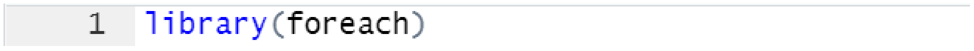
Final Project HY390.51
June 2023

Csd4656 – Christina Kopidaki
Csd4730 – Manos Platakis

Before reading the report, we need to note that:

- The code will be explained step by step, as the questions of the exercise were given
- For running our code, you will need to download the following library:



```r
library(foreach)
```

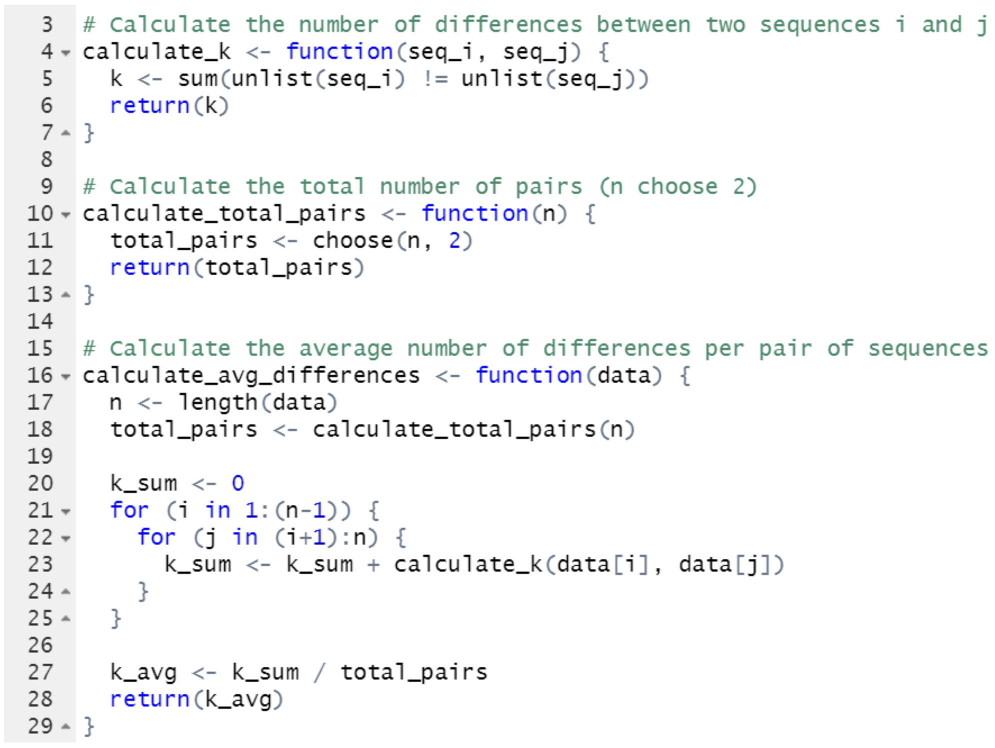
You can do that with the following command:

install.packages("foreach")

Question 1

Write your own functions to calculate each of the above and eventually the value of w, k and D.

Let's start with the **calculation of k**, the average number of pairwise differences between a pair of sequences. These are the function needed for its calculation:

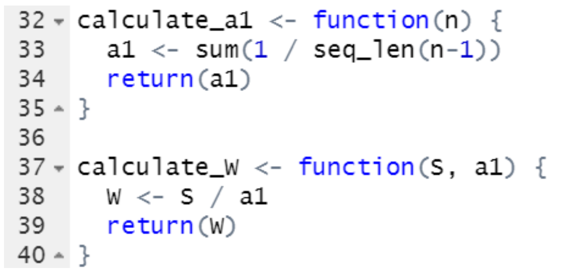


```r
# Calculate the number of differences between two sequences i and j
calculate_k <- function(seq_i, seq_j) {
  k <- sum(unlist(seq_i) != unlist(seq_j))
  return(k)
}

# Calculate the total number of pairs (n choose 2)
calculate_total_pairs <- function(n) {
  total_pairs <- choose(n, 2)
  return(total_pairs)
}

# Calculate the average number of differences per pair of sequences
calculate_avg_differences <- function(data) {
  n <- length(data)
  total_pairs <- calculate_total_pairs(n)

  k_sum <- 0
  for (i in 1:(n-1)) {
    for (j in (i+1):n) {
      k_sum <- k_sum + calculate_k(data[i], data[j])
    }
  }

  k_avg <- k_sum / total_pairs
  return(k_avg)
}
```

Let's dive in a bit deeper into the explanation of each function.

