Final Project

Dawson Kinsman and Sabastian Zuhorski

2023-04-25

Question 1: Infectious Disease

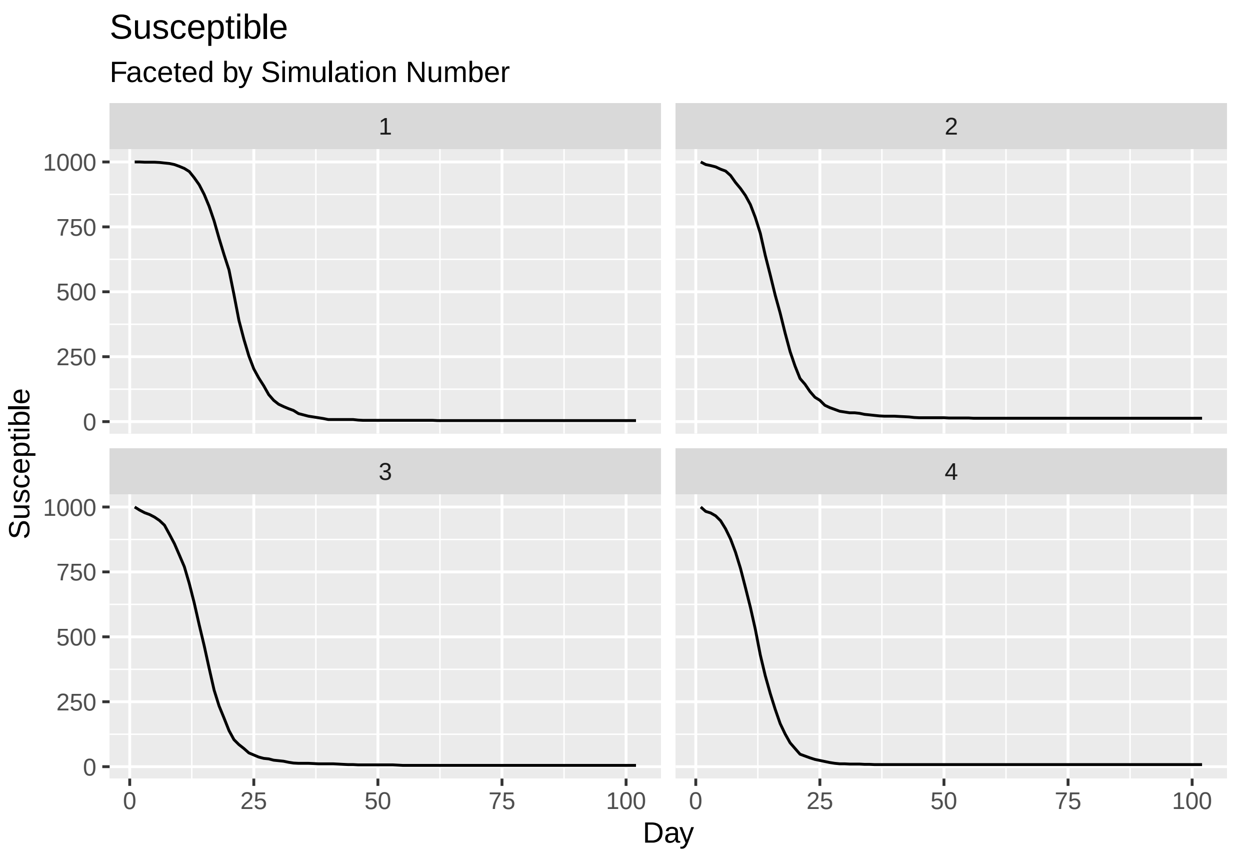
- Below is the code used to model an infectious disease that may turn into an epidemic. It is based upon the SIR epidemic model. The function starts with a few parameters that you want to initialize your simulation with: initial_infection_rate, removal_rate, initial_susceptible, simulation_length, initial_number_infected. Then we used a few variables that store the initial_infection_rate and initial_susceptible variables as they will be dynamically changing throughout. After this comes the need to store the outputs of the model calculations, variables s, i, r, and day. We then start our loop. We will loop through the number of times the simulation_length was entered in as. each iteration is considered a day which will be represented as 'd'. So to initialize, we start on day zero where we have our initial conditions. We go through and add the different value calculations to our s, i, r and day variables. After weve looped through the simualtion length, we create a dataframe to ourganize our data together nicely.

```
library(ggplot2)
library(tidyverse)

SIR = function(initial_infection_rate, removal_rate, initial_susceptible, simulation_length, initial_nu

  N = initial_susceptible
  alpha = initial_infection_rate
  B = removal_rate

  s = c()
  i = c()
  r = c()
  day = c()

  for (d in 0:simulation_length){
   if (d == 0){
     p = 1 - (1 - alpha)^initial_number_infected
     s[d+1] = N
     i[d+1] = initial_number_infected
     r[d+1] = 0
     day[d+1] = 1

     splus1 = sum(rbinom(s[1], 1, (1-alpha)^initial_number_infected))
     rplus1 = sum(rbinom(1,1,B))
     iplus1 = N + 1 - rplus1 - splus1

     s[2] = splus1
     i[2] = iplus1
```

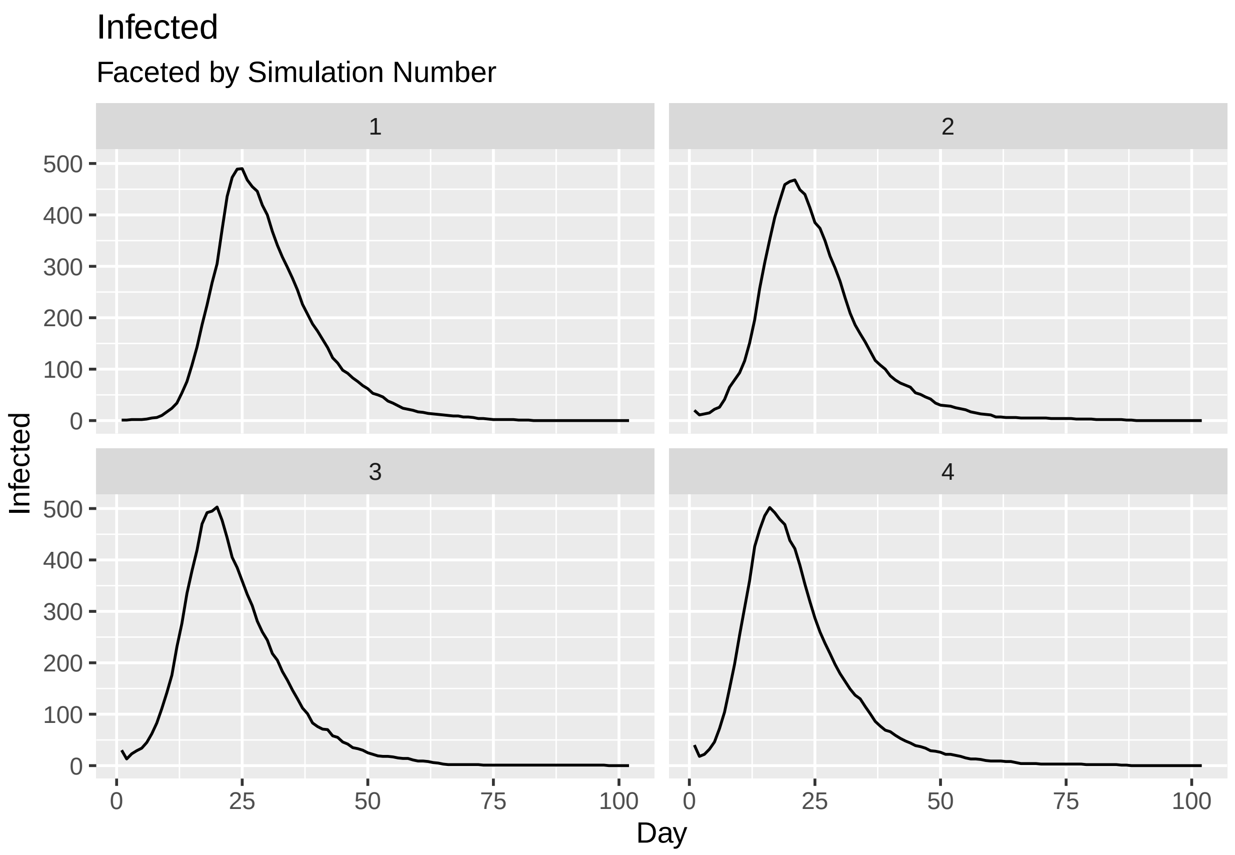
```
      r[2] = rplus1
      day[2] = 2

    } else {
      p = 1 - (1 - alpha)^i[d+1]

      splus1 = sum(rbinom(s[d+1], 1, (1-alpha)^i[d+1]))
      rplus1 = r[d+1] + sum(rbinom(i[d+1],1,B))
      iplus1 = N + 1 - rplus1 - splus1

      s[d+2] = splus1
      i[d+2] = iplus1
      r[d+2] = rplus1
      day[d+2] = d+2

    }
  }
  df = data.frame(list(Day = day, Susceptible = s, Infected = i, Recover = r))
  return(df)
}
```

Looking at what happens when you increasing the number of people infected at the start of the simulation

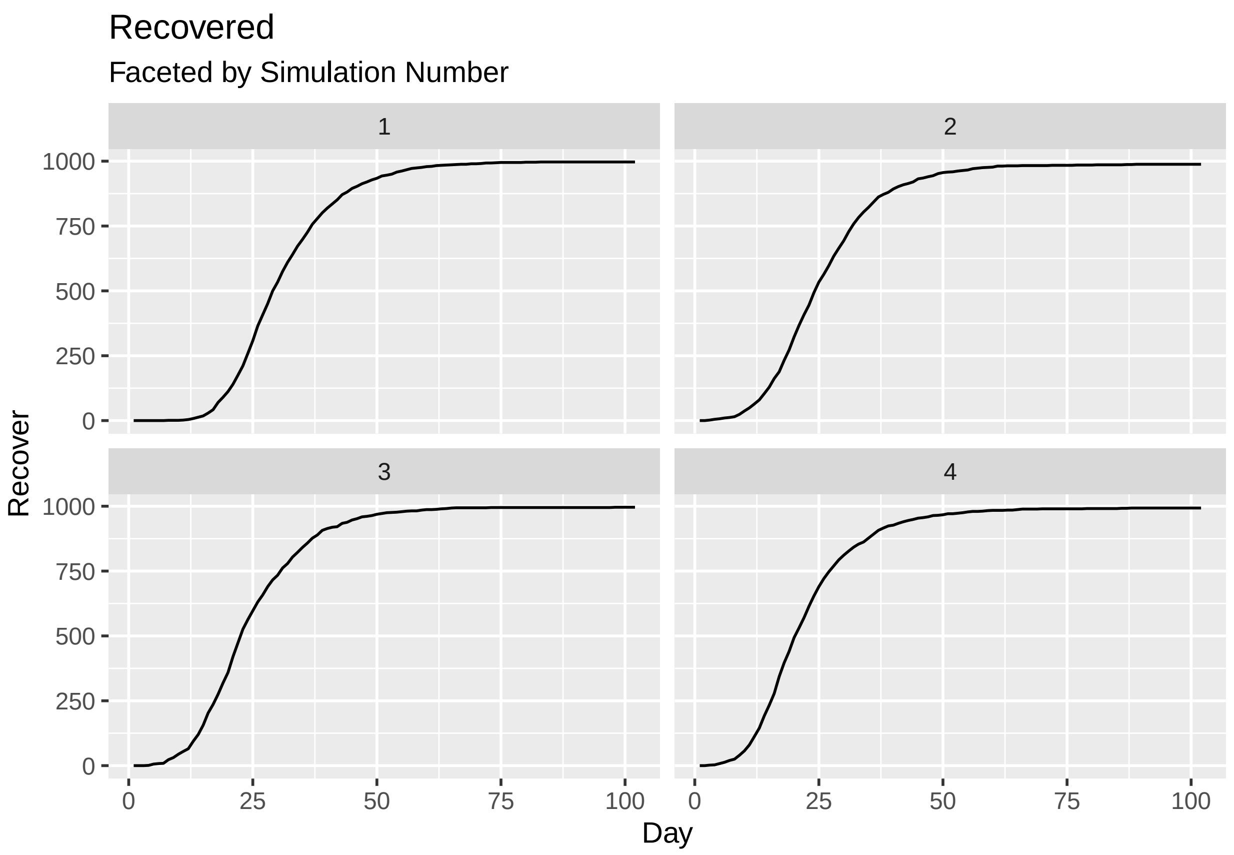
```
set.seed(4)
epidemic1 = SIR(0.0005, 0.1, 1000, 100, 1)
epidemic1 = epidemic1 %>% mutate(SIM = 1)

epidemic2 = SIR(0.0005, 0.1, 1000, 100, 20)
epidemic2 = epidemic2 %>% mutate(SIM = 2)

epidemic3 = SIR(0.0005, 0.1, 1000, 100, 30)
epidemic3 = epidemic3 %>% mutate(SIM = 3)

epidemic4 = SIR(0.0005, 0.1, 1000, 100, 40)
epidemic4 = epidemic4 %>% mutate(SIM = 4)

new = rbind(epidemic1, epidemic2, epidemic3, epidemic4)

ggplot(new, aes(x=Day))+geom_line(aes(y=Susceptible))+facet_wrap(~SIM)+ggtitle("Susceptible", subtitle
```



```
ggplot(new, aes(x=Day))+geom_line(aes(y=Infected))+facet_wrap(~SIM)+ggtitle("Infected", subtitle = "Face
```



```
ggplot(new, aes(x=Day))+geom_line(aes(y=Recover))+facet_wrap(~SIM)+ggtitle("Recovered", subtitle = "Face
```

