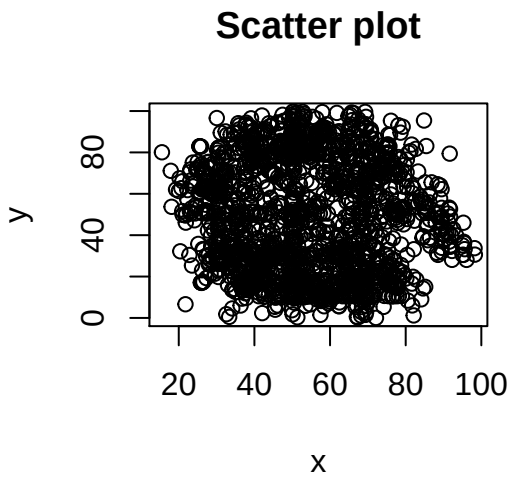
In Homework 3, we had a dataset we were to compute some summary statistics from. The description of the data was given as "a dataset which has multiple repeated measurements from two devices by thirteen Observers". Where the device measurements were in columns "dev1" and "dev2". Reimport that dataset, change the names of "dev1" and "dev2" to x and y and do the following:

1. create a function that accepts a dataframe of values, title, and x/y labels and creates a scatter plot
2. use this function to create:
   (a) a single scatter plot of the entire dataset
   (b) a seperate scatter plot for each observer (using the apply function)

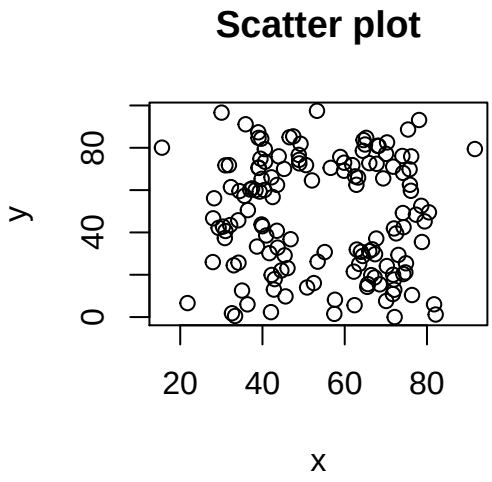
```
# Import data
setwd("C:/Users/User/Downloads")
hw3_data <- readRDS("HW3_data.rds")
names(hw3_data)=c("Observer","x","y")
head(hw3_data)
```

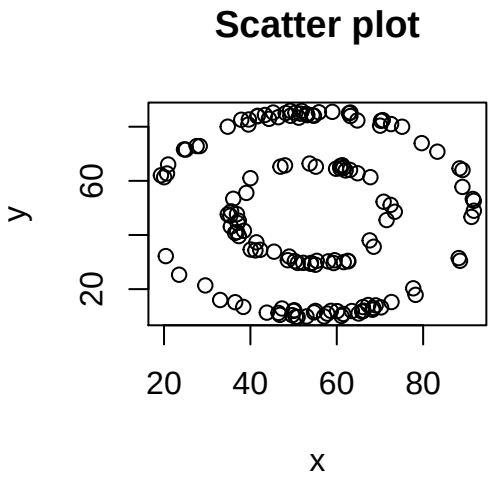
```
##   Observer       x       y
## 1        4 55.3846 97.1795
## 2        4 51.5385 96.0256
## 3        4 46.1538 94.4872
## 4        4 42.8205 91.4103
## 5        4 40.7692 88.3333
## 6        4 38.7179 84.8718
```

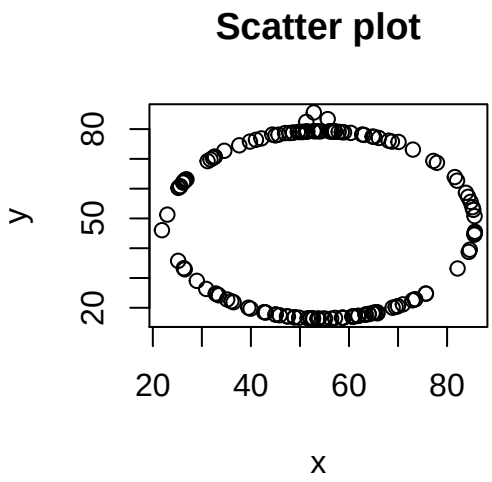
```
# Function
plot_HW3_data <- function(Data){
  plot(Data[,2],Data[,3], xlab="x", ylab="y", main="Scatter plot")
}

# Plot all data
plot_HW3_data(hw3_data)
```

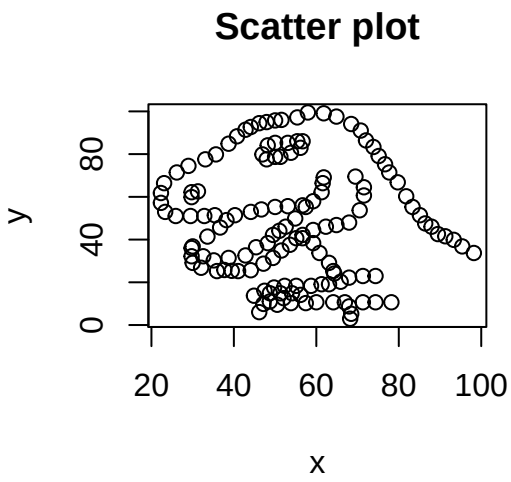


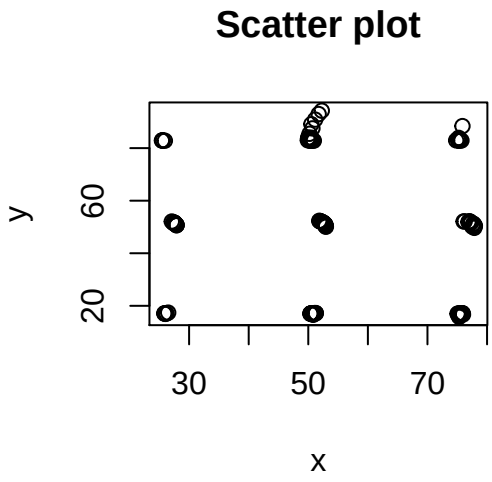
```r
# Plot by observers
by(hw3_data,hw3_data[,1],plot_HW3_data)
```