- **calculate_k:** Takes as arguments 2 sequences and returns the total differences of these sequences.
- **calculate_total_pairs:** Takes as argument a number n and returns the number of ways to choose 2 items from a set of n items without considering the order.
- **calculate_avg_differences:** Takes as argument a list of vectors and with the help of the 2 previous functions, finds the average number of differences per pair of sequences, which is the k.

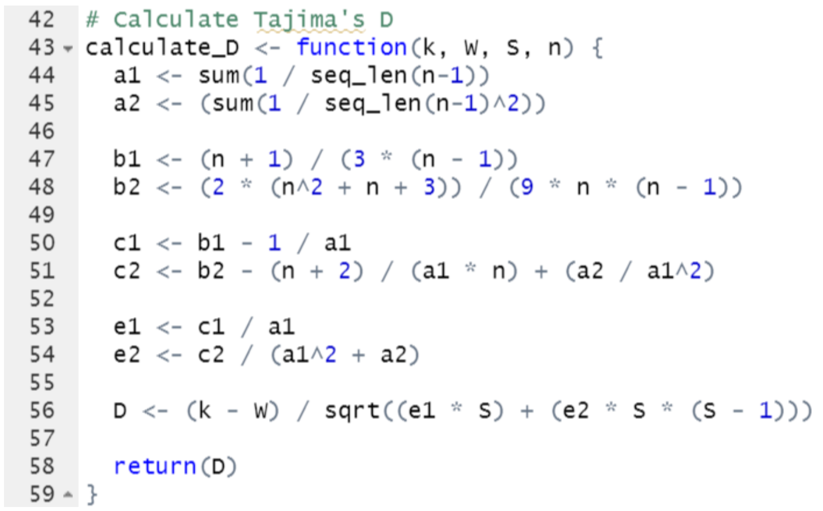
Let's continue with the **calculation of w**.



```r
calculate_a1 <- function(n) {
  a1 <- sum(1 / seq_len(n-1))
  return(a1)
}

calculate_W <- function(S, a1) {
  W <- S / a1
  return(W)
}
```

- **calculate_a1:** Takes as argument the number of sequences "n" and calculates the value of a1 by summing the reciprocal of integers from 1 to n-1.
- **calculate_W:** calculates W by dividing the number of polymorphic positions, with a1, found earlier.

Now, the **calculation of Tajima's D.**



```r
# Calculate Tajima's D
calculate_D <- function(k, W, S, n) {
  a1 <- sum(1 / seq_len(n-1))
  a2 <- (sum(1 / seq_len(n-1)^2))

  b1 <- (n + 1) / (3 * (n - 1))
  b2 <- (2 * (n^2 + n + 3)) / (9 * n * (n - 1))

  c1 <- b1 - 1 / a1
  c2 <- b2 - (n + 2) / (a1 * n) + (a2 / a1^2)

  e1 <- c1 / a1
  e2 <- c2 / (a1^2 + a2)

  D <- (k - W) / sqrt((e1 * S) + (e2 * S * (S - 1)))

  return(D)
}
```

- **calculate_D:** Takes as parameters k, W, S, and n, calculated earlier. Then calculates a1, a2, b1, b2, c1, c2, e1, e2 based on the following formulas:

$$e_1 = \frac{c_1}{a_1}, e_2 = \frac{c_2}{a_1^2 + a_2}, c_1 = b_1 - \frac{1}{a_1}, c_2 = b_2 - \frac{n+2}{a_1 n} + \frac{a_2}{a_1^2}, b_1 = \frac{n+1}{3(n-1)}, b_2 = \frac{2(n^2+n+3)}{9n(n-1)}, a_1 = \sum_{i=1}^{n-1} \frac{1}{i}, a_2 = \sum_{i=1}^{n-1} \frac{1}{i^2}$$

