Lab Assignment 5 - Parallel Computing

Instructions

Please complete the exercises below. Submit your completed assignment as a PDF, HTML or Word document outputted by knitr or compiled manually showing both the code and output. (It should be in a similar format to this document).

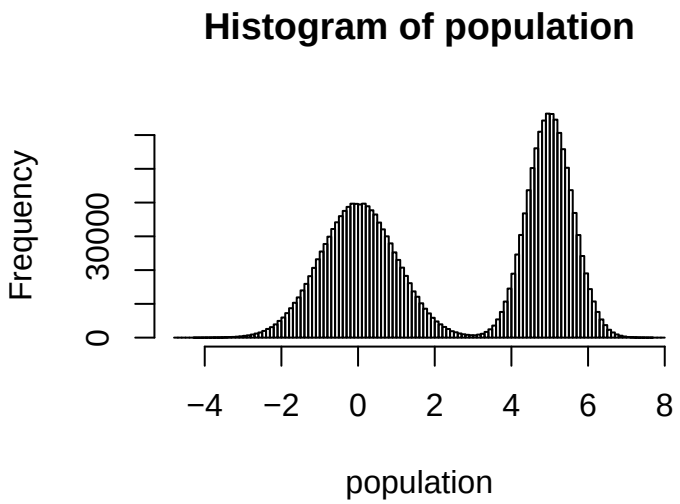
Note that in this document code blocks are shown with a grey background and output from running the code blocks is displayed with ## preceding the output.

There may be some packages used in this assignment that you have not yet installed. In many cases the instructions are to "modify" the code provided, and it is implied in all cases that you should make sure the modified code successfully runs on your computer.

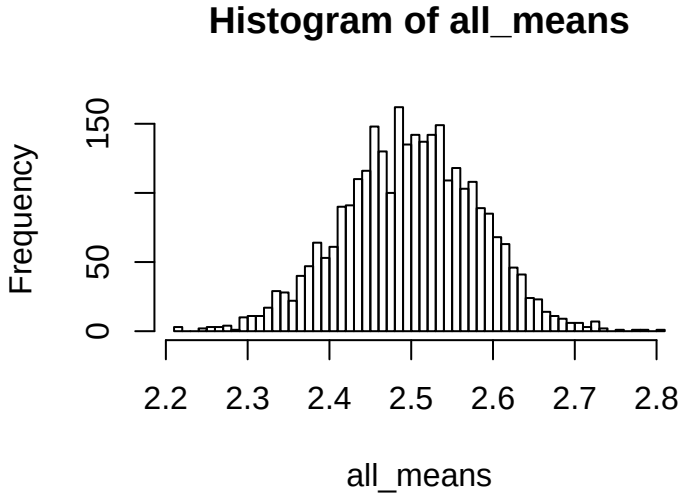
In this lab assignment, you will implement and learn more about parallel computing. We'll start by developing some useful code that we can run in parallel.

The Central Limit Theorem (CLT) tells us that if we sample from a population many times and calculate the mean each time, with a large enough sample size the distribution of means will approximate a normal (Gaussian) distribution. This holds regardless of if the distribution of the population is normal or not. Let's run some simulations to see it in action.

```
#let's create a funky bi-modal population to sample from
population <- c(rnorm(1e6), rnorm(1e6,5,.6))
hist(population, 100)
```

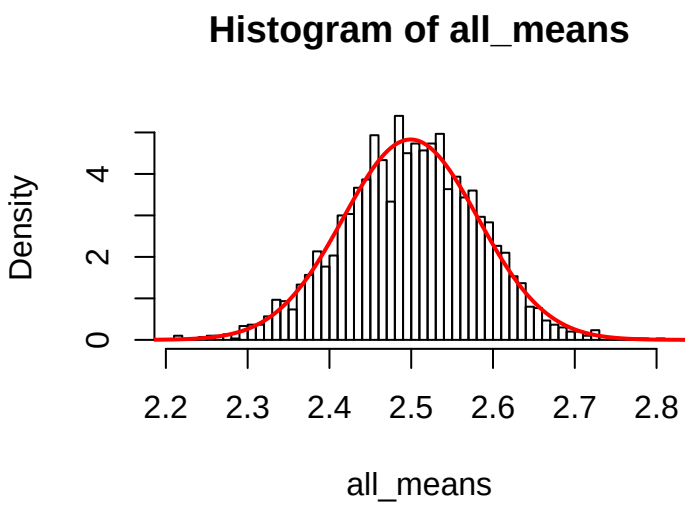


```r
#now we create a function to sample and calculate the mean of each sample
sample_and_calculate_mean <- function()
{
  x <- sample(population, size=1000)#N=1000 from the population
  mean <- mean(x)
  return(mean)
}

#do it 3,000 times
all_means <- replicate(3e3, sample_and_calculate_mean())
hist(all_means, 50)#visualize
```