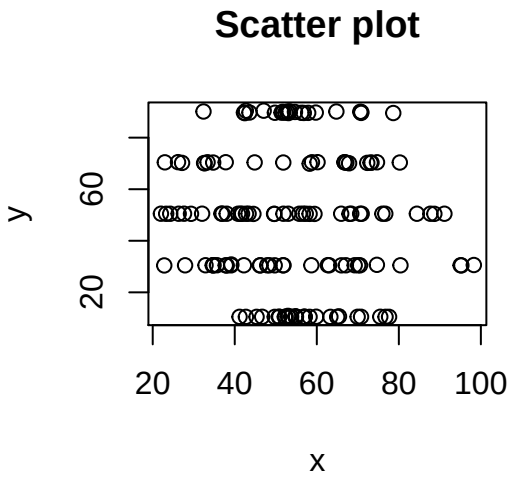


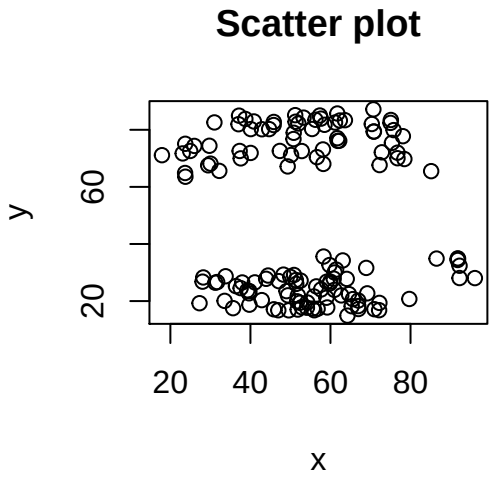


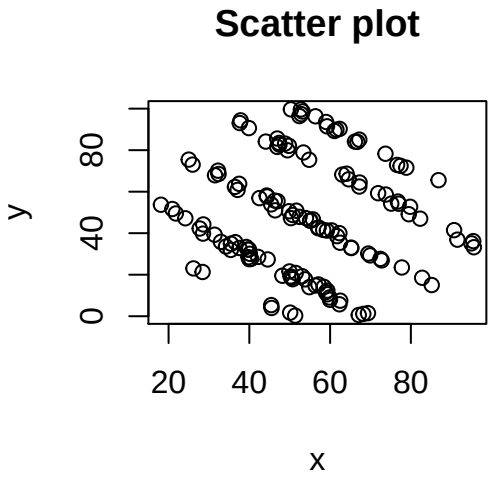


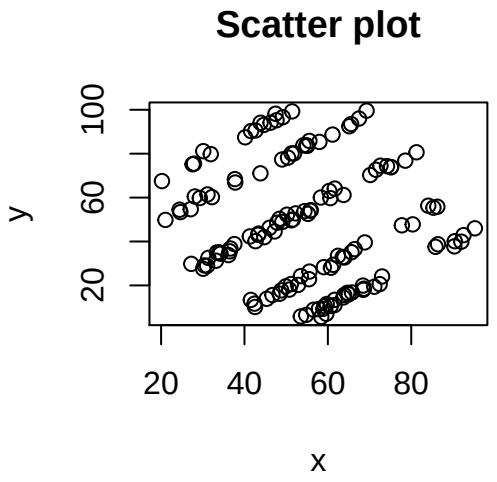


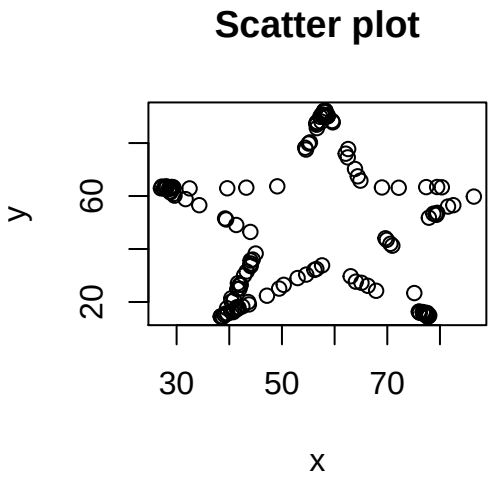

Scatter plot
y
20
60
30
50
70
x

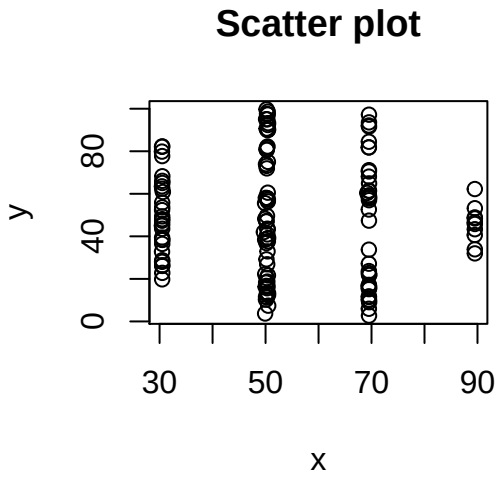

Scatter plot
y
20
60
20
40
60
80
100
x

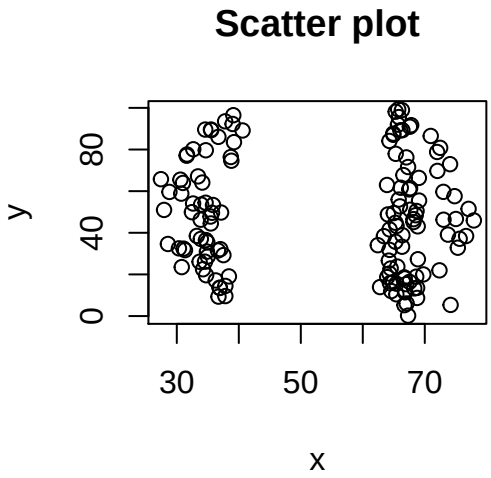

Scatter plot
y
20
60
20
40
60
80
x

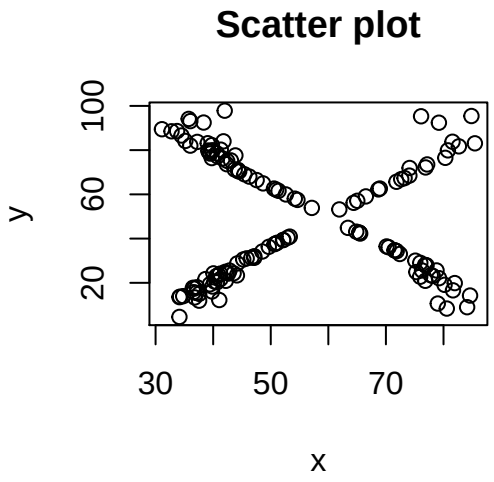

Scatter plot
y
0
40
80
20
40
60
80
x


Scatter plot
y
x
20
40
60
80
20
60
100


Scatter plot
y
x
30
50
70
20
60


Scatter plot
y
x
30
50
70
90
0
40
80


Scatter plot
y
x
30
50
70
0
40
80




```
## hw3_data[, 1]: 1
## NULL
## ------------------------------------------------------------
## hw3_data[, 1]: 2
## NULL
## ------------------------------------------------------------
## hw3_data[, 1]: 3
## NULL
## ------------------------------------------------------------
## hw3_data[, 1]: 4
## NULL
## ------------------------------------------------------------
## hw3_data[, 1]: 5
## NULL
## ------------------------------------------------------------
## hw3_data[, 1]: 6
## NULL
## ------------------------------------------------------------
## hw3_data[, 1]: 7
## NULL
## ------------------------------------------------------------
## hw3_data[, 1]: 8
## NULL
## ------------------------------------------------------------
## hw3_data[, 1]: 9
## NULL
## ------------------------------------------------------------
## hw3_data[, 1]: 10
## NULL
## ------------------------------------------------------------
## hw3_data[, 1]: 11
## NULL
## ------------------------------------------------------------
## hw3_data[, 1]: 12
## NULL
```

```
## ------------------------------------------------------------
## hw3_data[, 1]: 13
## NULL
```

Problem 7

Our ultimate goal in this problem is to create an annotated map of the US. I am giving you the code to create said map, you will need to customize it to include the annotations.