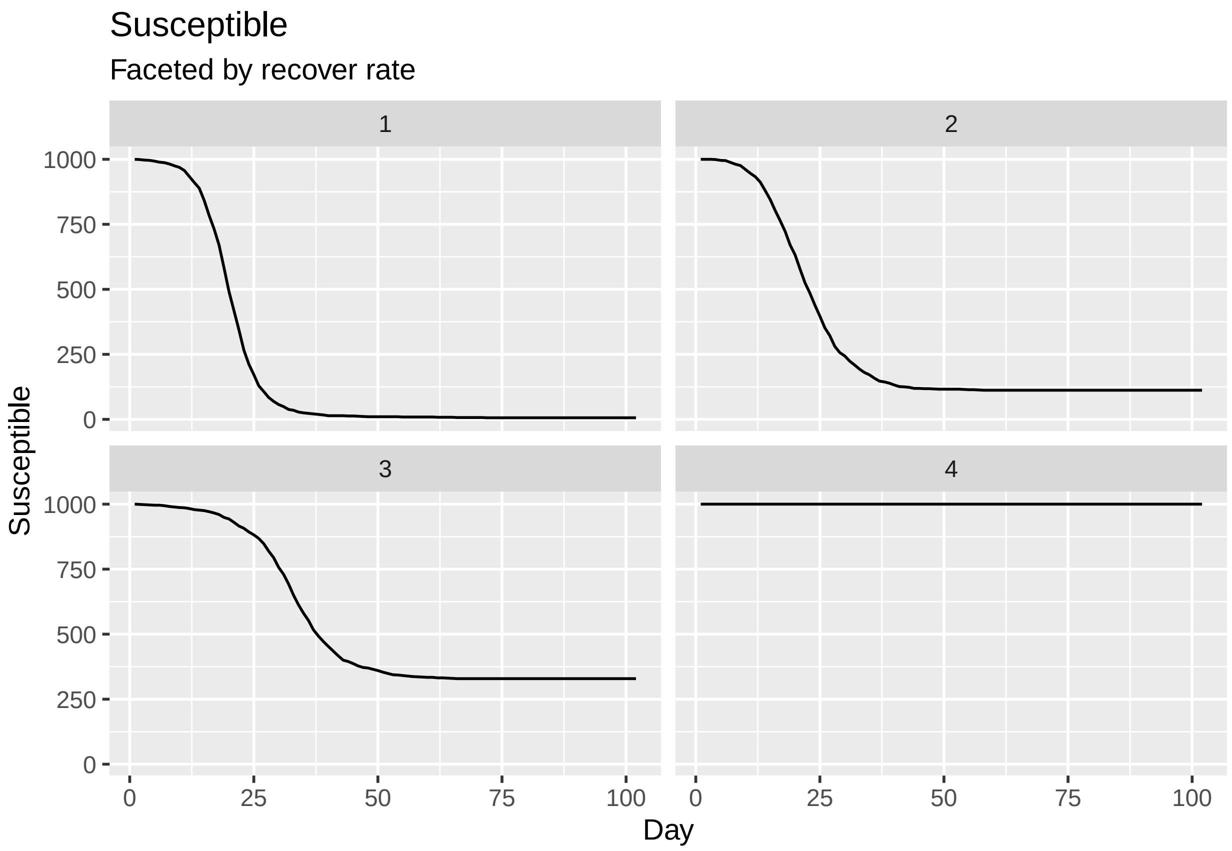


For this question we look at for different simulations where the only parameter that changes is the number of people infected at the start of the simulation (1, 20, 30, 40 people). Seen in plot number 1 for the susceptible data, we can see it takes a little bit longer to for infections to spread compared to the other plots. The other 3 plots have a more pronounced slope to them fairly quickly into the epidemic. - Switching over to the graphs representing the infected, we can see that as we move along to the 4 graph which has the most people infected to begin with, the epidemic reaches its peak in half the amount of time as the first plot. This would be expected as the more people that start out with it means there are more people going around infected other people. - Finally we look at the graph that details the recovered people. Surprisingly the plots all show a fairly similar recovery. It appears that the amount of people infected to begin with didnt play a factor here. Although future simulations may show differently if we were to use different numbers to start with.

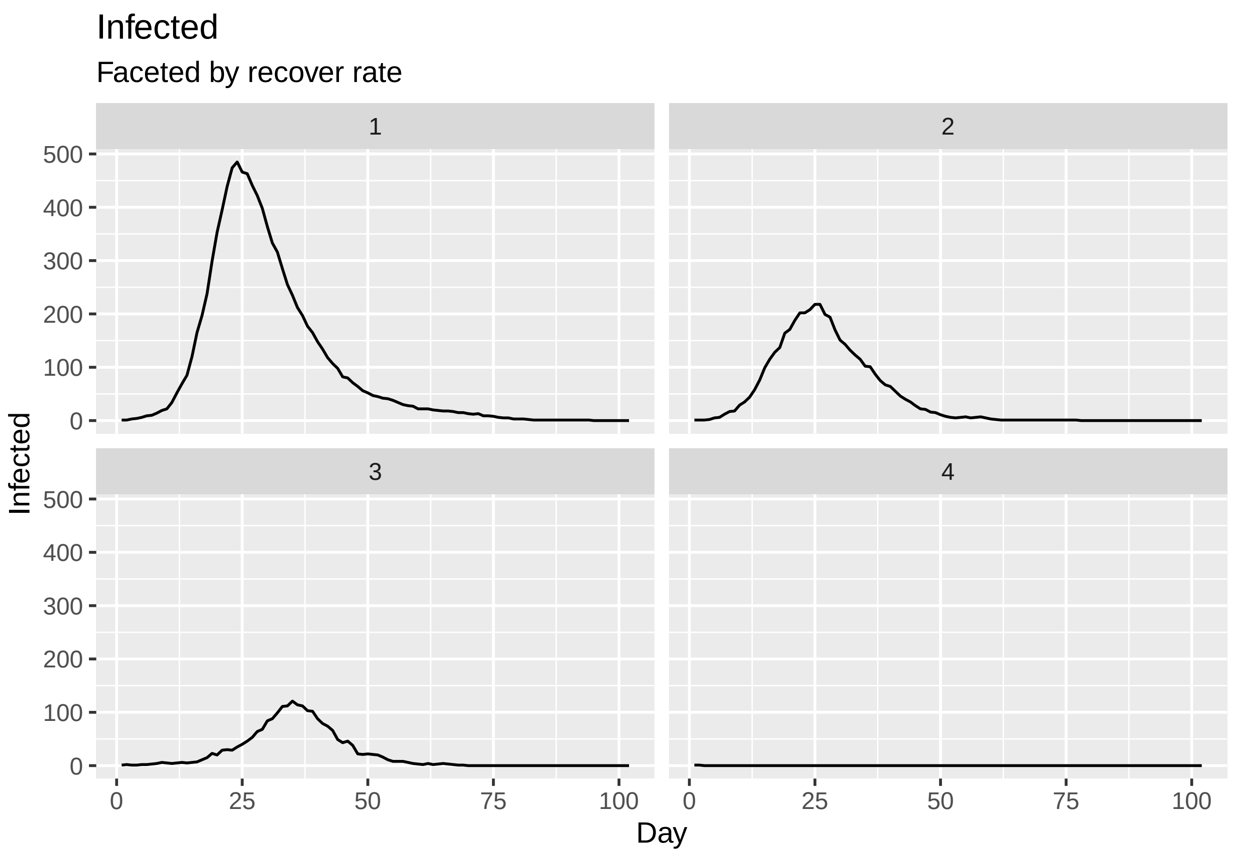
Looking into increasing the recover rate of the disease for a given simulation

```r
set.seed(17)
epidemic1 = SIR(0.0005, 0.1, 1000, 100, 1)
epidemic1 = epidemic1 %>% mutate(SIM = 1)

epidemic2 = SIR(0.0005, 0.2, 1000, 100, 1)
epidemic2 = epidemic2 %>% mutate(SIM = 2)

epidemic3 = SIR(0.0005, 0.3, 1000, 100, 1)
epidemic3 = epidemic3 %>% mutate(SIM = 3)

epidemic4 = SIR(0.0005, 0.4, 1000, 100, 1)
epidemic4 = epidemic4 %>% mutate(SIM = 4)

new = rbind(epidemic1, epidemic2, epidemic3, epidemic4)
```

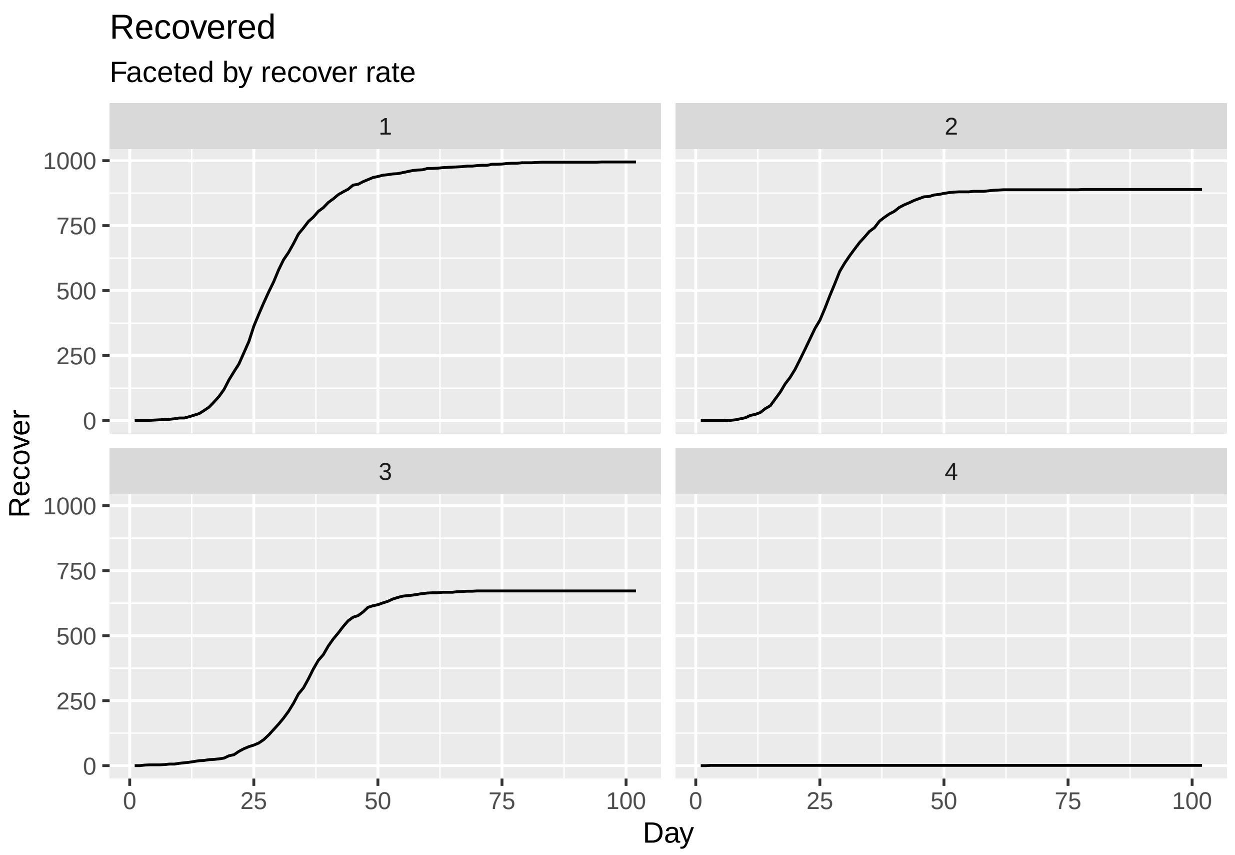
```
ggplot(new, aes(x=Day))+geom_line(aes(y=Susceptible))+facet_wrap(~SIM)+ggtitle("Susceptible", subtitle =
```



```
ggplot(new, aes(x=Day))+geom_line(aes(y=Infected))+facet_wrap(~SIM)+ggtitle("Infected", subtitle = "Fac
```



```
ggplot(new, aes(x=Day))+geom_line(aes(y=Recover))+facet_wrap(~SIM)+ggtitle("Recovered", subtitle = "Face
```