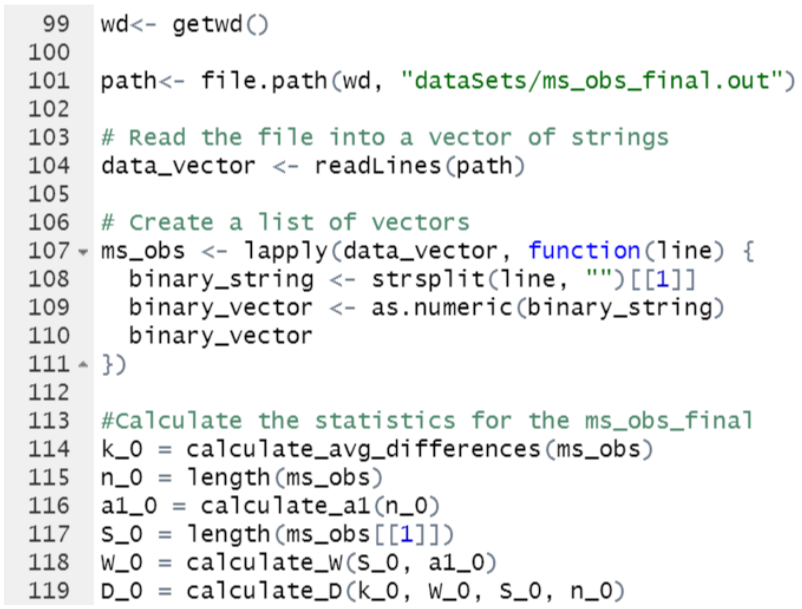
Then with the help of all these values and the following formula of Tajima's D, calculated and returns D:

$$D = \frac{k - w}{\sqrt{e_1 S + e_2 S(S-1)}}$$

<u>Question 2</u>

*Thus, for the observed dataset (ms_obs_final.out) and **each of** the simulated dataset (ms_sim_final.out), calculate the value of the three statistics. Thus, you will get a single value w_0, k_0, and D_0 for the observed dataset. You will get a vector **w = (**w_1, w_2, w_3… w_{10000}), **k = (**k_1, k_2, k_3… k_{10000}), **D = (**D_1, D_2, D_3… D_{10000})*

ms_obs_final.out



```r
wd<- getwd()

path<- file.path(wd, "dataSets/ms_obs_final.out")

# Read the file into a vector of strings
data_vector <- readLines(path)

# Create a list of vectors
ms_obs <- lapply(data_vector, function(line) {
  binary_string <- strsplit(line, "")[[1]]
  binary_vector <- as.numeric(binary_string)
  binary_vector
})

#Calculate the statistics for the ms_obs_final
k_0 = calculate_avg_differences(ms_obs)
n_0 = length(ms_obs)
a1_0 = calculate_a1(n_0)
S_0 = length(ms_obs[[1]])
W_0 = calculate_W(S_0, a1_0)
D_0 = calculate_D(k_0, W_0, S_0, n_0)
```