Part a. Get and import a database of US cities and states. Here is some R code to help:

```
#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_state
#unzip("us_cities_states.zip")

#read in data, looks like sql dump, blah
library(data.table)
states <- fread(input = "./us_cities_and_states/cities_extended.sql",skip = 23,sep = "'", sep2 = ",", he
states_2 <- fread(input = "./us_cities_and_states/states.sql",skip = 23,sep = "'", sep2 = ",", header =
states_2$V2 <- tolower(states_2$V2)

### 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?
head(states)
```

```
##            V2 V4
## 1: Holtsville NY
## 2: Holtsville NY
## 3:   Adjuntas PR
## 4:     Aguada PR
## 5:  Aguadilla PR
## 6:  Aguadilla PR
```

```
head(states_2)
```

```
##            V2 V4
## 1:     alaska AK
## 2:    alabama AL
## 3:   arkansas AR
## 4:    arizona AZ
## 5: california CA
## 6:   colorado CO
```

Imported the data.

Part b. Create a summary table of the number of cities included by state.

```
# Method 1
summary(as.factor(states$V4))
```

```
##   AK   AL   AR   AZ   CA   CO   CT   DC   DE   FL   GA   HI   IA   ID   IL   IN
##  273  838  709  532 2651  659  438  284   98 1487  972  139 1060  325 1587  989
##   KS   KY   LA   MA   MD   ME   MI   MN   MO   MS   MT   NC   ND   NE   NH   NJ
##  756  961  725  703  619  489 1170 1031 1170  533  405 1090  407  620  284  733
##   NM   NV   NY   OH   OK   OR   PA   PR   RI   SC   SD   TN   TX   UT   VA   VT
##  426  253 2207 1446  774  484 2208  176   91  539  394  795 2650  344 1238  309
##   WA   WI   WV   WY
##  732  898  859  195
```

```
# Method 2
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:data.table':
##
##     between, first, last

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
states %>%
  group_by(V4) %>%
  tally()
```

```
## # A tibble: 52 x 2
##    V4        n
##    <chr> <int>
##  1 AK      273
##  2 AL      838
##  3 AR      709
##  4 AZ      532
##  5 CA     2651
##  6 CO      659
##  7 CT      438
##  8 DC      284
##  9 DE       98
## 10 FL     1487
## # ... with 42 more rows
```

I think both methods are useful.

Part c. Create a function that counts the number of occurances of a letter in a string. The input to the function should be "letter" and "state_name". The output should be a scalar with the count for that letter. Create a for loop to loop through the state names imported in part a. Inside the for loop, use an apply family function to iterate across a vector of letters and collect the occurance count as a vector.

```
library(stringr)

# Letter finding function
letter_finder <- function(state,letter){
  sum(str_count(states_2[states_2$V4==state]$V2,letter))
}
```

I made a function.

```
# Table
counted_table <- as.data.frame(matrix(NA,length(summary(as.factor(states$V4))),26))
row.names(counted_table) <- names(summary(as.factor(states$V4)))
names(counted_table) <- c("a","b","c","d","e","f","g","h","i","j","k","l","m","n","o","p","q","r","s","t

# Loop
for (i in 1:nrow(counted_table)){
  counted_table[i,] <- mapply(letter_finder,row.names(counted_table)[i],names(counted_table))
}
counted_table
```

```
##    a b c d e f g h i j k l m n o p q r s t u v w x y z
## AK 3 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0
## AL 4 1 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
## AR 3 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 1 2 0 0 0 0 0 0 0
## AZ 2 0 0 0 0 0 0 0 1 0 0 0 0 1 1 0 0 1 0 0 0 0 0 0 0 1
## CA 2 0 1 0 0 1 0 0 2 0 0 1 0 1 1 0 0 1 0 0 0 0 0 0 0 0
## CO 1 0 1 1 0 0 0 0 0 0 0 0 1 0 0 3 0 0 1 0 0 0 0 0 0 0
## CT 0 0 3 0 1 0 0 0 1 0 0 0 0 2 1 0 0 0 0 2 1 0 0 0 0 0
## DC 1 1 2 1 0 1 0 0 3 0 0 1 1 0 2 0 0 1 1 2 1 0 0 0 0 0
## DE 2 0 0 1 2 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1 0 0
## FL 1 0 0 1 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0 0 0
## GA 1 0 0 0 1 0 2 0 1 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0
## HI 2 0 0 0 0 0 0 1 2 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
## IA 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0
## ID 1 0 0 1 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
## IL 0 0 0 0 0 0 0 0 3 0 0 2 0 1 1 0 0 0 1 0 0 0 0 0 0 0
## IN 2 0 0 1 0 0 0 0 2 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0
## KS 2 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 2 0 0 0 0 0 0
## KY 0 0 1 0 1 0 0 0 0 0 2 0 0 1 0 0 0 0 0 0 1 1 0 0 0 1 0
## LA 2 0 0 0 0 0 0 0 0 2 0 0 1 0 1 1 0 0 0 1 0 1 0 0 0 0 0
## MA 2 0 1 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 4 2 1 0 0 0 0
## MD 2 0 0 1 0 0 0 0 0 0 0 0 1 1 1 0 0 0 1 0 0 0 0 0 0 1 0
## ME 1 0 0 0 1 0 0 0 0 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0
## MI 1 0 1 0 0 0 1 1 2 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
## MN 1 0 0 0 1 0 0 0 0 1 0 0 0 0 1 2 1 0 0 0 0 1 1 0 0 0 0
## MO 0 0 0 0 0 0 0 0 0 2 0 0 0 0 1 0 1 0 0 0 1 2 0 1 0 0 0
```

```
## MS 0 0 0 0 0 0 0 0 4 0 0 0 1 0 0 2 0 0 4 0 0 0 0 0 0 0
## MT 2 0 0 0 0 0 0 0 0 0 0 0 0 1 2 1 0 0 0 0 1 0 0 0 0 0 0
## NC 2 0 1 0 0 0 0 1 1 0 0 1 0 2 2 0 0 2 0 1 0 0 0 0 0 0
## ND 2 0 0 1 0 0 0 1 0 0 1 0 0 1 2 0 0 1 0 2 0 0 0 0 0 0
## NE 2 1 0 0 1 0 0 0 0 0 1 0 0 1 0 0 0 1 1 0 0 0 0 0 0 0
## NH 1 0 0 0 2 0 0 2 1 0 0 0 1 1 0 1 0 1 1 0 0 0 1 0 0 0
## NJ 0 0 0 0 0 3 0 0 0 0 0 1 0 0 0 0 1 0 0 0 1 1 0 0 0 1 0 1 0
## NM 0 0 1 0 2 0 0 0 0 1 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 1 1 0 0
## NV 2 0 0 1 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0
## NY 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 1 1 0 0 1 0 0 0 0 0 1 0 1 0
## OH 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0
## OK 2 0 0 0 0 0 0 0 1 0 0 1 1 1 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0
## OR 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 1 2 0 0 1 0 0 0 0 0 0 0 0 0
## PA 2 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 3 0 1 0 0 1 0 0 0 1 0 0 1 0
## PR 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## RI 1 0 0 2 1 0 0 0 1 1 0 0 0 1 0 1 1 0 0 0 1 1 0 0 0 0 0 0 0 0
## SC 2 0 1 0 0 0 0 0 1 1 0 0 0 1 0 1 2 0 0 0 1 1 1 1 0 0 0 0 0 0
## SD 2 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0 0 2 0 0 0 0 1 2 1 0 0 0 0 0
## TN 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 2 1 0 0 0 0 0 0 0
## TX 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 0 0
## UT 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0
## VA 1 0 0 0 0 0 0 1 0 3 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0
## VT 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 1 1 0 0 1 0 1 0 1 0 0 0 0 0
## WA 1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 2 1 0 0 0 0 1 1 0 0 1 0 0 0
## WI 0 0 1 0 0 0 0 0 0 2 0 0 0 0 0 2 1 0 0 0 0 2 0 0 0 1 0 0 0
## WV 1 0 0 0 0 1 0 1 0 3 0 0 0 0 0 1 0 0 0 0 1 1 1 0 1 1 0 0 0
## WY 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 1 0 1 0
```