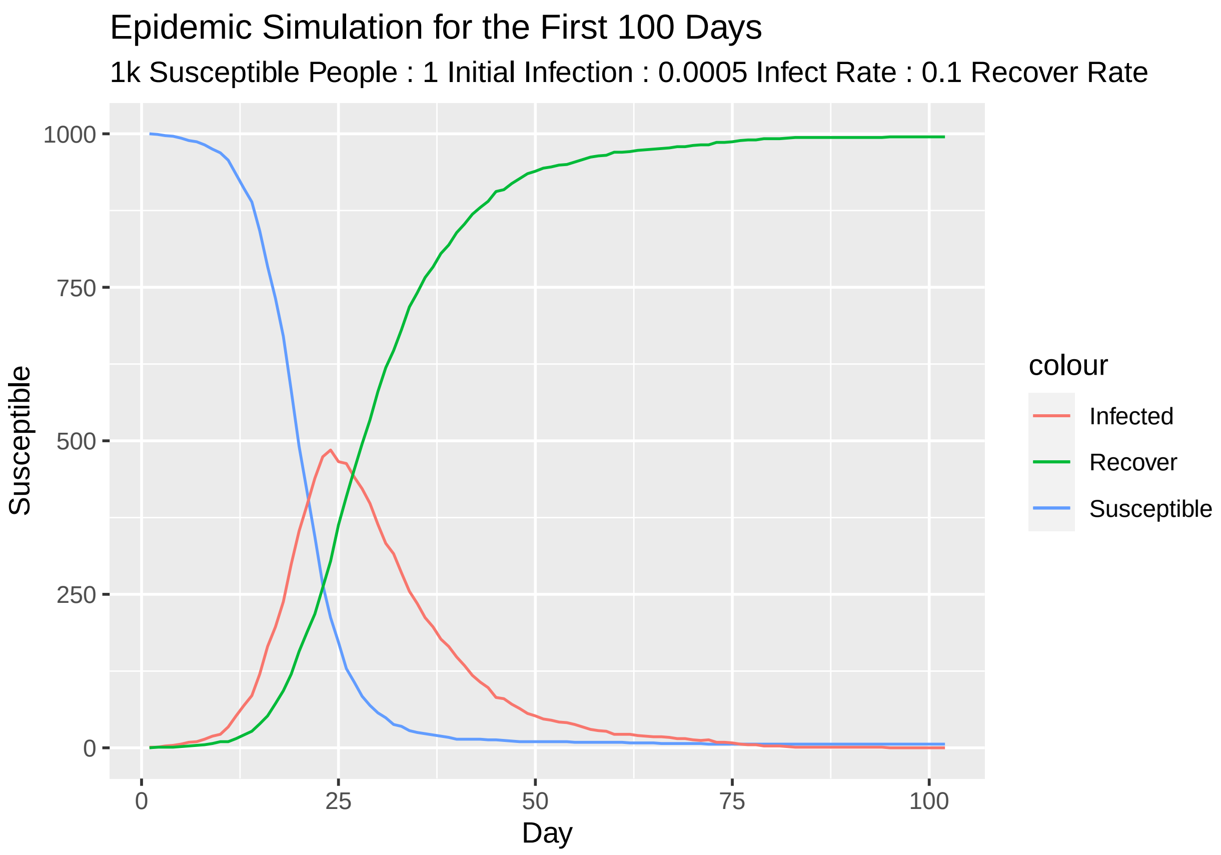


-

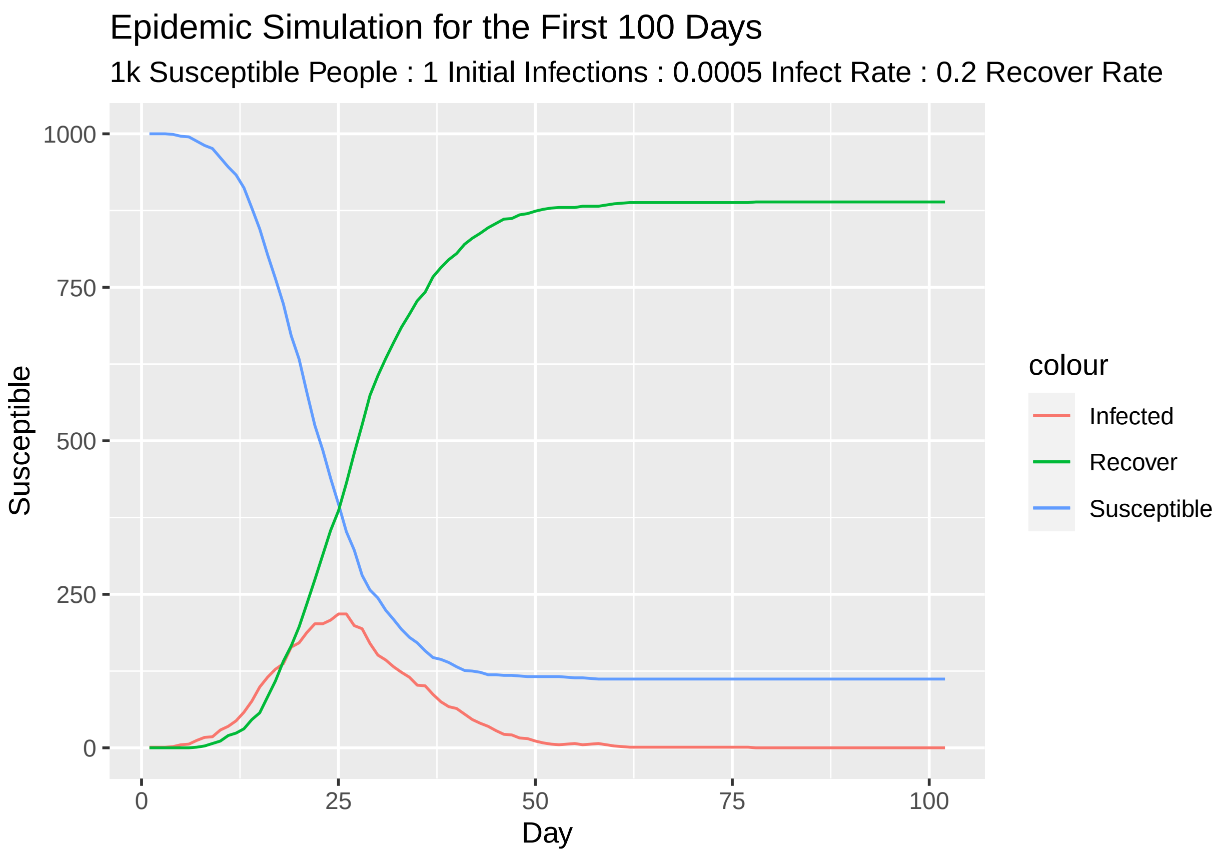
Starting with the graph for the Susceptible, we can see that as the the recover rate increases, the more people there are that remain susceptible to the disease. People may be recovering to quick before the disease can infect another person. - Looking at the infection graph, the amount of infections at the peak drops by nearly half every time we increase he recovery rate by 10 percent. The peak also takes longer to get to as we move to a higher recovery rate. - The graph for recovery appears to be relatively the same for each plot, given that each plot for each plot the recovery rate is higher and less people are infected as previously stated.

A visualization of the graphs previously shown except with S,I,R all plotted on the same plot.

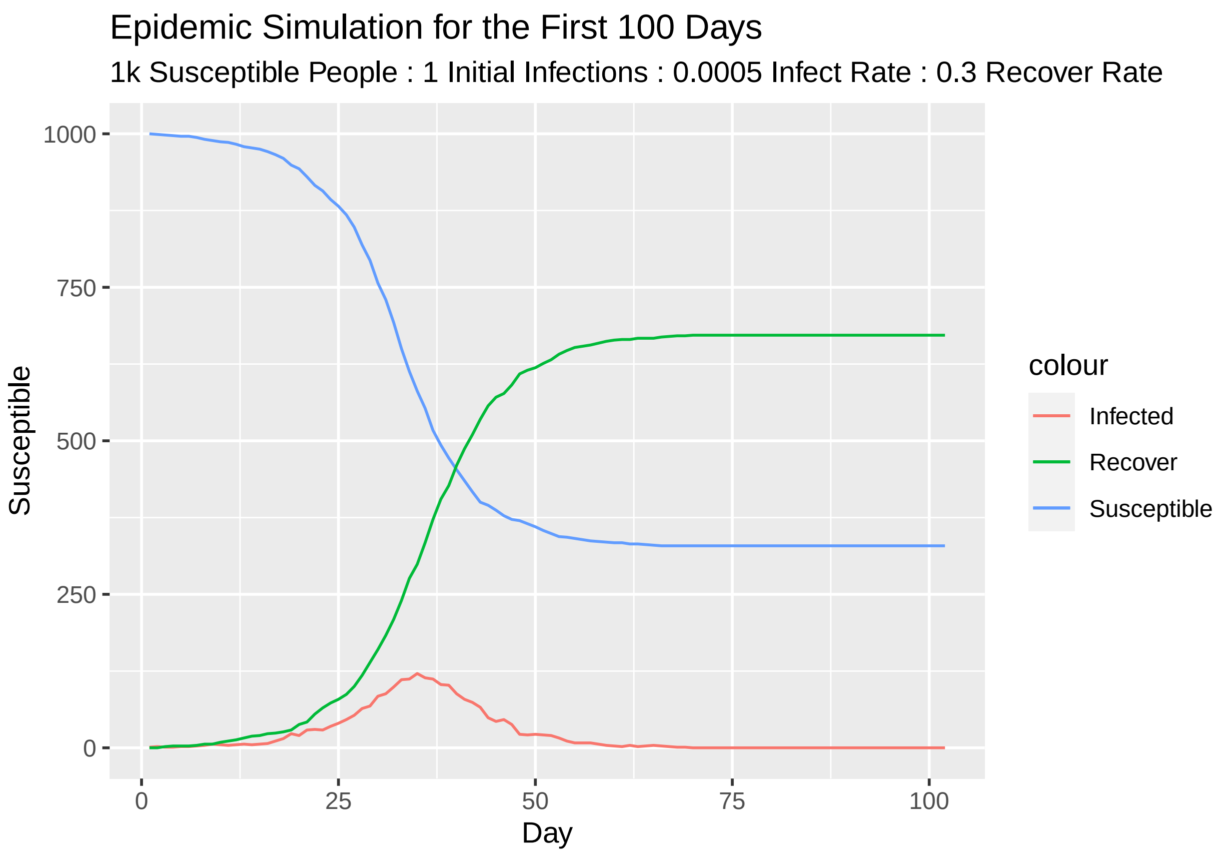
```
ggplot(epidemic1)+
  geom_line(aes(x = Day, y = Susceptible, color="Susceptible"))+
  geom_line(aes(x = Day, y = Infected, color="Infected"))+
  geom_line(aes(x = Day, y = Recover, color="Recover"))+
  labs(title="Epidemic Simulation for the First 100 Days", subtitle = "1k Susceptible People : 1 Initia
```