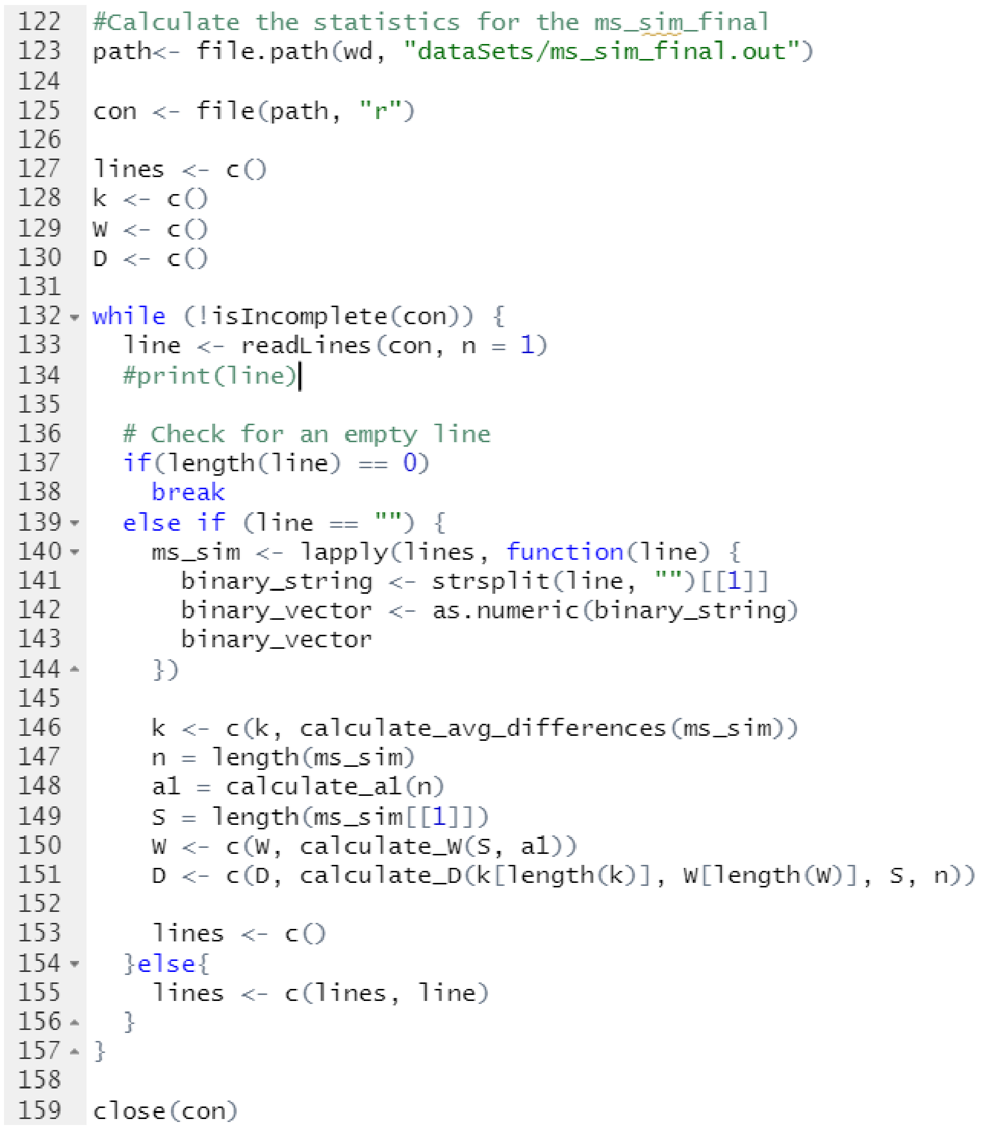
1. We first set the path to our current path and then to the ms_obs_final.out file which is in the dataSets folder.
2. Then we create a vector of strings, with each sequence being an element of the vector.
3. Then by using lapply, we create a list where each element is a vector and each vector contains the numeric values 0 or 1 that represent the sequence. More explicitly, the lapply command does the following:

    a. It applies the lapply() function to the data_vector. The lapply() function is used to apply a function to each element of a list or vector and returns a list containing the results.
    b. For each element (line) in the data_vector, it executes the function defined within lapply().
    c. Within the function, strsplit(line, "") is used to split the line into individual characters by using an empty string as the separator. The result is a list with a single element, which contains the characters of the line.
    d. [[1]] is used to access the first (and only) element of the list returned by strsplit().
    e. binary_string now holds the character vector obtained from the splitting operation.
    f. as.numeric(binary_string) converts each character in binary_string into a numeric value. The characters are "0" and "1" representing a binary string, so this operation converts them into the numeric values 0 and 1, respectively.

g. binary_vector now holds the numeric vector obtained from the conversion.
h. The binary_vector is the result returned by the function defined within lapply().
i. After iterating through all elements in the data_vector, lapply() returns a list (ms_obs) containing the converted binary vectors for each element of data_vector. Each element in ms_obs corresponds to the result obtained from the function applied to the corresponding element in data_vector.