I found letters by states easily by mapply and for loop.

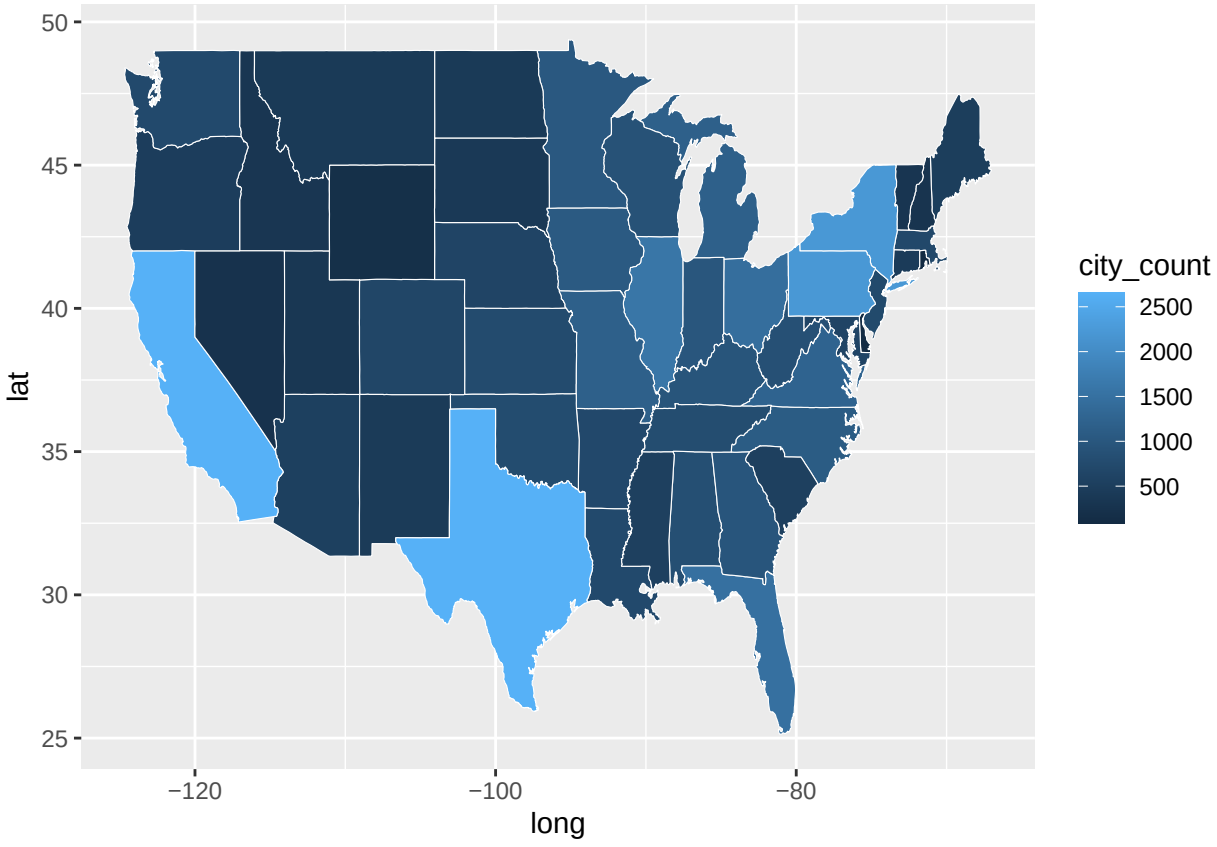
Part d. Create 2 maps to finalize this. Map 1 should be colored by count of cities on our list within the state. Map 2 should highlight only those states that have more than 3 occurances of ANY letter in thier name.

```
library(ggplot2)
library(maps)
library(dplyr)

State_Abb <- states_2
names(State_Abb) <- c("region", "abb")

MainStates <- map_data("state")
MainStates <- left_join(MainStates,State_Abb,by='region')

city_count <- as.data.frame(summary(as.factor(states$V4)))
city_count$region <- rownames(city_count)
names(city_count) <- c("city_count","abb")

a_count <- data.frame(counted_table[,1],rownames(counted_table))
names(a_count) <- c("a_count","abb")

MainStates <- left_join(MainStates,city_count,by='abb')
```