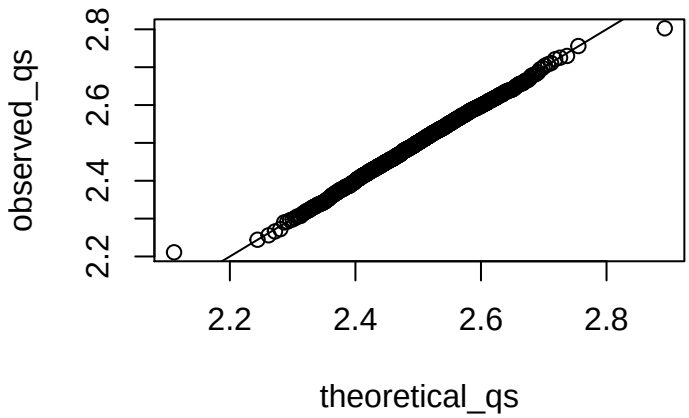
Above we can see the simulated sampling distribution of the mean with sample size N=1,000. It does look normally distributed, but we cannot know for sure with the naked eye. Let's calculate the mean and sd of the sampling distribution (in other words, the distribution of means) and draw a normal distribution with those parameters on top of the histogram to see if it matches up.

```r
mean <- mean(all_means)
standard_error <- sd(all_means)

hist(all_means, 50, freq = F)#visualize histogram

x <- seq(-10, 10,.001)#evenly spaced x values
normal_density <- dnorm(x, mean, standard_error)#dnorm returns density values at each x
lines(x, normal_density, col="red", lwd=2)#add the curve to the plot
```



That does appear to match up well. We can also assess normality with a Q-Q plot. The logic is that the values at each quantile of the distributions should be almost the same if the observed distribution matches the theoretical distribution. For example, the median is the 50% quantile, so the medians in both distributions should be about the same value. We can assess the linear relationship more formally by calculating a correlation coefficient.

```
observed_qs <- quantile(all_means, seq(0,1,.001))#extract quantiles from all_means
theoretical_qs <- quantile(rnorm(1e6, mean, standard_error), seq(0,1,.001))
plot(theoretical_qs, observed_qs); abline(0,1)
```




```
cor(theoretical_qs, observed_qs)#calculate the correlation coefficient
```

```
## [1] 0.998438
```

We can see through this visualization as well as the correlation we calculated that the observed sampling distribution is indeed approximately normal. The CLT tells us that will be the case when the sample size is *large enough*.

At N=1,000, the sample size is large enough. Let's see what happens with N=2.

1. Adjust the code above to simulate drawing 100,000 samples of size N=2. Run the simulations, show the correlation, and graph the density plot.

You should see that this distribution of means (i.e. the sample distribution) is not quite normally distributed with this population at N=2. Let's find out at what sample size the sampling distribution becomes approximately normal by looking at the normality of the sampling distribution as sample sizes increase from 2 to 100. This is a fitting task to run in parallel.

To do this, you'll first need to modify the *sample_and_calculate_mean()* function so that there is a parameter N that it requires as an input.

2. Modify the *sample_and_calculate_mean()* function to take a parameter *N*. Set *N* to 10 and run it once.

If you could use some documentation on creating and modifying functions in R, see here: https://r4ds.had.co.nz/functions.html. To explain the instructions in other words, you should modify *sample_and_calculate_mean()* so that a parameter N can be set to adjust the sample size taken from the population each time it is run. Run it once with N set to 10.