4. Now we have our data structure with all of the sequences so we are ready to call the functions we talked about earlier (Question 1). We have to note that the number of sequences(n_0) is calculated by finding the size of our list and the number of polymorphic positions(S_0) is the length of any of our vectors so we find it by taking the length(ms_obs[[1]]). The values of k, W and D for the ms_obs_final.out are saved in the k_0, W_0 and D_0 variables.

ms_sim_final.out



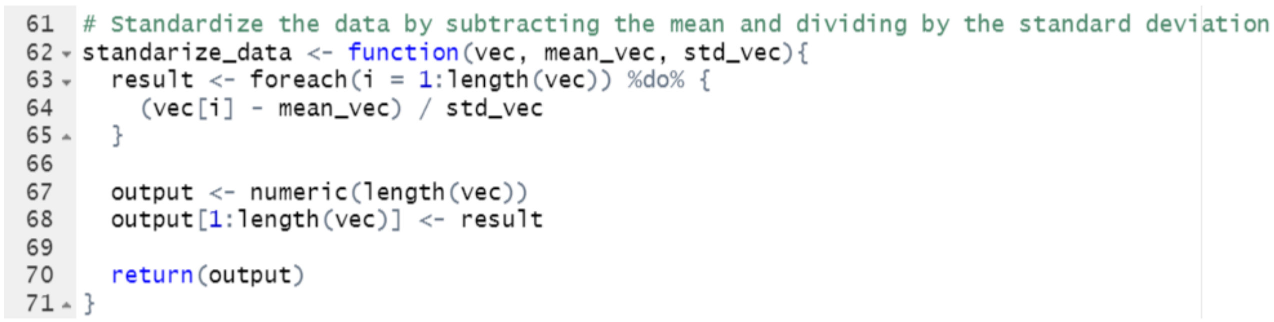
```
#Calculate the statistics for the ms_sim_final
path<- file.path(wd, "dataSets/ms_sim_final.out")

con <- file(path, "r")

lines <- c()
k <- c()
W <- c()
D <- c()

while (!isIncomplete(con)) {
  line <- readLines(con, n = 1)
  #print(line)

  # Check for an empty line
  if(length(line) == 0)
    break
  else if (line == "") {
    ms_sim <- lapply(lines, function(line) {
      binary_string <- strsplit(line, "")[[1]]
      binary_vector <- as.numeric(binary_string)
      binary_vector
    })

    k <- c(k, calculate_avg_differences(ms_sim))
    n = length(ms_sim)
    a1 = calculate_a1(n)
    S = length(ms_sim[[1]])
    W <- c(W, calculate_W(S, a1))
    D <- c(D, calculate_D(k[length(k)], W[length(W)], S, n))

    lines <- c()
  }else{
    lines <- c(lines, line)
  }
}

close(con)
```

1. We now set the path to the ms_sim_final.out, the file that contains the simulated data and create a connection for the file in read mode.
2. Then we create 4 empty vectors. One for the lines we have read and one for each statistic k, W and D.
3. With the help of a while loop with the condition “!isIncomplete(con)”, we read one by one the lines and save them to the lines vector (else case).
4. The if statement (length(line) == 0) checks if the file has ended and the second condition (line == “”) checks if the line we just read is an empty line. In that case, it means the dataset we want to process is complete so we do the following:
    a. We save our current dataset into a list of vectors, using lapply() as we previously explained.
    b. We calculate each of the statistics k, W and D, using the functions we created earlier and we save the results into the according vectors.
    c. We empty the lines vector in order to save there the next dataset, if there is one.
5. After we have read and processed the whole file, we close the connection.

Question 3

Normalize each of the vectors, i.e. for each of the vectors w, k, D, (e.g. for w) estimate the mean (You can use the function mean() in R) and the variance (you can use the function var() in R) and find the normalized values $w_i' = \frac{w_i - \bar{w}}{var(\mathbf{w})}$ *(do the same for the k and D values). Finally using the same and the same var(w) transform the w0. (Do the same for the k0 and D0 values).*