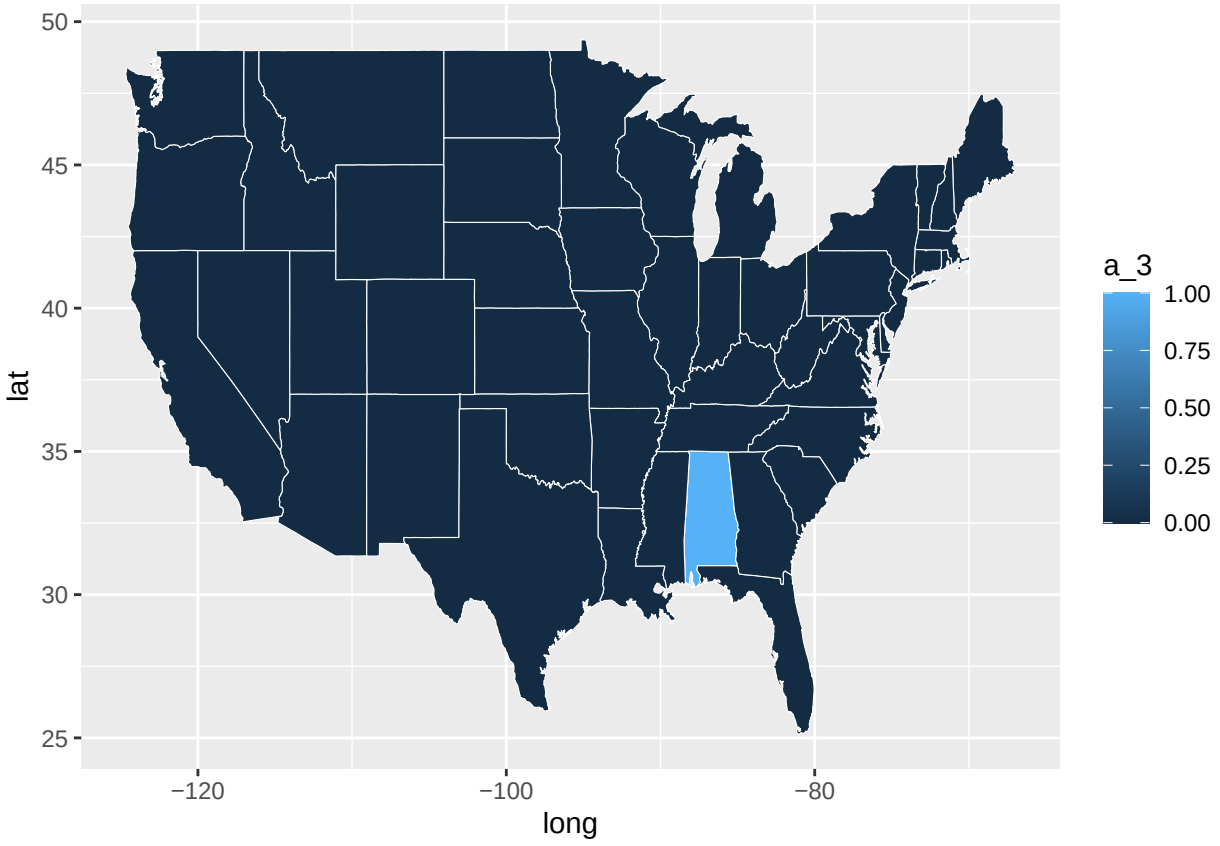
```
MainStates <- left_join(MainStates,a_count,by='abb')
MainStates$a_3 <- as.numeric(MainStates$a_count>3)
head(MainStates)
```

```
##        long      lat group order  region subregion abb city_count a_count a_3
## 1 -87.46201 30.38968     1     1 alabama      <NA>  AL        838       4   1
## 2 -87.48493 30.37249     1     2 alabama      <NA>  AL        838       4   1
## 3 -87.52503 30.37249     1     3 alabama      <NA>  AL        838       4   1
## 4 -87.53076 30.33239     1     4 alabama      <NA>  AL        838       4   1
## 5 -87.57087 30.32665     1     5 alabama      <NA>  AL        838       4   1
## 6 -87.58806 30.32665     1     6 alabama      <NA>  AL        838       4   1
```

```
# Map 1 : Plot by city count
ggplot() + geom_polygon( data=MainStates,
                         aes(x=long, y=lat, group=group, fill = city_count),
                         color="white", size = 0.2)
```




```
# Map 2 : Plot highlighted states with letter "a" appears more than 3 times in its name
ggplot() + geom_polygon( data=MainStates,
                         aes(x=long, y=lat, group=group, fill = a_3),
                         color="white", size = 0.2)
```



Problem 8

Bootstrapping

Recall the sensory data from five operators:
http://www2.isye.gatech.edu/~jeffwu/wuhamadabook/data/Sensory.dat

Sometimes, you really want more data to do the desired analysis, but going back to the "field" is often not an option. An often used method is bootstrapping. Check out the second answer here for a really nice and detailed description of bootstrapping: https://stats.stackexchange.com/questions/316483/manually-bootstrapping-linear-regression-in-r.

What we want to do is bootstrap the Sensory data to get non-parametric estimates of the parameters. Assume that we can neglect item in the analysis such that we are really only interested in a linear model lm(y~operator).

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? If you want to duplicate the code to test it, use the quantreg package to get the data.

```
library(quantmod)
```

```
## Loading required package: xts
```

```
## Loading required package: zoo

##
## Attaching package: 'zoo'

## The following objects are masked from 'package:base':
##
##     as.Date, as.Date.numeric

##
## Attaching package: 'xts'

## The following objects are masked from 'package:dplyr':
##
##     first, last

## The following objects are masked from 'package:data.table':
##
##     first, last

## Loading required package: TTR

## Registered S3 method overwritten by 'quantmod':
##   method            from
##   as.zoo.data.frame zoo

## Version 0.4-0 included new data defaults. See ?getSymbols.
```

```
#1)fetch data from Yahoo
#AAPL prices
apple08 <- getSymbols('AAPL', auto.assign = FALSE, from = '2008-1-1', to =
                        "2008-12-31")[,6]
```

```
## 'getSymbols' currently uses auto.assign=TRUE by default, but will
## use auto.assign=FALSE in 0.5-0. You will still be able to use
## 'loadSymbols' to automatically load data. getOption("getSymbols.env")
## and getOption("getSymbols.auto.assign") will still be checked for
## alternate defaults.
##
## This message is shown once per session and may be disabled by setting
## options("getSymbols.warning4.0"=FALSE). See ?getSymbols for details.
```

```
#market proxy
rm08<-getSymbols('^ixic', auto.assign = FALSE, from = '2008-1-1', to =
                   "2008-12-31")[,6]

#log returns of AAPL and market
logapple08<- na.omit(ROC(apple08)*100)
logrm08<-na.omit(ROC(rm08)*100)

#OLS for beta estimation
```

```
beta_AAPL_08<-summary(lm(logapple08~logrm08))$coefficients[2,1]
```

```
#create df from AAPL returns and market returns
df08<-cbind(logapple08,logrm08)
set.seed(777)
Boot_times=1000
sd.boot=rep(0,Boot_times)
for(i in 1:Boot_times){
  # nonparametric bootstrap
  bootdata=df08[sample(nrow(df08), size = 251, replace = TRUE),]
  names(bootdata)=c("logapple08","logrm08")
  sd.boot[i]= coef(summary(lm(logapple08~logrm08, data = bootdata)))[2,2]
}

#after fix
sd.boot[1:10]
```

```
##  [1] 0.07697246 0.05463012 0.05986866 0.05464304 0.06391270 0.06532540
##  [7] 0.06322353 0.06480742 0.06139695 0.06432005
```

I inserted $names(bootdata) = c(\text{"}logapple08\text{"}, \text{"}logrm08\text{"})$ in the for loop. The problem with this code is column names for bootdata were not "logapple08" and "logrm08". So sd.boot was calculated with same original logapple08 and logrm08 variables over and over. After my fix, We can see that bootstrap works fine. No same results for bootstrap samples.