---

3. Now use replicate() to run the *sample_and_calculate_mean(N=10)* function 50,000 times to simulate a sampling distribution. Calculate the correlation between the theoretical and observed quantiles as shown above.

---

4. Create a single function that takes *N* as an input, does everything specified in problem #3, and returns the correlation coefficient.

To run code in parallel using the parallel package, we should combine it into a single function with one parameter. Here is a basic framework for the function we can use here:

```
get_correlation <- function(N){
    sample_and_calculate_mean <- function(N)
    {
      "input appropriate lines of code here"
    }

    all_means <- "input code here to run sample_and_calculate_mean() 50,000 times"

    observed_qs <- quantile(all_means, seq(0, 1, .001))#extract quantiles from all_means
    theoretical_qs <- quantile(rnorm(1e6, mean(all_means), sd(all_means)), seq(0,1,.001))
    correlation_coef <- cor(theoretical_qs, observed_qs)#calculate the correlation coef.
    return(correlation_coef)
}
```

---

5. Evaluate how long it takes to run this function once at sample size N=2, and then at sample size N=100.

Here is some example code that shows generally how to measure the run time of code:

```
start <- Sys.time()#record start time
"code to evaluate goes here"
run_time <- Sys.time() - start
run_time
```

Running on an Intel Corei7 processor they each take about five minutes. Your run times will differ somewhat depending on your processor.

We are building up to running this function on integers from 2 to 100. To make it manageable, we can run it in intervals of 2 instead of every consecutive integer. If the function takes about five minutes to run once, then running it about 50 times will take about 250 minutes (~4 hours) on a single core. **Let's speed it up by modifying it to run in parallel.**

To help you get started, here is an example of running another function in parallel using parSapply():

```
#parSapply is just a parallel version of sapply
new_function <- function(x){x^2 + mean(population)}
#sapply(1:8, new_function)

#to run in parallel we can use the parallel library
library(parallel)
#detectCores()#show how many cores are available on your machine
cl <- makeCluster(2)#then make a cluster of some number of cores
clusterExport(cl, "population")#export necessary object to cores
results <- parSapply(cl, 1:8, new_function)
stopCluster(cl)
```

Note the inclusion of the clusterExport() function. If the function you are running in parallel requires any other functions or R objects to run, you need to export those to the cluster using clusterExport().

6. Modify the parallel code example above to run the get_correlation() function from 2 to 1,000 in intervals of 20.

If you do not have multiple cores on your machine, just use cl <- makeCluster(1) to make a "cluster" with one core (running on one core is not parallel, but the code can be used to run in parallel on machines with more cores).

Hint: to create a sequential vector with multiples of five you can use:

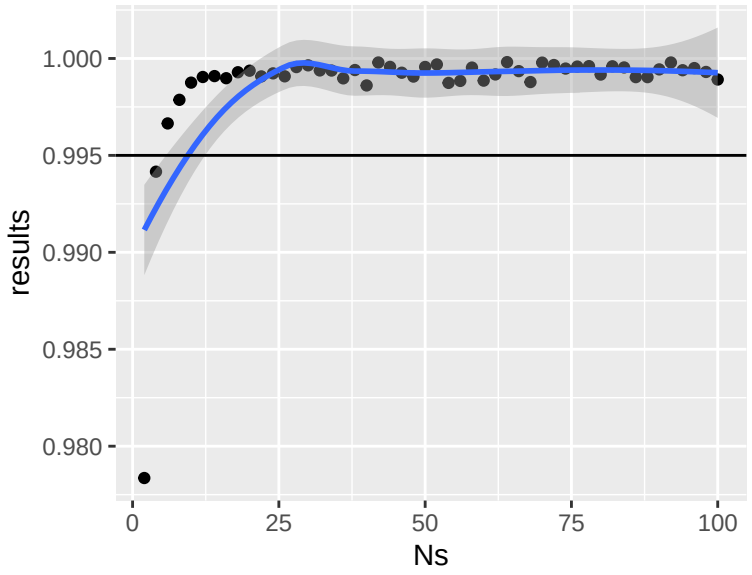
```
Ns <- seq(2,100, by=2)
head(Ns)
```

```
## [1]  2  4  6  8 10 12
```

7. Plot the results with sample size on the x axis and the correlation coefficient (the normality assessment) on the y axis.