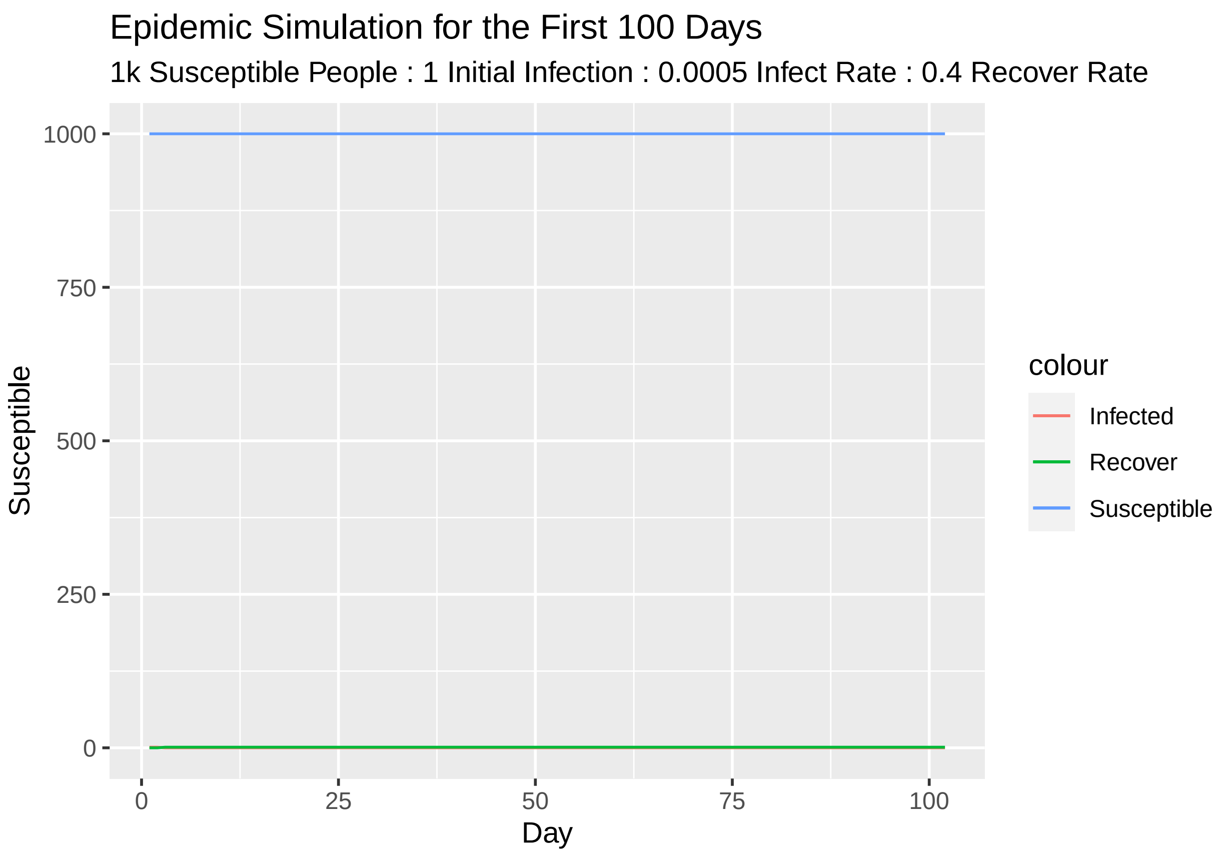
```
ggplot(epidemic2)+
  geom_line(aes(x = Day, y = Susceptible, color="Susceptible"))+
  geom_line(aes(x = Day, y = Infected, color="Infected"))+
  geom_line(aes(x = Day, y = Recover, color="Recover"))+
  labs(title="Epidemic Simulation for the First 100 Days", subtitle = "1k Susceptible People : 1 Initial
```



```
ggplot(epidemic3)+
  geom_line(aes(x = Day, y = Susceptible, color="Susceptible"))+
  geom_line(aes(x = Day, y = Infected, color="Infected"))+
  geom_line(aes(x = Day, y = Recover, color="Recover"))+
  labs(title="Epidemic Simulation for the First 100 Days", subtitle = "1k Susceptible People : 1 Initial
```



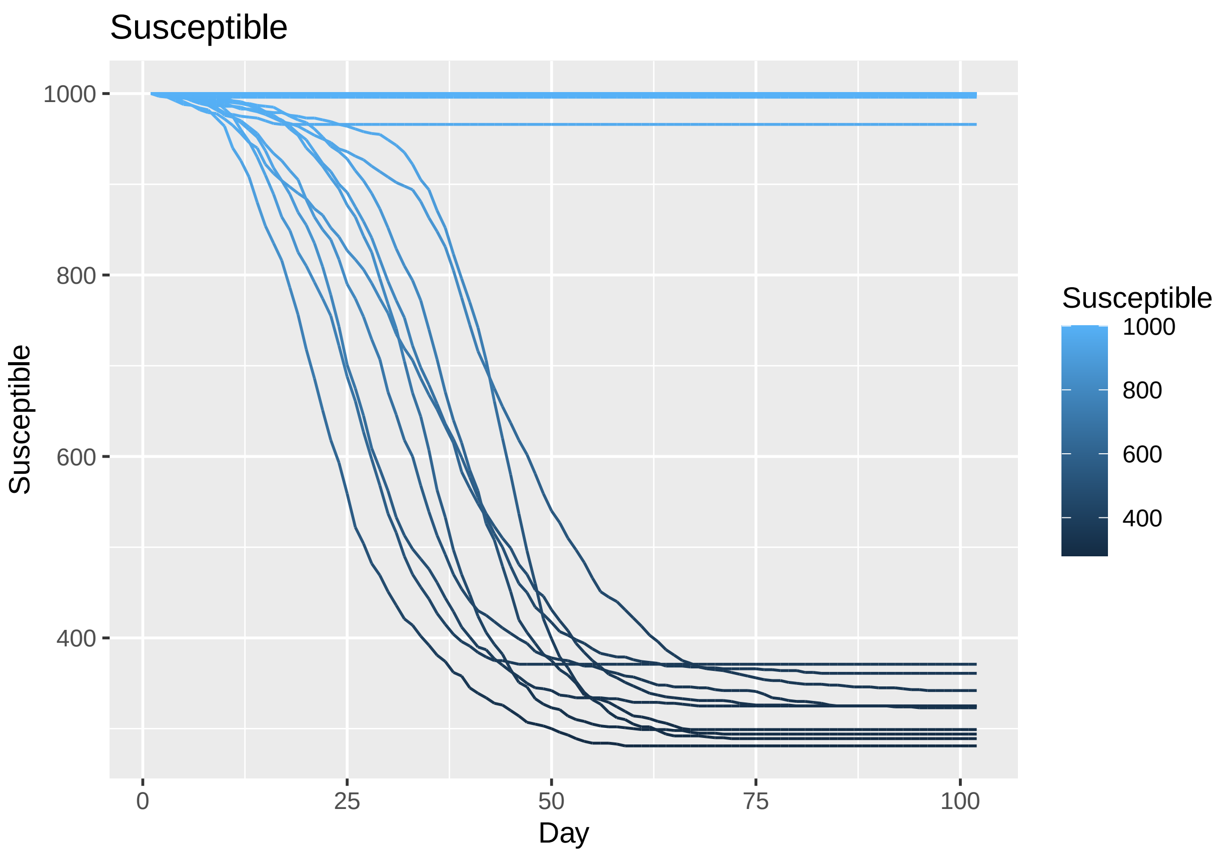
```
ggplot(epidemic4)+
  geom_line(aes(x = Day, y = Susceptible, color="Susceptible"))+
  geom_line(aes(x = Day, y = Infected, color="Infected"))+
  geom_line(aes(x = Day, y = Recover, color="Recover"))+
  labs(title="Epidemic Simulation for the First 100 Days", subtitle = "1k Susceptible People : 1 Initial
```



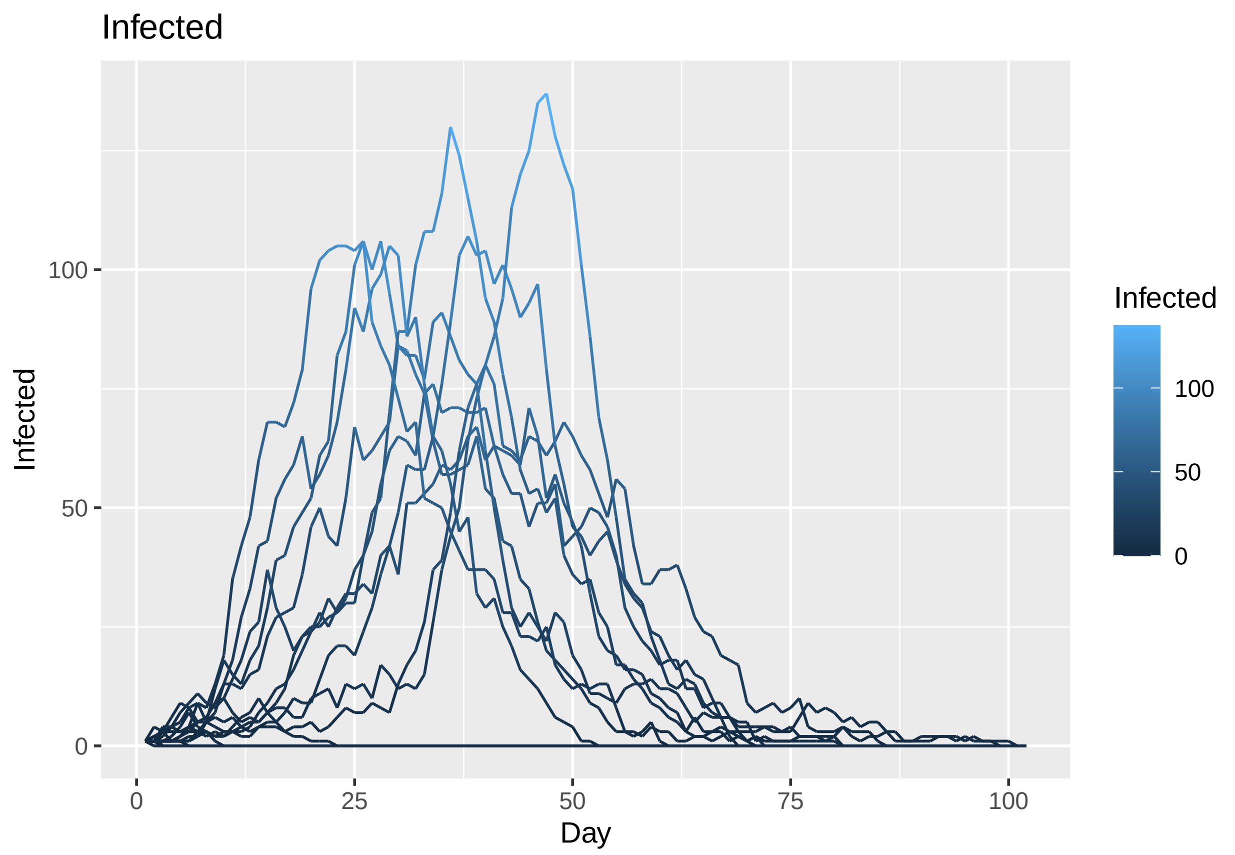
Question 2: 20 Simualtions with an infection rate of 0.0005 and a recover rate of 0.3

- Below is a a loop for 20 simulations of the epidemic with an infection rate of 0.0005 and a recovery rate of 0.3. So we call the SIR function and do a simulation. The results of the simulation are saved to a variable called df. Since the results of the SIR function returns a dataframe, i add a column to represent what simulation just occured with that data. If it the first iteration, ill just assign df to a new variable called "new". On the contrary, if the iteration is 2 or more then we bind together our newest simulation dataframe result with our "new" variable which holds all of the simulations weve done and combined.