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 operators, ie each sample draws for each operator.

```
library(tidyr)

## Sensory Import
url_Sensory <- "http://www2.isye.gatech.edu/~jeffwu/wuhamadabook/data/Sensory.dat"
Sensory_raw <- fread(url_Sensory, fill=TRUE, data.table = FALSE, skip=1)
knitr::kable(head(Sensory_raw))
```

Item	1	2	3	4	5
1.0	4.3	4.9	3.3	5.3	4.4
4.3	4.5	4.0	5.5	3.3	NA
4.1	5.3	3.4	5.7	4.7	NA
2.0	6.0	5.3	4.5	5.9	4.7
4.9	6.3	4.2	5.5	4.9	NA
6.0	5.9	4.7	6.3	4.6	NA

```
## Sensory data : tidyverse : dividing data
Sensory_new <- cbind(1:length(Sensory_raw[,6]),Sensory_raw)
colnames(Sensory_new) <- c("number_key","trash",paste("Operator", 1:5))

Sensory_A <- Sensory_new %>%
  subset(is.na(Sensory_new[,7]))

Sensory_B <- Sensory_new %>%
  subset(Sensory_new[,7]>0)

Sensory_A <- Sensory_A[,-7]
Sensory_B <- Sensory_B[,-2]

## Sensory data : tidyverse : combining data
colnames(Sensory_A) <- colnames(Sensory_B)
Sensory_new <- rbind(Sensory_A,Sensory_B)
Sensory_new <- Sensory_new %>%
  arrange(number_key) %>%
  select(-1)
knitr::kable(head(Sensory_new))
```

Operator 1	Operator 2	Operator 3	Operator 4	Operator 5
4.3	4.9	3.3	5.3	4.4
4.3	4.5	4.0	5.5	3.3
4.1	5.3	3.4	5.7	4.7
6.0	5.3	4.5	5.9	4.7
4.9	6.3	4.2	5.5	4.9
6.0	5.9	4.7	6.3	4.6

```
colnames(Sensory_new)=1:5

Sensory_tidy <- gather(data=Sensory_new,key="Operator",value="value",)
head(Sensory_tidy)
```

```
##   Operator value
## 1        1   4.3
## 2        1   4.3
## 3        1   4.1
## 4        1   6.0
## 5        1   4.9
## 6        1   6.0
```

I imported the data and refined it.

```
# Bootstrap Regression

set.seed(777)

Sensory_tidy_1 <- Sensory_tidy[Sensory_tidy$Operator=="1",]
Sensory_tidy_2 <- Sensory_tidy[Sensory_tidy$Operator=="2",]
Sensory_tidy_3 <- Sensory_tidy[Sensory_tidy$Operator=="3",]
```

```r
Sensory_tidy_4 <- Sensory_tidy[Sensory_tidy$Operator=="4",]
Sensory_tidy_5 <- Sensory_tidy[Sensory_tidy$Operator=="5",]


bootstrap_estimate <- data.frame(matrix(NA,100,4))
colnames(bootstrap_estimate) <- c("2 vs 1", "3 vs 1", "4 vs 1", "5 vs 1")

# System Time for Bootstrap
system.time(
  for (i in 1:100){
    bootsample <- rbind(Sensory_tidy_1[sample(nrow(Sensory_tidy_1),nrow(Sensory_tidy_1),replace=TRUE),]
                        Sensory_tidy_2[sample(nrow(Sensory_tidy_2),nrow(Sensory_tidy_2),replace=TRUE),]
                        Sensory_tidy_3[sample(nrow(Sensory_tidy_3),nrow(Sensory_tidy_3),replace=TRUE),]
                        Sensory_tidy_4[sample(nrow(Sensory_tidy_4),nrow(Sensory_tidy_4),replace=TRUE),]
                        Sensory_tidy_5[sample(nrow(Sensory_tidy_5),nrow(Sensory_tidy_5),replace=TRUE),])
    bootstrap_estimate[i,] <- summary(lm(value~Operator,data=bootsample))$coefficients[2:5,1]
  }
)
```

```
##    user  system elapsed
##    0.22    0.00    0.22
```

```r
head(bootstrap_estimate)
```

```
##       2 vs 1     3 vs 1     4 vs 1     5 vs 1
## 1  0.2200000 -0.4500000  0.8333333 -0.9166667
## 2  1.3700000  0.3000000  0.9333333 -0.4733333
## 3  0.8566667 -0.6033333  1.2033333 -0.2533333
## 4 -0.1433333 -0.9133333  0.1133333  0.1833333
## 5 -0.8866667 -1.0366667 -0.0800000 -1.4766667
## 6  0.5966667 -0.6866667  0.9066667  0.0400000
```

```r
# Regression Estimate
summary(lm(value~Operator,data=Sensory_tidy))$coefficients[2:5,1]
```

```
##  Operator2  Operator3  Operator4  Operator5
##  0.4700000 -0.4266667  0.6000000 -0.3266667
```

```r
# Bootstrap Regression estimate
apply(bootstrap_estimate,2,mean)
```

```
##     2 vs 1     3 vs 1     4 vs 1     5 vs 1
##  0.5273333 -0.2695000  0.7875000 -0.2602667
```

Bootstrap estimate and original estimate are quite similar.

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.

Create a single table summarizing the results and timing from part a and b. What are your thoughts?

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

```
## Loading required package: iterators
```

```
## Loading required package: parallel
```

```r
boots_reg=function(){
  bootsample <- rbind(Sensory_tidy_1[sample(nrow(Sensory_tidy_1),nrow(Sensory_tidy_1),replace=TRUE),],
                      Sensory_tidy_2[sample(nrow(Sensory_tidy_2),nrow(Sensory_tidy_2),replace=TRUE),],
                      Sensory_tidy_3[sample(nrow(Sensory_tidy_3),nrow(Sensory_tidy_3),replace=TRUE),],
                      Sensory_tidy_4[sample(nrow(Sensory_tidy_4),nrow(Sensory_tidy_4),replace=TRUE),],
                      Sensory_tidy_5[sample(nrow(Sensory_tidy_5),nrow(Sensory_tidy_5),replace=TRUE),])
  return(summary(lm(value~Operator,data=bootsample))$coefficients[2:5,1])
}

boots_reg()
```

```
##   Operator2   Operator3   Operator4   Operator5
##  0.02333333 -0.45333333  1.16333333 -0.26000000
```

```r
cl <- makeCluster(8)
registerDoParallel(cl)

# System Time for Bootstrap (foreach)
system.time(
  bootstrap_foreach_estimate <- foreach(i = 1:100, .combine= 'rbind') %dopar% {
    boots_reg()
  }
)
```

```
##    user  system elapsed
##    0.11    0.00    0.25
```

```r
colnames(bootstrap_foreach_estimate) <- c("2 vs 1", "3 vs 1", "4 vs 1", "5 vs 1")

stopCluster(cl)

head(bootstrap_foreach_estimate)
```

```
##               2 vs 1     3 vs 1     4 vs 1      5 vs 1
## result.1  0.11666667 -0.8600000 -0.2866667 -1.21666667
## result.2  0.73000000 -0.5700000  0.5000000 -0.74000000
## result.3 -0.12666667 -1.3200000  1.0200000 -0.55666667
## result.4  1.06666667 -0.1533333  1.5900000  0.03666667
## result.5  1.05000000  0.4133333  1.1566667 -0.45000000
## result.6  0.01333333 -0.3200000  1.5200000 -0.68333333
```

```r
# Regression Estimate
summary(lm(value~Operator,data=Sensory_tidy))$coefficients[2:5,1]
```

```
##  Operator2  Operator3  Operator4  Operator5
##  0.4700000 -0.4266667  0.6000000 -0.3266667
```

```r
# Bootstrap Regression estimate (foreach)
apply(bootstrap_foreach_estimate,2,mean)
```

```
##     2 vs 1     3 vs 1     4 vs 1     5 vs 1
##  0.4425333 -0.4635667  0.5089000 -0.4065333
```

As bootstrap sampling does not affect each others, we can use foreach, which makes computation faster. We can see that bootstrap by foreach is faster than bootstrap by for loop.