It should look something like this:

```
library(ggplot2)
plot_data <- data.frame(Ns, results)
ggplot(plot_data, aes(Ns, results)) + geom_point() + geom_smooth(method="loess") +
  geom_hline(yintercept=.995)
```



8. At about what sample size does the sampling distribution approximate a normal distribution?

---

9. Modify the code to do the same simulations and calculations, but with the sampling distribution of sums rather than the sampling distribution of means.

The Central Limit Theorem shows that when independent random variables are summed, the distribution of sums approximates a normal distribution with large enough sample sizes. We usually think of the CLT in terms of means (since we use means more often). Means also converge to normal distributions because means are sums that have been rescaled.

OPTIONAL

Now let's use parallel computing for a different problem. In the original Google Flu Trends model, they searched over 50 million search terms for correlations with flu trends. A criticism of this approach is that it substantially left open the possibility of finding spurious correlations (i.e. correlations due to chance rather than evidence of a real relationship).

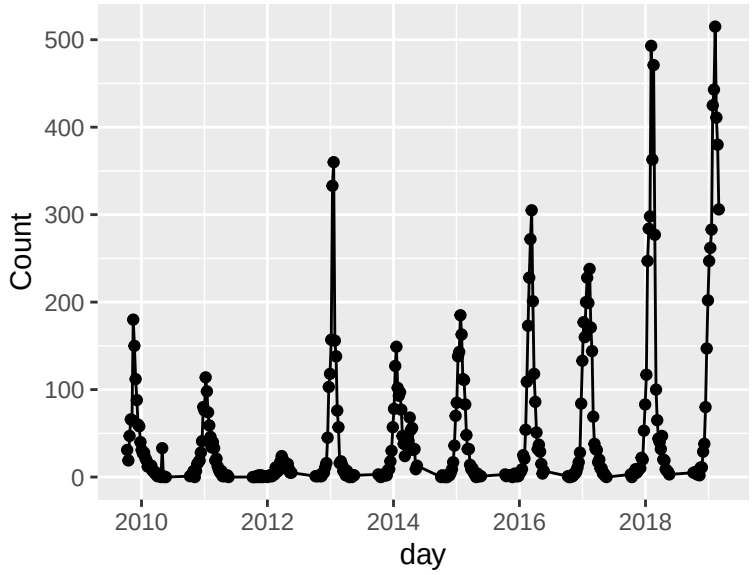
In this optional exercise, we'll load in flu infection data and check for correlations with many completely unrelated vectors of observations.

First let's load in and prep the flu data:

```
#load in NY state flu data
data <- read.csv("https://query.data.world/s/phdseqcd4htyzkazklx7rr6677chyd",
                 header=TRUE, stringsAsFactors=FALSE);
#obtained from: https://www.kaggle.com/titustitus/h1n1-new-york-2009
head(data)
```

```
##      Season Region County CDC.Week Week.Ending.Date     Disease Count
## 1 2009-2010    NYC  BRONX        3       01/23/2010 INFLUENZA_B     1
## 2 2009-2010    NYC  BRONX       17       05/01/2010 INFLUENZA_A     7
## 3 2009-2010    NYC  BRONX       17       05/01/2010 INFLUENZA_B     0
## 4 2009-2010    NYC  BRONX       50       12/19/2009 INFLUENZA_A    62
## 5 2009-2010    NYC  BRONX       50       12/19/2009 INFLUENZA_B     2
## 6 2009-2010    NYC  KINGS       12       03/27/2010 INFLUENZA_A     3
##       County.Centroid  FIPS
## 1 (40.8448, -73.8648) 36005
## 2 (40.8448, -73.8648) 36005
## 3 (40.8448, -73.8648) 36005
## 4 (40.8448, -73.8648) 36005
## 5 (40.8448, -73.8648) 36005
## 6 (40.6782, -73.9442) 36047
```