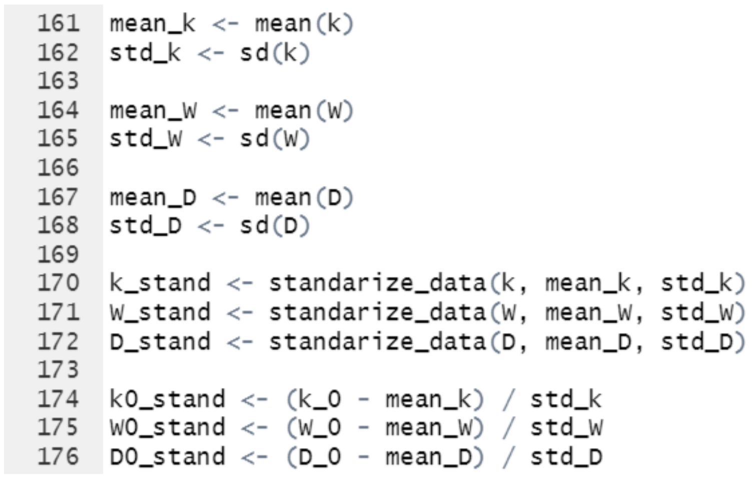
The next step is the standarization of our data. This is accomplished with the formula given above.



```r
# Standardize the data by subtracting the mean and dividing by the standard deviation
standarize_data <- function(vec, mean_vec, std_vec){
  result <- foreach(i = 1:length(vec)) %do% {
    (vec[i] - mean_vec) / std_vec
  }

  output <- numeric(length(vec))
  output[1:length(vec)] <- result

  return(output)
}
```

In order to perform the standarization on all the values of the vectors we created the **standarize_data** function.

- **standarize_data:** Takes 3 parameters, a vector, the mean of its values and the standard deviation of its values. It standarizes each value of the vector and returns the results in a new vector.

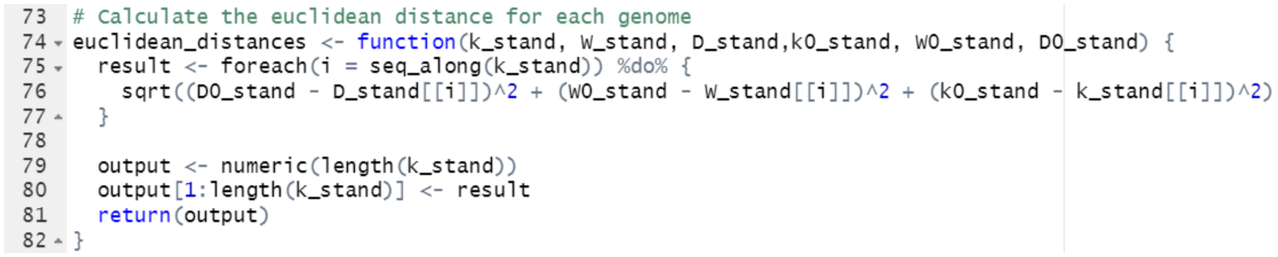


```r
mean_k <- mean(k)
std_k <- sd(k)

mean_W <- mean(W)
std_W <- sd(W)

mean_D <- mean(D)
std_D <- sd(D)

k_stand <- standarize_data(k, mean_k, std_k)
W_stand <- standarize_data(W, mean_W, std_W)
D_stand <- standarize_data(D, mean_D, std_D)

k0_stand <- (k_0 - mean_k) / std_k
W0_stand <- (W_0 - mean_W) / std_W
D0_stand <- (D_0 - mean_D) / std_D
```

The code above calculates all the means and standard deviations needed and calls the standardize_data function 3 times, one for each vector. We then also standardize the values of k_0, W_0 and D_0. This means we have 3 new vectors with the standardized values of the simulated data and 3 new variables with the standardized values of the observed data.

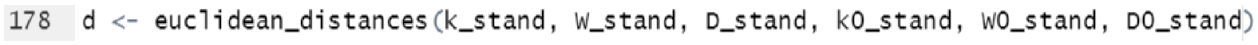
Question 4

Write a function that will calculate the Euclidean distances between the observed and each of the simulated datasets. Remember that a Euclidean distance $d = \sqrt{(D_0' - D_1')^2 + (w_0' - w_1')^2 + (k_0' - k_1')^2}$ *Let these distances be: d1, d2, … d10000. Now, the important issue is that for each of these distances, it corresponds a parameter value in pars_final.txt .*



```r
# Calculate the euclidean distance for each genome
euclidean_distances <- function(k_stand, W_stand, D_stand,k0_stand, W0_stand, D0_stand) {
  result <- foreach(i = seq_along(k_stand)) %do% {
    sqrt((D0_stand - D_stand[[i]])^2 + (W0_stand - W_stand[[i]])^2 + (k0_stand - k_stand[[i]])^2)
  }

  output <- numeric(length(k_stand))
  output[1:length(k_stand)] <- result
  return(output)
}
```

- **euclidean_distances:** Takes the three vectors and the three values of the observed data, all standardized and finds the Euclidean distance for each dataset according to the formula given. It returns all the Euclidean distances in a list.