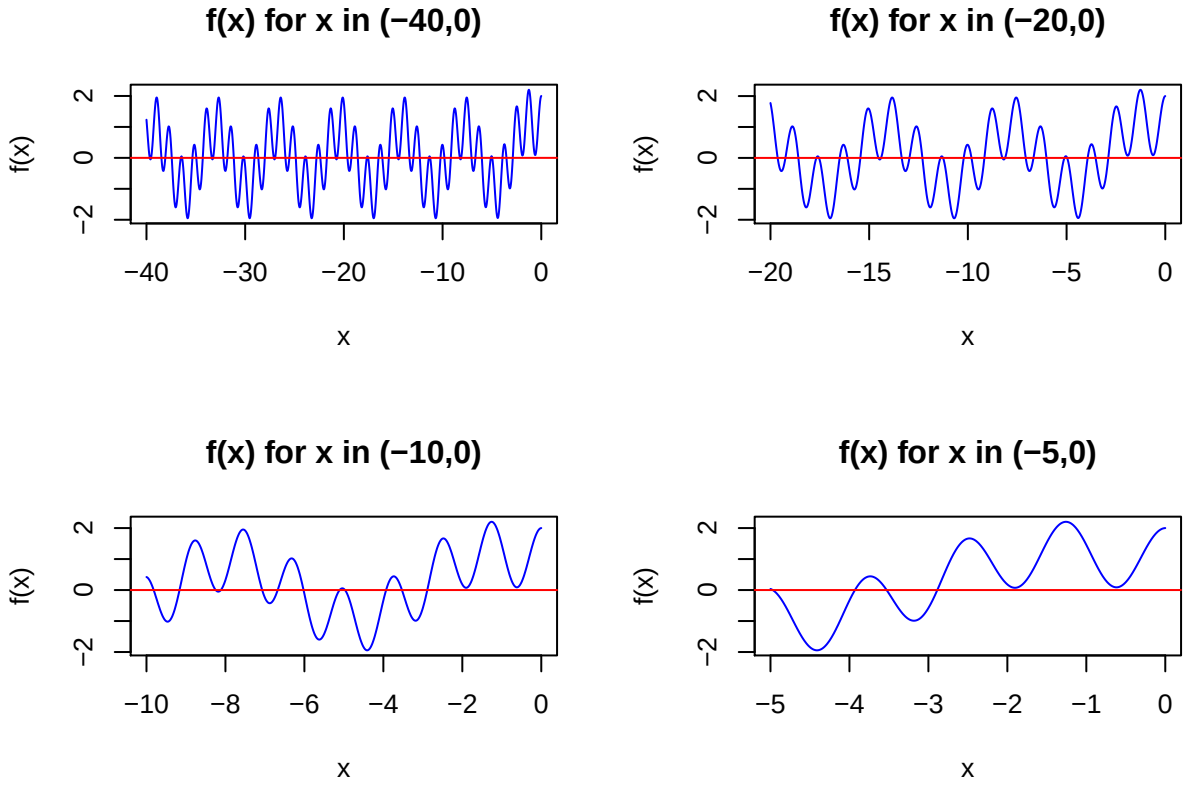
Problem 9

Newton's method gives an answer for a root. To find multiple roots, you need to try different starting values. There is no guarantee for what start will give a specific root, so you simply need to try multiple. From the plot of the function in HW3, problem 8, how many roots are there?

Create a vector (`length.out=1000`) as a "grid" covering all the roots and extending +/-1 to either end.

```r
given_func <- function(x){3^x-sin(x)+cos(5*x)}
given_func_derivative <- function(x){log(3)*3^x-cos(x)-5*sin(5*x)}
```

```r
interval_1 <- seq(-40,0,length=1000)
interval_2 <- seq(-20,0,length=1000)
interval_3 <- seq(-10,0,length=1000)
interval_4 <- seq(-5,0,length=1000)

par(mfrow=c(2,2))
plot(interval_1,given_func(interval_1),type="l", lwd=1, col="blue",main="f(x) for x in (-40,0)", xlab="
abline(h=0, col="red", lwd=1)
plot(interval_2,given_func(interval_2),type="l", lwd=1, col="blue",main="f(x) for x in (-20,0)", xlab="
abline(h=0, col="red", lwd=1)
plot(interval_3,given_func(interval_3),type="l", lwd=1, col="blue",main="f(x) for x in (-10,0)", xlab="
abline(h=0, col="red", lwd=1)
plot(interval_4,given_func(interval_4),type="l", lwd=1, col="blue",main="f(x) for x in (-5,0)", xlab="x
abline(h=0, col="red", lwd=1)
```



There are so many solutions.

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

```
newton_for_given_func <- function(initial,max_iter,tolerance,lower_bound,upper_bound){

  # "initial" is initial value for the algorithm.
  # "max_iter" is maximum value for interation.
  # "tolerance" is minimum tolerance for absolute error in the solution.
  # "lower_bound" and "upper_bound" are lower and upper bound for the values.
  # If initial value is out of the boundary, the algorithm will give null values with warning message.
  # Also, If the solution of the algorithm is out of the boundary, there will be warning message.

  # Warning message and null output for initial value out of the boundary.

  if(initial<lower_bound | initial>upper_bound)
  {warning("Initial value should be included in (lower_bound, upper_bound)")
    return(NA)}

  # Set iteration value and values vector. outputs will be stored in values vector.
  iter <- 1
  values <- numeric(max_iter)
  values[1] <- initial
```

```
  # first iteration should be completed.
  values[2] <- values[1]-given_func(values[1])/given_func_derivative(values[1])
  iter <- 2

  # the other iterations.
  # If the iteration reaches max iteration or absolute error is less than tolerance,
  # the algorithm stops.
  while ( (iter<max_iter) && (abs(values[iter]-values[iter-1])>=tolerance) ){
    values[iter+1] <- values[iter]-given_func(values[iter])/given_func_derivative(values[iter])
    iter <- iter+1
  }

  # Plotting absolute errors for iterations.
  # The algorithm stops when the error reaches the tolerance or iteration reaches the max iteration.
  #plot(2:iter,abs(values[2:iter]-values[1:iter-1]),type="l",lwd=2, col="blue",
       #main="Absolute Error by Iteration", xlab="Iteration", ylab="Absolute Error")
  #abline(h=tolerance,lwd=2,col="red")

  # Algorithm gives below list as a result.
  result <- list(values[iter],iter,abs(values[iter]-values[iter-1]), given_func(values[iter]))
  names(result) <- c("Solution", "The Number of Iterations", "Absolute Error", "f(x)")

  # Warning message for iteration reaches max iteration.
  if(iter==max_iter)
    {warning("The algorithm did not converged during iterations.")}

  # Warning message for the solution outside of the boundary.
  if(values[iter]<lower_bound | values[iter]>upper_bound)
    {warning("The solution is out of the boundary.")}
  return(result)
}

newton_only_solution <- function(initial,max_iter,tolerance,lower_bound,upper_bound){
  result <- newton_for_given_func(initial,max_iter,tolerance,lower_bound,upper_bound)
  return(as.numeric(result[1]))
}

newton_only_solution_one_argument <- function(initial){

  given_func <- function(x){3^x-sin(x)+cos(5*x)}
  given_func_derivative <- function(x){log(3)*3^x-cos(x)-5*sin(5*x)}

  max_iter=1000
  tolerance=0.1^10
  lower_bound=-Inf
  upper_bound=Inf

  # "initial" is initial value for the algorithm.
  # "max_iter" is maximum value for interation.
  # "tolerance" is minimum tolerance for absolute error in the solution.
  # "lower_bound" and "upper_bound" are lower and upper bound for the values.
  # If initial value is out of the boundary, the algorithm will give null values with warning message.
  # Also, If the solution of the algorithm is out of the boundary, there will be warning message.
```

```
  # Warning message and null output for initial value out of the boundary.

  if(initial<lower_bound | initial>upper_bound)
  {warning("Initial value should be included in (lower_bound, upper_bound)")
    return(NA)}

  # Set iteration value and values vector. outputs will be stored in values vector.
  iter <- 1
  values <- numeric(max_iter)
  values[1] <- initial

  # first iteration should be completed.
  values[2] <- values[1]-given_func(values[1])/given_func_derivative(values[1])
  iter <- 2

  # the other iterations.
  # If the iteration reaches max iteration or absolute error is less than tolerance,
  # the algorithm stops.
  while ( (iter<max_iter) && (abs(values[iter]-values[iter-1])>=tolerance) ){
    values[iter+1] <- values[iter]-given_func(values[iter])/given_func_derivative(values[iter])
    iter <- iter+1
  }

  # Plotting absolute errors for iterations.
  # The algorithm stops when the error reaches the tolerance or iteration reaches the max iteration.
  #plot(2:iter,abs(values[2:iter]-values[1:iter-1]),type="l",lwd=2, col="blue",
       #main="Absolute Error by Iteration", xlab="Iteration", ylab="Absolute Error")
  #abline(h=tolerance,lwd=2,col="red")

  # Algorithm gives below list as a result.
  result <- list(values[iter],iter,abs(values[iter]-values[iter-1]), given_func(values[iter]))
  names(result) <- c("Solution", "The Number of Iterations", "Absolute Error", "f(x)")

  # Warning message for iteration reaches max iteration.
  if(iter==max_iter)
  {warning("The algorithm did not converged during iterations.")}

  # Warning message for the solution outside of the boundary.
  if(values[iter]<lower_bound | values[iter]>upper_bound)
  {warning("The solution is out of the boundary.")}

  return(as.numeric(result[1]))
}
```