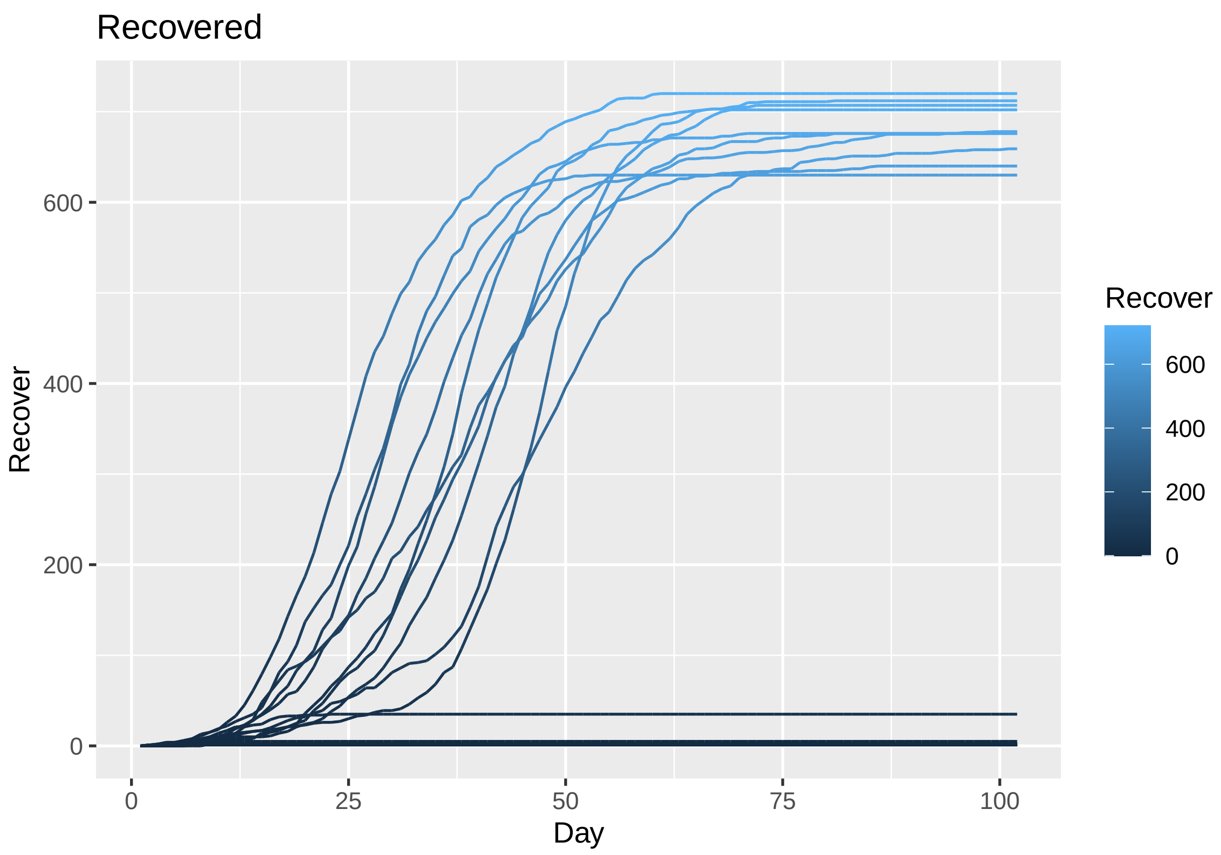
```
for (i in 1:20){
  df = SIR(0.0005, 0.3, 1000, 100, 1)
  df = df %>% mutate(SIM = i)
  if (i == 1){
    new = df
  } else {
    new = rbind(new, df)
  }
}

ggplot(new, aes(x=Day))+geom_line(aes(y=Susceptible, group=SIM, color=Susceptible))+ggtitle("Susceptibl
```



```
ggplot(new, aes(x=Day))+geom_line(aes(y=Infected, group=SIM, color=Infected))+ggtitle("Infected")
```



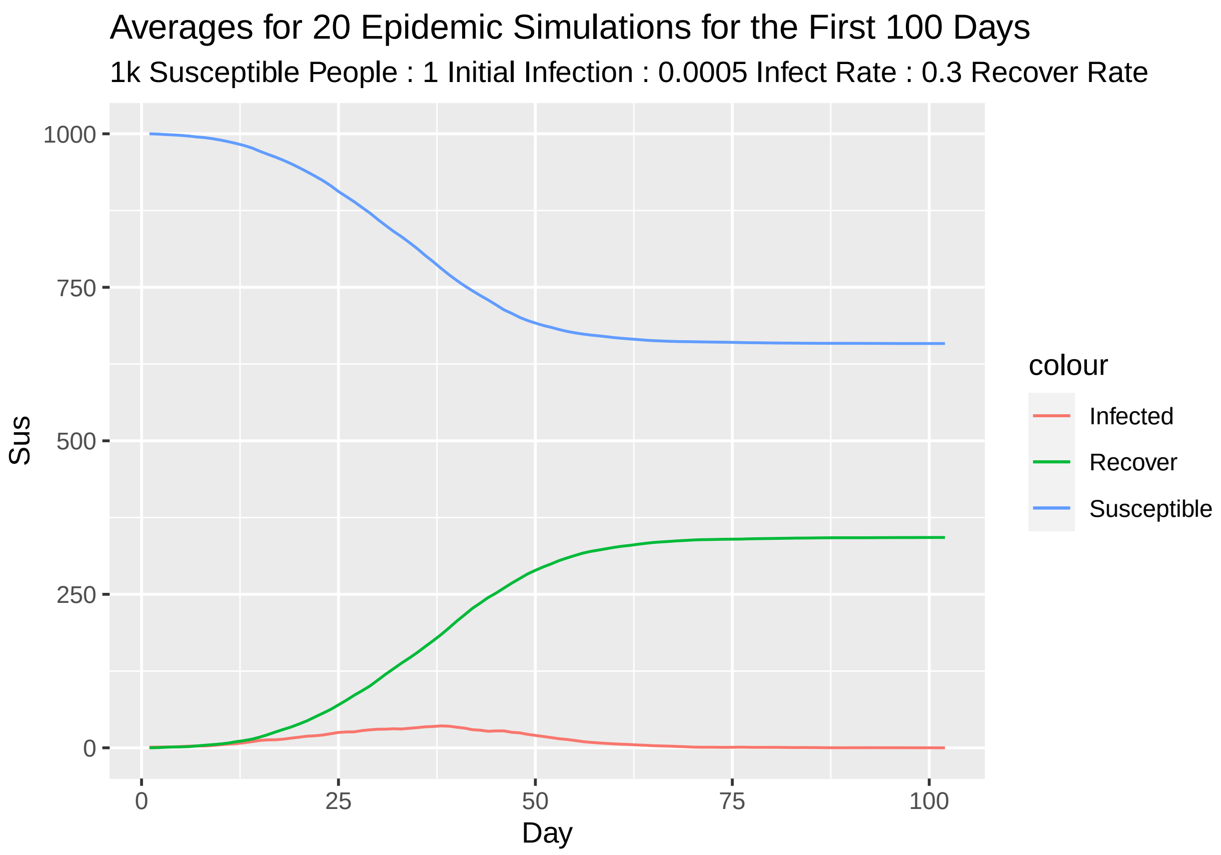
```
ggplot(new, aes(x=Day))+geom_line(aes(y=Recover, group=SIM, color=Recover))+ggtitle("Recovered")
```



The 3 graphs above show the result of every simulation with each graph individually representing Susceptible, Infected and Recovered

Below is a graph the represents the average values for a specific day for this 20 simualtions

```
averages = new %>% group_by(Day) %>% summarise(Sus = mean(Susceptible), Infect = mean(Infected), rec = 
ggplot(averages)+
  geom_line(aes(x = Day, y = Sus, color="Susceptible"))+
  geom_line(aes(x = Day, y = Infect, color="Infected"))+
  geom_line(aes(x = Day, y = rec, color="Recover"))+
  labs(title="Averages for 20 Epidemic Simulations for the First 100 Days", subtitle = "1k Susceptible 
```



Question 3: Decryption

```
# decode.R
start3 = proc.time()
message <- "coincidences in general are great stumbling blocks in the way of that class of thinkers who
set.seed(1)
#  mat <- read.table("ShakeCount.txt",header=F)
mat <- read.table("AustenCount.txt",header=F)
logmat <- log(mat + 1)
# Computes the score of the decoded message using the given code
score <- function(code)
{
    p <- 0
    # For each pair of letters in the decoded message
    # query the transition matrix for the probability of that pair
    for (i in 1:(nchar(message)-1)){
        p <- p + logmat[charIndex(substr(code, i, i)),charIndex(substr(code, i+1, i+1))]
    }
    # return the sum of these probabilities
    p
}
# ascii(char) returns the numerical ascii value for char
ascii <- function(char)
{
    strtoi(charToRaw(char),16L) #get 'raw' ascii value
```

```r
}
# charIndex takes in a character and returns its 'char value'
# defined as a=1, b=2, ..., z=26, space=27
# this matches the array created by read.table
charIndex <- function(char)
{
    aValue <- ascii(char)
    if (aValue == 32)
    {
        # return 27 if a space
        27
    } else
    {
        #ascii sets "a" as 97, so subtract 96
        aValue - 96
    }
}
# Decrypts code according to curFunc
decrypt <- function(code,curFunc)
{
    out <- code
    # for each character in the message, decode it according to the curFunc
    for (i in 1:nchar(message))
    {
        charInd <- charIndex(substr(code,i,i))
        if (charInd < 27)
        {
            # change the ith character to the character determined by the curFunc
            substr(out,i,i) <- rawToChar(as.raw(curFunc[charInd] + 96))
        }
    }
    out
}
# codemess holds the scrambled message
codemess <- decrypt(message,sample(1:26))
# instantiate a map to hold previously computed codes' scores
map <- new.env(hash=T, parent=emptyenv())
# we begin with a basic (a->a, z->z) function for decrypting the codemess
curFunc <- 1:27
# calculate the score for curFunc and store it in the map
oldScore <- score(decrypt(codemess,curFunc))
map[[paste(curFunc, collapse='')]] <- oldScore

df = data.frame(iterations = numeric(), old = numeric(), new = numeric())
# run 10000 iterations of the Metropolis-Hastings algorithm
for (iteration in 1:10000) {
    # sample two letters to swap
    swaps <- sample(1:26,2)
    oldFunc <- curFunc

    # let curFunc be oldFunc but with two letters swapped
    curFunc[swaps[1]] <- oldFunc[swaps[2]]
    curFunc[swaps[2]] <- oldFunc[swaps[1]]
    # if we have already scored this decoding,
    # retrieve score from our map
    if (exists(paste(curFunc, collapse =''), map)){
        newScore <- map[[paste(curFunc, collapse ='')]]
    } else
    # if we have not already scored this decoding,
    # calculate it and store it in the map
    {
        newScore <- score (decrypt(codemess,curFunc))
        map[[paste(curFunc, collapse = '')]] <- newScore
    }

    # decide whether to accept curFunc or to revert to oldFunc
    if (runif(1) > exp(newScore-oldScore))
    {
        curFunc <- oldFunc
    } else
    {
        oldScore <- newScore
    }

    # print out our decryption every 10000 iterations
    if ((iteration %%  10000) == 0)
    {
        print(c(iteration,decrypt(codemess,curFunc)))
    }
    scoreOutputs = c(iteration, round(oldScore), round(newScore))
  df = rbind(df, scoreOutputs)
}
```

```
## [1] "10000"
## [2] "coincidences in general are great stumbling blocks in the way of that class of thinkers who have
```

```
time3 = proc.time() - start3

colnames(df) = c("Iteration", "oldscore", "newscore")
library(ggplot2)
o_plot = ggplot(df, aes(x=Iteration))+geom_line(aes(y=oldscore), color="red", size=1.5)+geom_point(aes(y
```

```
## Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.
## i Please use `linewidth` instead.
```

Differences between the old code and new.