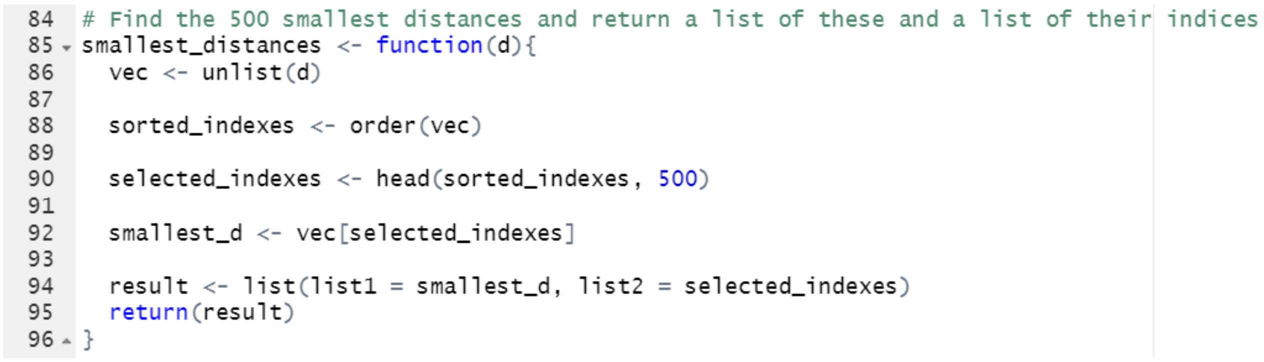


```r
d <- euclidean_distances(k_stand, W_stand, D_stand, k0_stand, W0_stand, D0_stand)
```

The call of the previous function in our code.

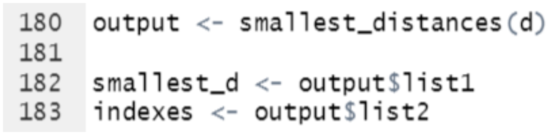
Question 5

Find the 500 smallest distances and the keep the indexes.



```r
# Find the 500 smallest distances and return a list of these and a list of their indices
smallest_distances <- function(d){
  vec <- unlist(d)

  sorted_indexes <- order(vec)

  selected_indexes <- head(sorted_indexes, 500)

  smallest_d <- vec[selected_indexes]

  result <- list(list1 = smallest_d, list2 = selected_indexes)
  return(result)
}
```

- **smallest_distances:** Takes as an argument the list of the Euclidean distances and returns a list which contains two lists. The first one contains the numeric values of the smallest Euclidean distances and the second one contains the indices of these smallest values in the initial list.

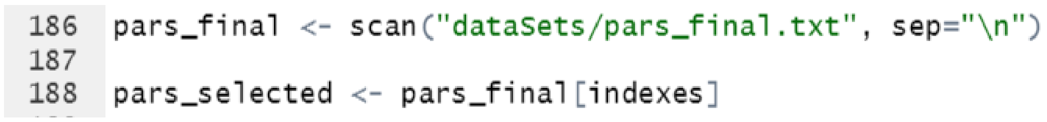


```r
output <- smallest_distances(d)

smallest_d <- output$list1
indexes <- output$list2
```

The call of the previous function in our code and how we save the smallest distances and their indexes which are saved in the return value, in 2 separate lists.

<u>Question 6</u>

Get the corresponding values from the pars_final.txt.



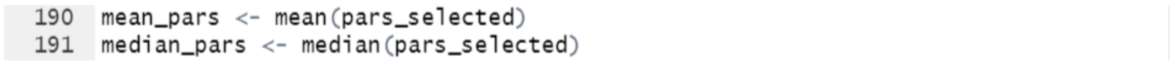
```r
pars_final <- scan("dataSets/pars_final.txt", sep="\n")

pars_selected <- pars_final[indexes]
```

When **scan()** is executed with the specified file path and separator, it reads the contents of the text file and stores them in the **pars_final** variable. Each line of the file will correspond to an element in the resulting data, creating a vector-like structure.

Then, we create a new variable named **pars_selected** that contains a subset of elements from the **pars_final** vector based on the specified **indexes.**

<u>Question 7</u>

Calculate the mean, the median of these 500 values.