In last HW, I made function to get solution from $f(x)$ using Newton's Method.

```
# Time to find multiple solutions
time_newton <- system.time(
  solutions <- mapply(newton_only_solution,seq(-3,3,length=20),max_iter=1000,tolerance=0.1^10,lower_bou
)
time_newton
```

```
##    user  system elapsed
```

```
##    0.06    0.00    0.06
```

```
# Initial values and their solutions
result_newton <- data.frame(seq(-3,3,length=20),solutions)
names(result_newton) <- c("initial","solution")
result_newton
```

```
##       initial   solution
## 1  -3.0000000  -2.887058
## 2  -2.6842105  -2.887058
## 3  -2.3684211  -2.887058
## 4  -2.0526316  -2.887058
## 5  -1.7368421 -30.237829
## 6  -1.4210526  -2.887058
## 7  -1.1052632  -4.971508
## 8  -0.7894737  -2.887058
## 9  -0.4736842 -21.729349
## 10 -0.1578947  -2.887058
## 11  0.1578947  -2.887058
## 12  0.4736842  -7.068483
## 13  0.7894737  -2.887058
## 14  1.1052632  -2.887058
## 15  1.4210526  -9.162986
## 16  1.7368421 -13.351769
## 17  2.0526316  -3.528723
## 18  2.3684211 -44.374996
## 19  2.6842105  -2.887058
## 20  3.0000000  -3.528723
```

I used mapply function rather than apply function since there are many arguments in my function.

Part b. Repeat the apply command using the equivelant parApply command. Use 8 workers. `cl <- makeCluster(8)`.

```
library(parallel)
library(doParallel)

cl <- makeCluster(8)
registerDoParallel(cl)

# Time to find multiple solutions in parallel computing
time_newton_par <- system.time(
  solutions_par <- parSapply(cl,seq(-3,3,length=20),newton_only_solution_one_argument)
)

stopCluster(cl)

time_newton_par
```

```
##    user  system elapsed
##    0.00    0.00    0.15
```

```r
# Initial values and their solutions in parallel computing
result_newton_par <- data.frame(seq(-3,3,length=20),solutions_par)
names(result_newton_par) <- c("initial","solution")
result_newton_par
```

```
##       initial   solution
## 1  -3.0000000  -2.887058
## 2  -2.6842105  -2.887058
## 3  -2.3684211  -2.887058
## 4  -2.0526316  -2.887058
## 5  -1.7368421 -30.237829
## 6  -1.4210526  -2.887058
## 7  -1.1052632  -4.971508
## 8  -0.7894737  -2.887058
## 9  -0.4736842 -21.729349
## 10 -0.1578947  -2.887058
## 11  0.1578947  -2.887058
## 12  0.4736842  -7.068483
## 13  0.7894737  -2.887058
## 14  1.1052632  -2.887058
## 15  1.4210526  -9.162986
## 16  1.7368421 -13.351769
## 17  2.0526316  -3.528723
## 18  2.3684211 -44.374996
## 19  2.6842105  -2.887058
## 20  3.0000000  -3.528723
```

I did the same thing using parSapply.

Create a table summarizing the roots and timing from both parts a and b. What are your thoughts?

```r
result_compare <- cbind(result_newton,result_newton_par[,2])
names(result_compare) <- c("initial","solution", "solution (par)")
time_compare <- rbind(time_newton,time_newton_par)
time_compare <- time_compare[,1:3]

library(knitr)

kable(result_compare)
```

initial	solution	solution (par)
-3.0000000	-2.887058	-2.887058
-2.6842105	-2.887058	-2.887058
-2.3684211	-2.887058	-2.887058
-2.0526316	-2.887058	-2.887058
-1.7368421	-30.237829	-30.237829
-1.4210526	-2.887058	-2.887058
-1.1052632	-4.971508	-4.971508
-0.7894737	-2.887058	-2.887058
-0.4736842	-21.729349	-21.729349
-0.1578947	-2.887058	-2.887058
0.1578947	-2.887058	-2.887058

initial	solution	solution (par)
0.4736842	-7.068484	-7.068484
0.7894737	-2.887058	-2.887058
1.1052632	-2.887058	-2.887058
1.4210526	-9.162986	-9.162986
1.7368421	-13.351769	-13.351769
2.0526316	-3.528723	-3.528723
2.3684211	-44.374996	-44.374996
2.6842105	-2.887058	-2.887058
3.0000000	-3.528723	-3.528723

```
kable(time_compare)
```

	user.self	sys.self	elapsed
time_newton	0.06	0	0.06
time_newton_par	0.00	0	0.15

I think computation result is same but b (parallel computing) is faster than a.

Problem 10

Finish this homework by pushing your changes to your repo.

Only submit the .Rmd and .pdf solution files. Names should be formatted HW4_pid.Rmd and HW4_pid.pdf