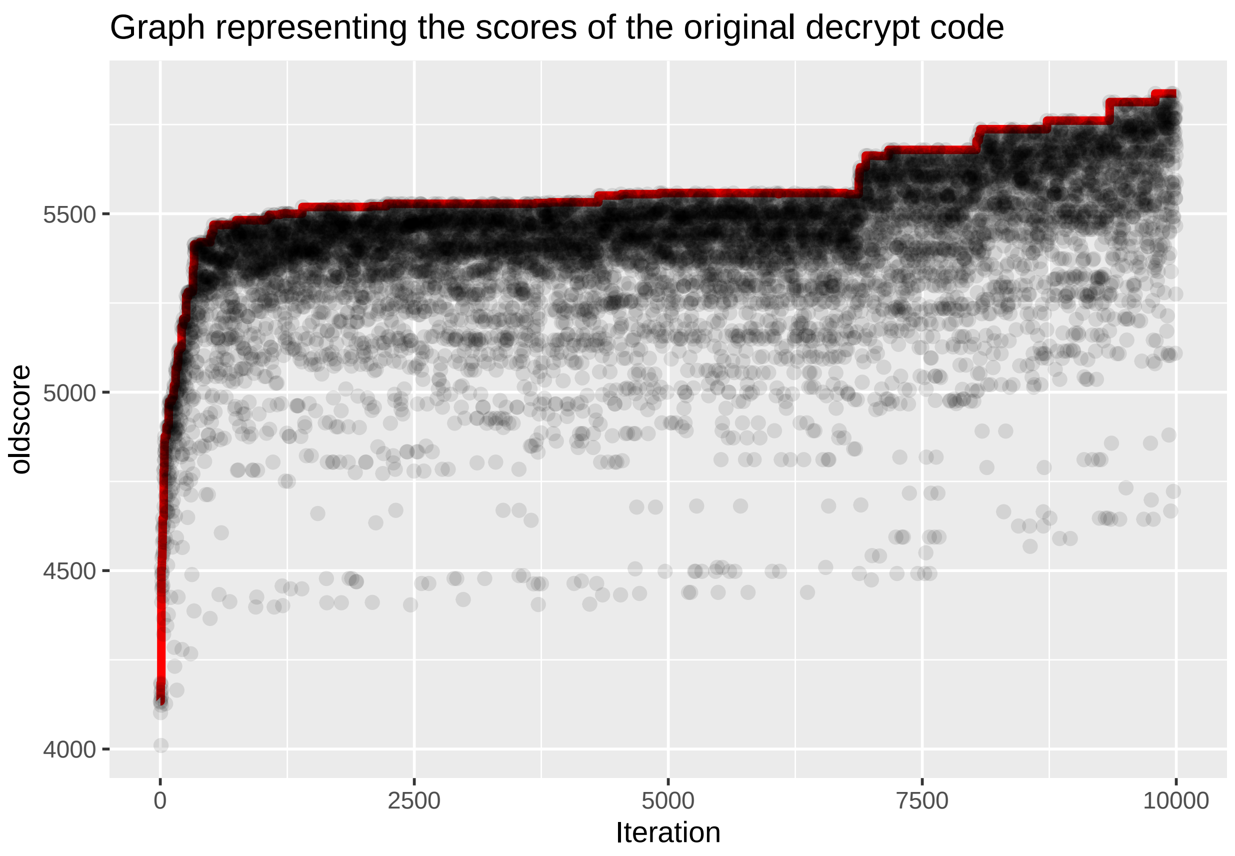
- The first difference comes in the score() function where we use vectorization. The score function currently uses a loop that iterates over every pair of characters in the input code. Instead of doing that, we use vectorization to compute the transition probabilities for all pairs of indices at once using matrix indexing. Vectorization operates on entire arrays or vectors of data at once, rather than processing each element of the array individually. This will make the code more efficient.
- The second difference is that we avoid unnecessary function calls. Specifically the charIndex function is called for every character in the input code. We can avoid unnecessary function calls by precomputing the indices for all possible characters and storing them in a table for lookup.
- These differences help speed up the running time in the script.

```r
# decode.R
start2 = proc.time()
message <- "coincidences in general are great stumbling blocks in the way of that class of thinkers who

set.seed(1)

logmat <- log(read.table("AustenCount.txt",header=F) + 1)


# Computes the score of the decoded message using the given code
score <- function(code) {
  char_indices <- sapply(strsplit(code, "")[[1]], charIndex)
  # sapply()  applies a function to each element of a vector or list and returns the results in a  simp
  char_indices_pairs <- cbind(char_indices[-length(char_indices)], char_indices[-1])
  # cbind() is used to combine vectors, matrices or data frames by columns.
  transition_probs <- logmat[char_indices_pairs]
  score <- sum(transition_probs)
  return(score)
}


# ascii(char) returns the numerical ascii value for char
ascii <- function(char){
  strtoi(charToRaw(char),16L) #get 'raw' ascii value
}


# Precompute indices for all possible characters
char_indices <- c(1:26, 27)
char_values <- c(1:26, 32)

# Define lookup table
char_index_lookup <- setNames(char_indices, letters)
char_index_lookup[" "] <- 27


# Modify charIndex function to use the lookup table
charIndex <- function(char) {
  char_index_lookup[char]
}


# Decrypts code according to curFunc
decrypt <- function(code,curFunc){
  out <- code
  # for each character in the message, decode it according to the curFunc
  for (i in 1:nchar(message))  {
    charInd <- charIndex(substr(code,i,i))
    if (charInd < 27)    {
      # change the ith character to the character determined by the curFunc
      substr(out,i,i) <- rawToChar(as.raw(curFunc[charInd] + 96))
    }
  }
  out
}


# codemess holds the scrambled message
codemess <- decrypt(message,sample(1:26))

# instantiate a map to hold previously computed codes' scores
map <- new.env(hash=T, parent=emptyenv())

# we begin with a basic (a->a, z->z) function for decrypting the codemess
curFunc <- 1:27

# calculate the score for curFunc and store it in the map
oldScore <- score(decrypt(codemess,curFunc))
map[[paste(curFunc, collapse='')]] <- oldScore


df1 = data.frame(iterations = numeric(), old = numeric(), new = numeric())
# run 10000 iterations of the Metropolis-Hastings algorithm
for (iteration in 1:10000) {
  # sample two letters to swap
  swaps <- sample(1:26,2)
  oldFunc <- curFunc

  # let curFunc be oldFunc but with two letters swapped
  curFunc[swaps[1]] <- oldFunc[swaps[2]]
  curFunc[swaps[2]] <- oldFunc[swaps[1]]

  # if we have already scored this decoding,
  # retrieve score from our map
  newCode <- decrypt(codemess, curFunc)
  if (exists(paste(curFunc, collapse=''), map)) {
    newScore <- map[[paste(curFunc, collapse='')]]
  } else {
    newScore <- score(newCode)
    map[[paste(curFunc, collapse='')]] <- newScore
  }

  # decide whether to accept curFunc or to revert to oldFunc
  if (runif(1) > exp(newScore-oldScore))  {
    curFunc <- oldFunc
  } else   {
    oldScore <- newScore
  }

  # print out our decryption every 100 iterations
  if ((iteration %%  10000) == 0)  {
    print(c(iteration,decrypt(codemess,curFunc)))
  }
  scoreOutputs = c(iteration, round(oldScore), round(newScore))
  df1 = rbind(df1, scoreOutputs)
}
```

```
## [1] "10000"
## [2] "coincidences in general are great stumbling blocks in the way of that class of thinkers who have
```

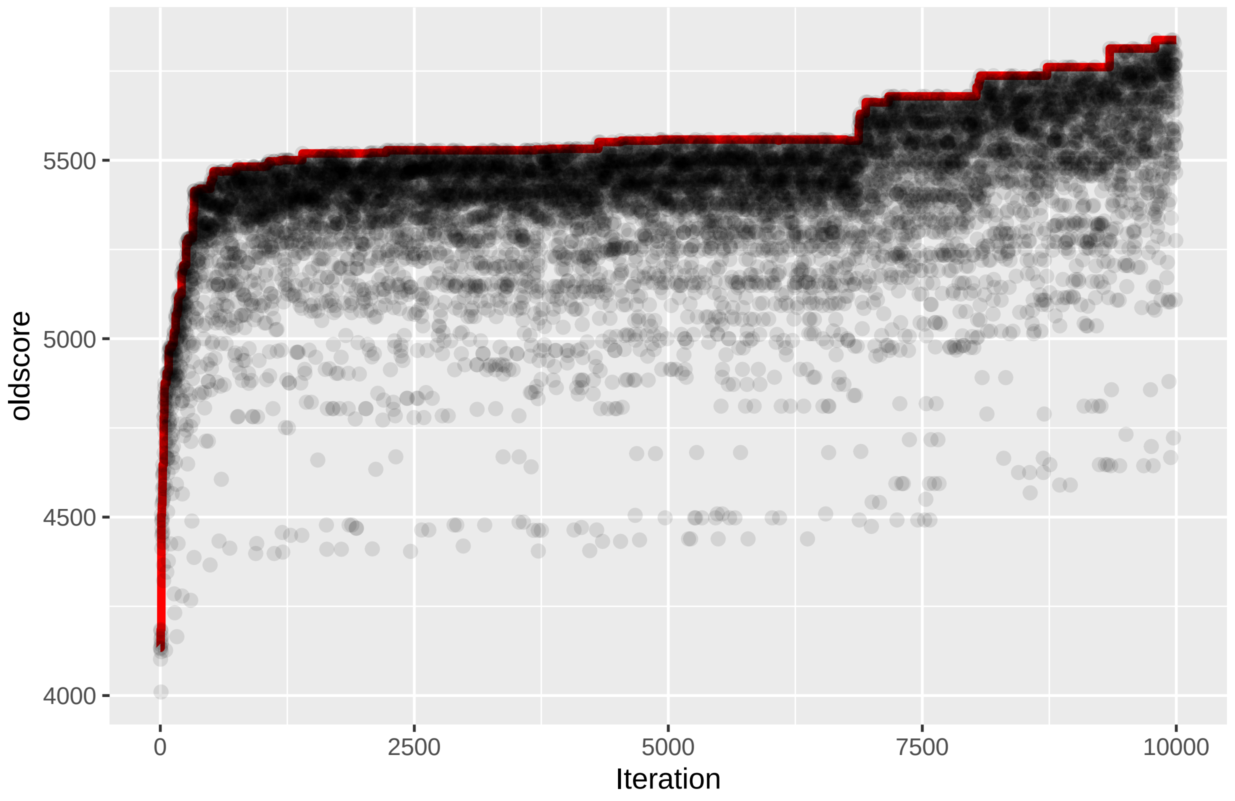
```
time2 = proc.time() - start2

colnames(df1) = c("Iteration", "oldscore", "newscore")
library(ggplot2)
n_plot = ggplot(df1, aes(x=Iteration))+geom_line(aes(y=oldscore), color="red", size=1.5)+geom_point(aes
```

```
o_plot
```



```
n_plot
```

Graph representing the scores of the new decrypt code



As you can see, they reach the best score at the same exact time.

Now for looking at the performance

```
cat("Old decode time:",time3['elapsed'], "\n")
```

```
## Old decode time: 78.309
```

```
cat("New decode time:",time2['elapsed'])
```

```
## New decode time: 42.845
```

- By changing two of the functions in the original code, we can cut down the time it takes to complete the code chunk by 1/2 the amount of time or even more!