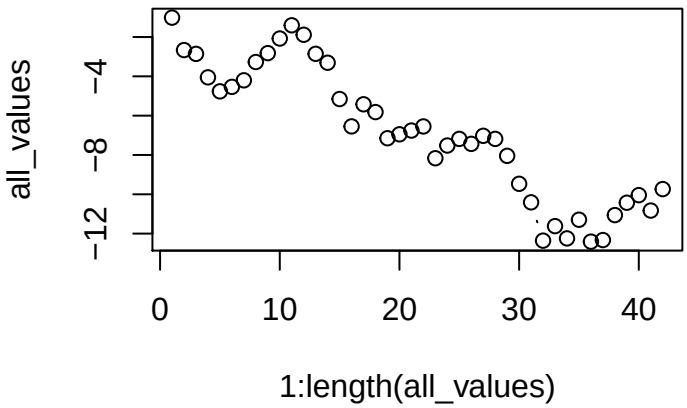
```
#subset to only Manhattan (New York County) and one strain of influenza for simplicity
data <- data[data$County=="NEW YORK",]
data <- data[data$Disease=="INFLUENZA_A",]

#create a clean day variable
data$day <- as.Date(data$Week.Ending.Date, format = "%m/%d/%Y")

#visualize the flu infection data over time
library(ggplot2)
ggplot(data, aes(day, Count)) + geom_line() + geom_point()
```




```
#subset to only one year for simplicity
data <- data[data$day>="2018-01-01",]
```

Now let's generate a vector of fake data of the same length and see if it's correlated. To generate completely unrelated but realistic data, we can simulate a "random walk":

```
all_values <- c()
 value <- 0
 for(i in 1:nrow(data))
 {
 value <- value + rnorm(1)
 all_values <- c(all_values, value)
 }
 plot(1:length(all_values), all_values, type="b")
```



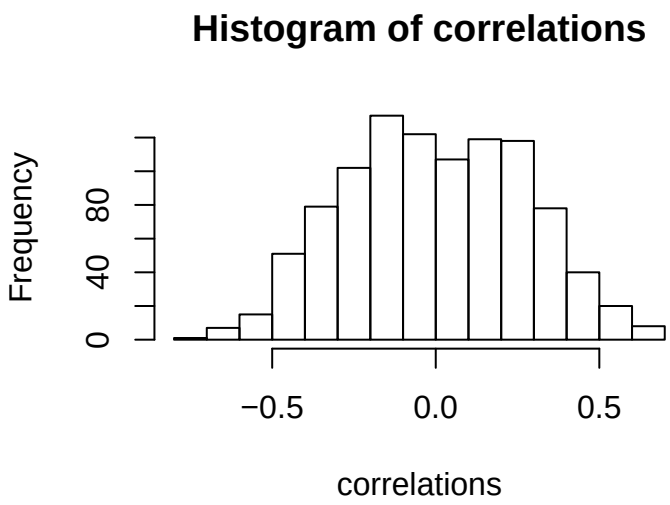
Next, let's embed that random vector code inside a function that creates a random vector and checks for a correlation with the real flu data.

```
#generate fake data and check correlation
gen_cor <- function(i){
  #generate a random walk to use as fake data
  all_values <- c()
  value <- 0
  for(i in 1:nrow(data))
  {
  value <- value + rnorm(1)
  all_values <- c(all_values, value)
  }
  data$fake_vector <- all_values
  correlation <- cor(data$fake_vector, data$Count)
  return(correlation)
}
gen_cor()
```

```
## [1] 0.3558608
```

Here is the code to check for correlations with 1,000 random vectors:

```
correlations <- sapply(1:1e3, gen_cor)
hist(correlations)
```




```
sum(abs(correlations)>.8) #how many correlations are above .8 (absolute value)
```

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

Very few, if any, correlations are above 0.8 when looking at 1,000 random vectors. But what if we look at many more?

10. (OPTIONAL) Make a parallel version of the code directly above to check for correlations with 5 million random vectors. How many of these spurious correlations were above 0.8 (absolute value)?