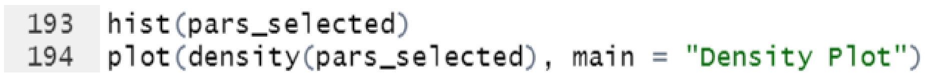


```r
mean_pars <- mean(pars_selected)
median_pars <- median(pars_selected)
```

The mean and the median of these 500 values is simply calculated using the mean() and median() functions of R.

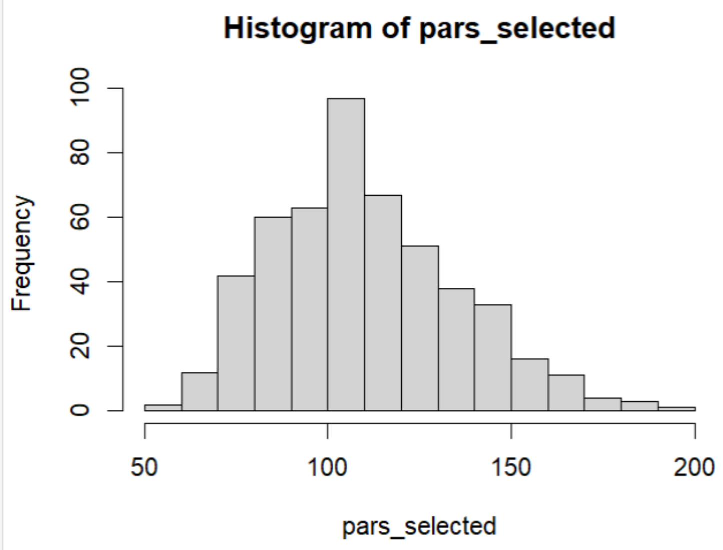
Question 8

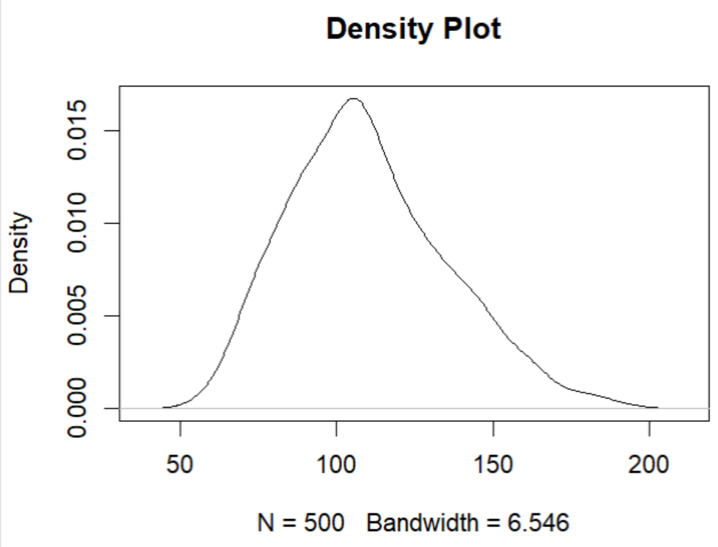
Construct the histogram (hint function hist) and the density plot (hint function density) of these 500 values.



```r
hist(pars_selected)
plot(density(pars_selected), main = "Density Plot")
```

With the help of the given functions we construct the histogram and the density plots of the 500 values and we get the following results:





Question 9

Explain the final result, i.e., if the rate values in pars_final.txt correspond to the time period of a year, then what does it mean for the population of the SARS-CoV-2?

In the context of our analysis, the parameter values in **pars_final.txt** represent the time period of a year. These parameter values are associated with the simulated datasets, which were generated based on specific parameter settings.

When we calculated the Euclidean distances between the observed dataset and each of the simulated datasets, we also kept track of the parameter value corresponding to each distance. The histogram of these parameter values shows the distribution of parameter values that correspond to the 500 closest matches between the observed dataset and the simulated datasets.

The histogram peaks around a value of 100. This means that a large number of the closest matches have parameter values around 100 in **pars_final.txt**.

This implies that the simulated datasets with a growth rate of approximately 100 per year are the closest matches to the observed dataset, as determined by the Euclidean distances.