Question 4

Functions and Classes

Some functions and classes are imported from a separate file.

```
source('functions.R')

# function to simulate a random vs. random game and record the outcome in a dataframe
randomSimulation = function(numSims, boardSize = 3, board = NA, df, seed = 0, colName = NA, player = 1,
  # set seed
  set.seed(seed)

  #simulate TTT game numSims times
  winners<- replicate(numSims, randtictactoe(boardSize, board, player = player))
  #table(winners)

  #create a dataframe from the winner results for plotting & tables
  if (colName == 'Board Size'){
    if (hist == T) hist(winners, main = paste0('Results of 1,000 ',
                                             boardSize,'x', boardSize,' Tic-Tac-Toe Games'))
    df2 <- data.frame(Winner = winners,
                  'Board Size' = rep(paste0(boardSize,'x', boardSize), numSims))
  }
  else if (colName == 'First Move' | colName == 'Second Move'){
    if (colName == 'First Move'){
      df2 <- data.frame(Winner = winners, 'Move' = rep(which(board==1), numSims))}
    else{
      df2 <- data.frame(Winner = winners, 'First Move' = rep(which(board==1)),
                      'Second Move' = rep(which(board==-1), numSims))}

  }
  else {
    df2 <- data.frame(Winner = winners, numSims = rep(as.character(colName),numSims))
  }

  # merge and return dataframes
  df <- rbind(df, df2)
  return(df)
}

# function to simulate MCTS vs. Random player game
MCTSvsRandomSim = function(numSims, first = c('MCTS', 'random'), iterations = 500, board = NA, player =
  if (length(which(is.na(board)))==1) board = array(rep(NA, 3^2),
                                                    dim = c(3, 3))

  if (first == 'MCTS') winner = replicate(numSims,
                                         MCTSvsRandom(iterations = iterations, board = board, player =
  else winner = replicate(numSims,
                          RandomvsMCTS(iterations = iterations, board = board, player = player))
  return(winner)
}
```

Constant Values

```
n = 1e3 # number of simulations
```

Introduction

Estimating the probability of a player winning a game can be a complex and difficult problem as the number of legal moves increase. In recent years, use of simulation and the Monte Carlo method have proven useful in the field of game theory and determining the optimal move or choice, given a game board or state. Since the success of the artificial intelligence (AI) game-bots AlphaGo and its successors, the Monte Carlo Tree Search (MCTS) has become an integral tool for developing game AIs. Go has 2.1×10^{170} legal moves which makes the construction of a complete game tree computationally impossible, and this is where MCTS excels. MCTS does not require the construction of a complete game tree to determine the optimal move; instead, MCTS uses repeated simulation at leaf nodes and backpropogation of results to determine the best move. For simplicity and demonstrability of the effectiveness of the Monte Carlo Tree Search method, we implement a MCTS for tic-tac-toe. Our research questions are:

- How does the size of the board effect the probability of winning?
- For each player, what are the probabilities of winning…
  - If each player moves randomly?
  - If one player strategizes and the other plays randomly?
- For which first move does Player 1 have the highest probability of winning?
- Given Player 1's first Move, for which second move does Player 2 have the highest probability of winning?

Monte Carlo Tree Search (MCTS)

The Monte Carlo Tree Search Algorithm is a tree construction and search algorithm consisting of 4 stages: selection, expansion, simulation, backpropogation (see [1]). - Selection: Beginnnig at the root, we traverse the tree, selecting children nodes until an expandable node is reached. A node is *expandable* if it represents a non-terminal state and has an unvisited child state. - Expansion: One or more child nodes are added to the tree, according the open moves. - Simulation: From a new node, we simulate a complete play-out of the game and determine the winner. - Back-propagation: The simulation result is back propagated up the tree through the selected nodes to the root.

Once this occurs for a set number of iterations, the MCTS algorithm selects one of the root's child nodes as the optimal move.

When the selection of the nodes is based on the Upper Confidence Bound, the MCTS is alternatively called an Upper Confidence Bound Tree (UCT). The Upper Confidence Boaund (UCB) formula is as follows.

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$$

The number of wins at a node is denoted w_i, the number of node visits are n_i and the number of visits at the parent's node is N_i. The exploration parameter c is st by the user, but the value $\sqrt{2}$ has been proven to satisfy the Hoeffding inequality with rewards in the range $[0, 1]$ (see [1]). Most importantly, we note that using the UCB as the selection policy allows MCTS to converge to the minimax tree and is thus optimal (see [1]). That is, Kocis and Szepesvari have proven that the failure probability at the root of the tree converges to 0 at a polynomial rate as the number of games simulated grows to infinity (see [1]).

The Monte Carlo Tree Search has several theoretical and computational properties that make it a popular choice for game-bot development. First, we discuss the advantages. MCTS is aheuristic, anytime, and asymmetric. It is aheuristic, which means that there is no need for domain-specific knowledge or an evaluation function to determine the optimal move. It is anytime, since the results of every simulation get back propagated to the root immediately, and this means that the information stored at the root can be returned at anytime. Lastly, it s asymmetric, meaning that the algorithm favors more promising nodes and the complete game tree (or all nodes at a given depth) need not be constructed.

Some of the disadvantages of MCTSare that the outcome of the algorithm is dependent on the number of iterations a user chooses to run and the algorithm performs poorly with games that have a lot of trap states. A trap state is a state that leads to a loss within a small number of moves, and with some games, such as Chess, MCTS can perform worse than other algorithms, such as minimax. Additionally, the performance of the MCTS algorithm is based on the number of simulations of the game that are run, and of course, this is dependent on a machine's architecture.

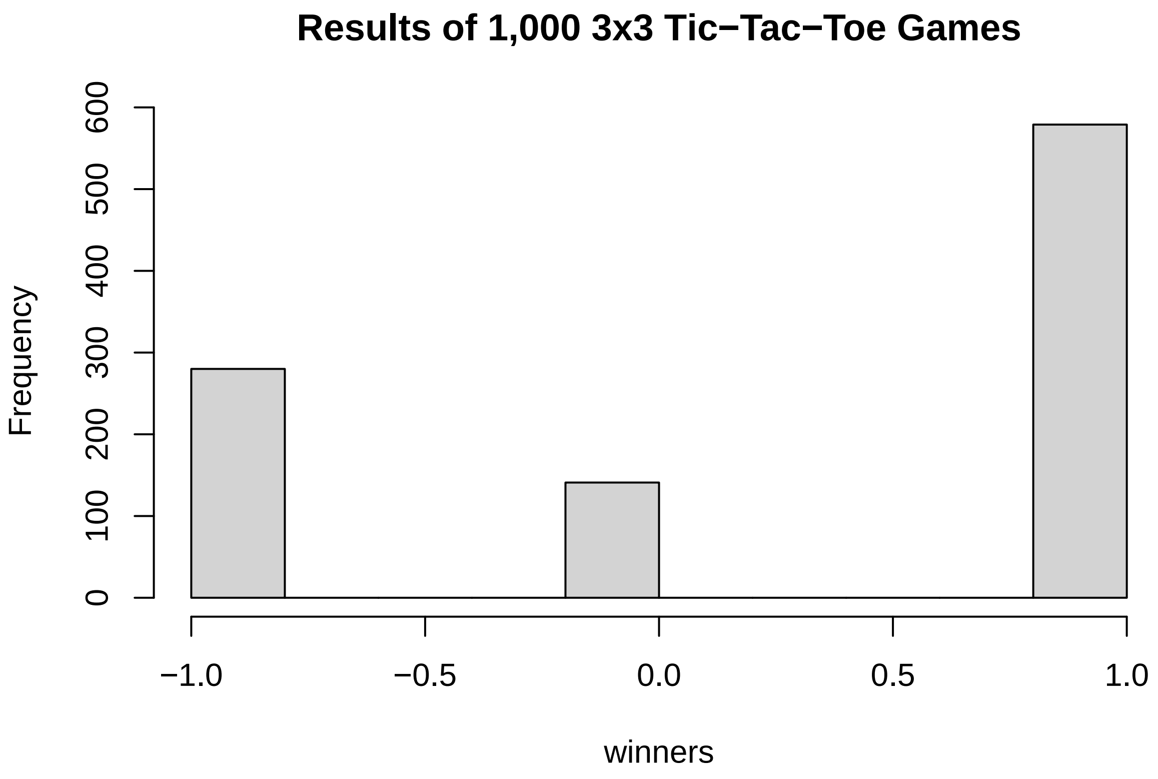
Random Player vs. Random Player:

For these simulations, we play 1,000 tc-tac-toe games and assume that Player 1 moves first.

How does the size of the board effect the probability of winning?

First, we simulate the data using the functions we have written. We choose to run 1,000 simulations for boards with dimensions 3×3, 4×4, 5×5, and 10×10.

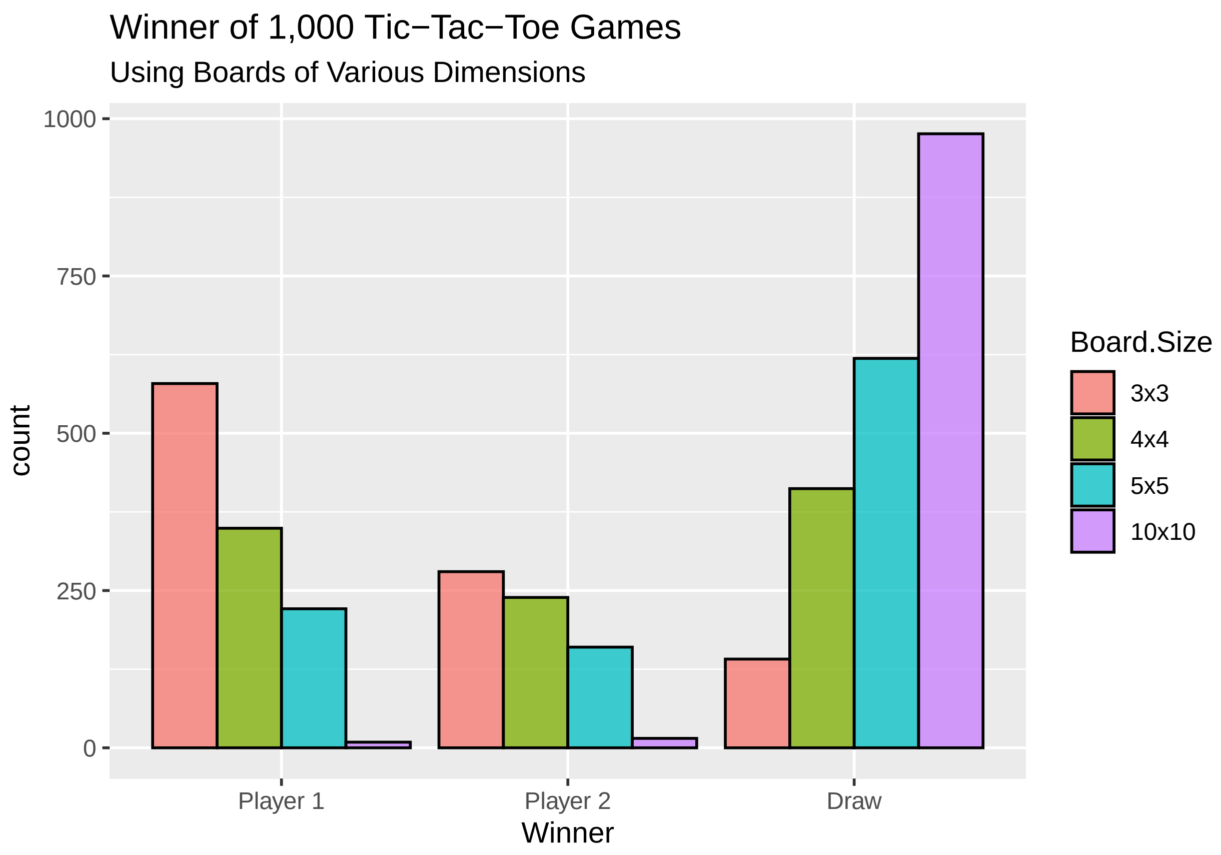
```r
#create data frame to store winner results
df <- data.frame(matrix(ncol = 2, nrow = 0))
colnames(df) <- c('Winner', 'Board Size')

# update the the DF with the winner results of 1,000 games on varying board sizes.
df <- randomSimulation(n, 3, df = df, seed = 0, colName = 'Board Size', hist =T)
```



```r
df <- randomSimulation(n, 4, df = df, seed = 1, colName = 'Board Size')
df <- randomSimulation(n, 5, df = df, seed = 2, colName = 'Board Size')
df <- randomSimulation(n, 10, df = df, seed = 3, colName = 'Board Size')
```

We can visualize the number of times each player wins using a bar chart.

```r
# make each of the variables a factor
boards <- c('3x3','4x4','5x5','10x10')
df$Board.Size <- factor(df$Board.Size, levels = boards)
df$Winner <- factor(df$Winner, levels = c(1, -1, 0),
                    labels = c('Player 1', 'Player 2', 'Draw'))

# plot a bar chart
ggplot(df, aes(Winner)) +
  geom_bar(aes(fill = Board.Size), position = 'dodge', color = 'black',
           alpha = 0.75) +
  ggtitle('Winner of 1,000 Tic-Tac-Toe Games',
          subtitle = 'Using Boards of Various Dimensions')
```



We can also calculate the probabilities of each outcome on each board.

```r
# create new DF to store the previous results and their probabilities
results <- data.frame(matrix(ncol = 3, nrow = 0))
colnames(results) <- c('Winner', 'Board Size', 'Probability')

# update the new DF with the probabilities of each player winning on each board size
for (i in boards){
  df_board <- df %>% filter(Board.Size == i)
  p1 <- length(df_board$Winner[df_board$Winner == 'Player 1']) / n; p1
  p2 <- length(df_board$Winner[df_board$Winner == 'Player 2']) / n; p2
  draw <- length(df_board$Winner[df_board$Winner == 'Draw']) / n; draw

  res <- data.frame(Winner = c('Player 1', 'Player 2', 'Draw'),
               'Board Size' = i, Probability = c(p1,p2, draw))
  results <- rbind(results, res)
}

# create and output a pivot table of the results
pt <- pivot_wider(results, id_cols = Board.Size, names_from = Winner, values_from = Probability);
knitr::kable(pt, align = 'cccc',
             caption = 'Probabilities of Winning Based on Board Size') %>%
  kable_styling(full_width = F)
```

From both the bar chart and the table of probabilities, we can see that on a standard 3×3 tic-tac-toe board, the probability of Player 1 winning is nearly 57.9%, while the probability of Player 2 winning is almost half of that (28%). As the game space increases, the probability of either player winning decreases, while the probability of ending in a draw increases. With a 10×10 board, the probability of either Player winning is

Table 1: Probabilities of Winning Based on Board Size

Board.Size	Player 1	Player 2	Draw
3x3	0.579	0.280	0.141
4x4	0.349	0.239	0.412
5x5	0.221	0.160	0.619
10x10	0.009	0.015	0.976

less than 3%, that is, draws are the most common outcome; however, Player 2, actually has a slightly larger probability of winning than Player 1.

For which first move does Player 1 have the highest probability of winning?

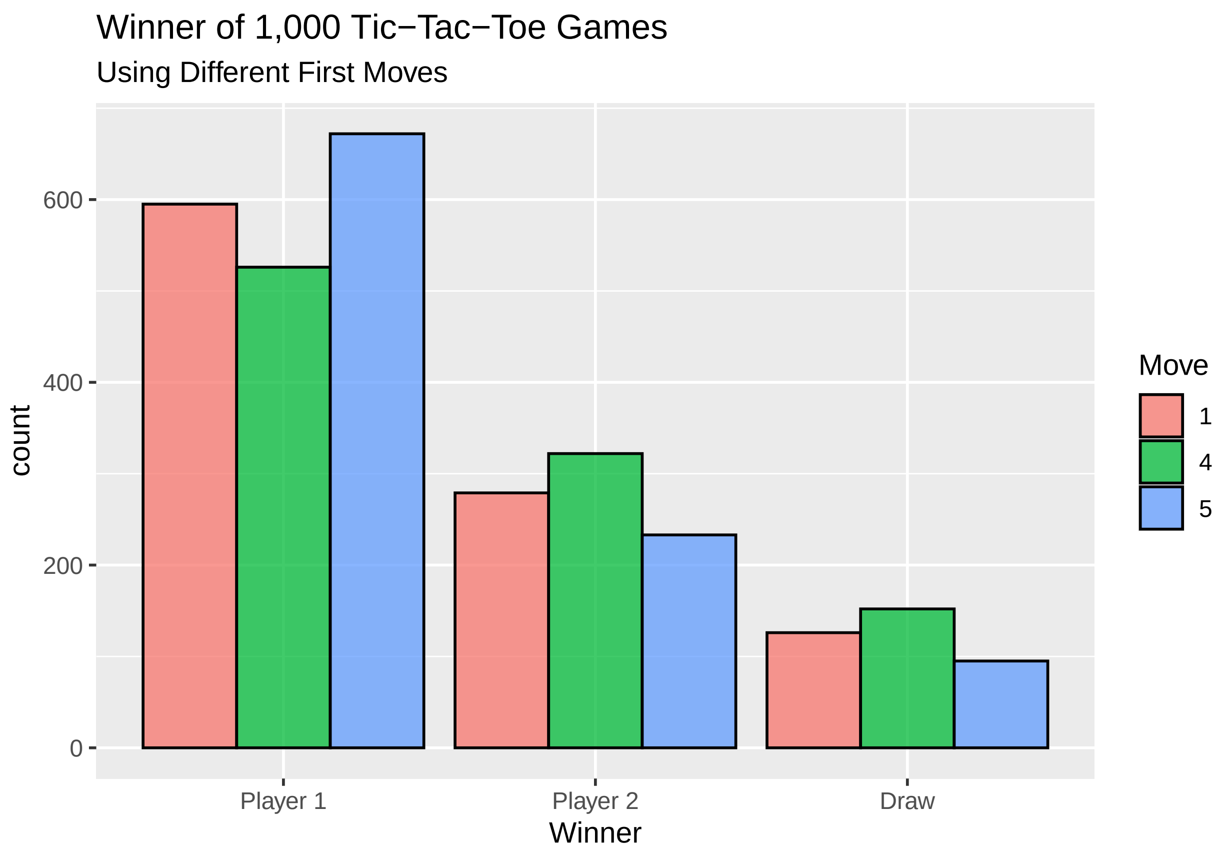
For the rest of the questions, we only look at the 3×3 tic-tac-toe board, which we implement using a 3×3 array. `R` indexing in a 3×3 array is as follows.

1	4	7
2	5	8
3	6	9

There are 3 general first moves, those being, 1, 4, and 5. All of the other first moves can be written as a rotation or reflection of these three (see [5]). Thus, given each of these starting boards, we can simulate the outcomes of 1,000 games for each board and record the outcomes.

```r
# create the boards to be simulated
board1 <- array(c(1, rep(NA, 8)), dim = c(3,3))
board4 <- array(c(rep(NA, 3), 1, rep(NA, 4)), dim = c(3,3))
board5 <- array(c(rep(NA,4),1,rep(NA,4)), dim = c(3,3))

# create new DF to store the results
df1 <- data.frame(matrix(ncol = 2, nrow = 0))
colnames(df1) <- c('Winner', 'Move')

# update the DF with the results
df1 <- randomSimulation(n, 3, board1, df = df1, seed = 0, 'First Move', -1)
df1 <- randomSimulation(n, 3, board4, df = df1, seed = 1, 'First Move', -1)
df1 <- randomSimulation(n, 3, board5, df = df1, seed = 2, 'First Move', -1)
```

```r
# make each variable in the DF a factor
df1$Move <- as.factor(df1$Move)
df1$Winner <- factor(df1$Winner, levels = c(1, -1, 0),
                     labels = c('Player 1', 'Player 2', 'Draw'))

# plot a barchart
ggplot(df1, aes(Winner)) +
  geom_bar(aes(fill = Move), position = 'dodge', color = 'black',
           alpha = 0.75) +
  ggtitle('Winner of 1,000 Tic-Tac-Toe Games',
          subtitle = 'Using Different First Moves')
```



```r
# similar to before create a new DF to store probabilities
results <- data.frame(matrix(ncol = 3, nrow = 0))
colnames(results) <- c('Winner', 'First Move', 'Probability')

# calculate and update the new DF with the probabilities
for (i in c(1,4,5)){
  df_fmove <- df1 %>% filter(Move == i)
  p1 <- length(df_fmove$Winner[df_fmove$Winner == 'Player 1']) / n; p1
  p2 <- length(df_fmove$Winner[df_fmove$Winner == 'Player 2']) / n; p2
  draw <- length(df_fmove$Winner[df_fmove$Winner == 'Draw']) / n; draw

  res <- data.frame(Winner = c('Player 1', 'Player 2', 'Draw'),
                'First Move' = i, Probability = c(p1,p2, draw))
  results <- rbind(results, res)
}

# create and output a pivot table of the probabilities
pt <- pivot_wider(results, id_cols = First.Move, names_from = Winner, values_from = Probability);
knitr::kable(pt, align = 'cccc',
              caption = "Probabilities of Winning Based on Player 1's First Move") %>%
  kable_styling(full_width = F)
```

To answer our research question, box 5, the center box, is the optimal first move for Player 1 to have the highest probability of wining. The probability of Player 1 winning with the first move in box 5 is higher than the probability of Player 1 winning in general on a 3×3 board while the probability of Player 1 winning given their first move is in a corner box (box 1) is approximately equal to the probability of Player 1 winning. These results were expected since picking the center or corner box is a common strategy employed to win

Table 2: Probabilities of Winning Based on Player 1's First Move

First.Move	Player 1	Player 2	Draw
1	0.595	0.279	0.126
4	0.526	0.322	0.152
5	0.672	0.233	0.095

tic-tac-toe. Lastly, we note that if Player 1 first picks a middle box that is *not* the center most box, Player 1's probability of winning is the lowest.

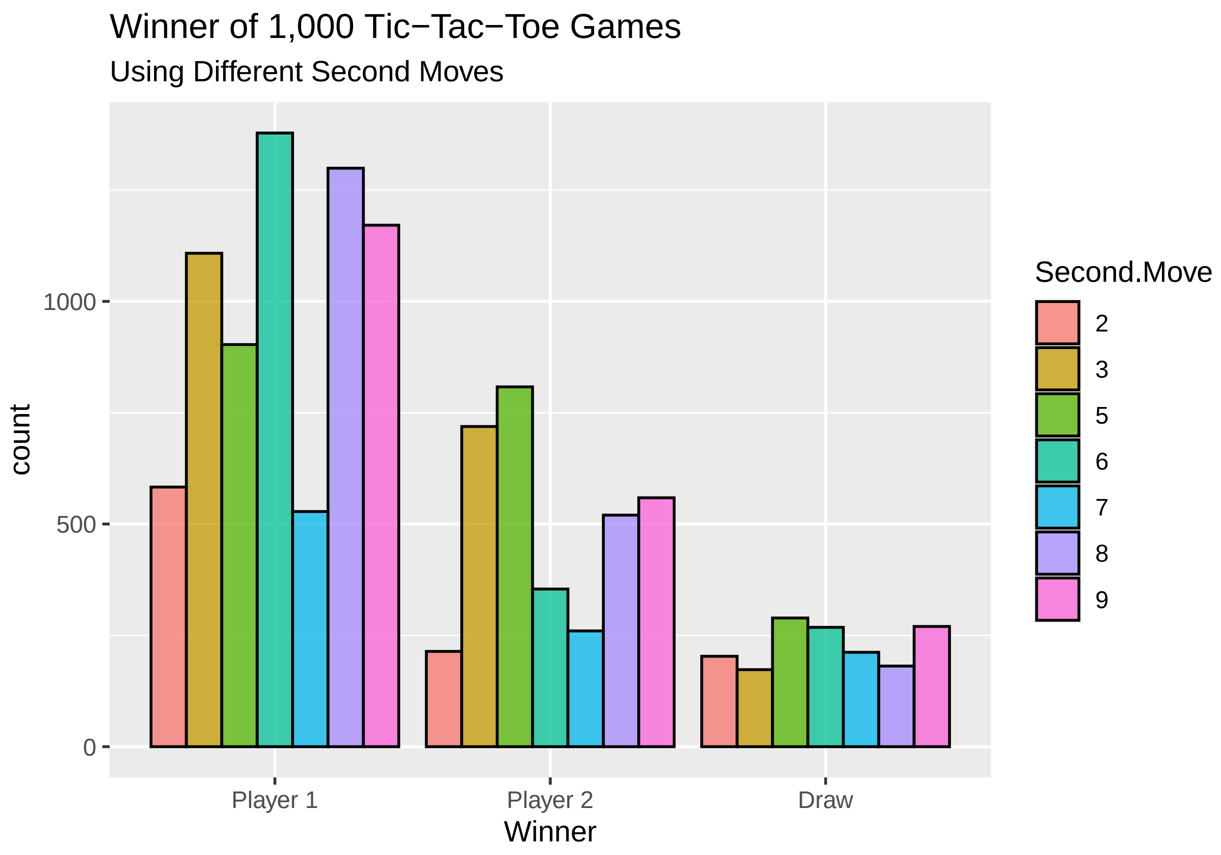
NOTE: Upon writing the report, we noticed an error in our code that was used to generate the figures for the presentation. We have fixed the code and updated the final presentation. Prior, given a board with Player 1's first move, Player one would make an additional second move before alternating to Player 2. We note that either way, box 5 is the optimal first move for Player 1.

Given Player 1's First Move, for which second move does Player 2 have the highest probability of winning?

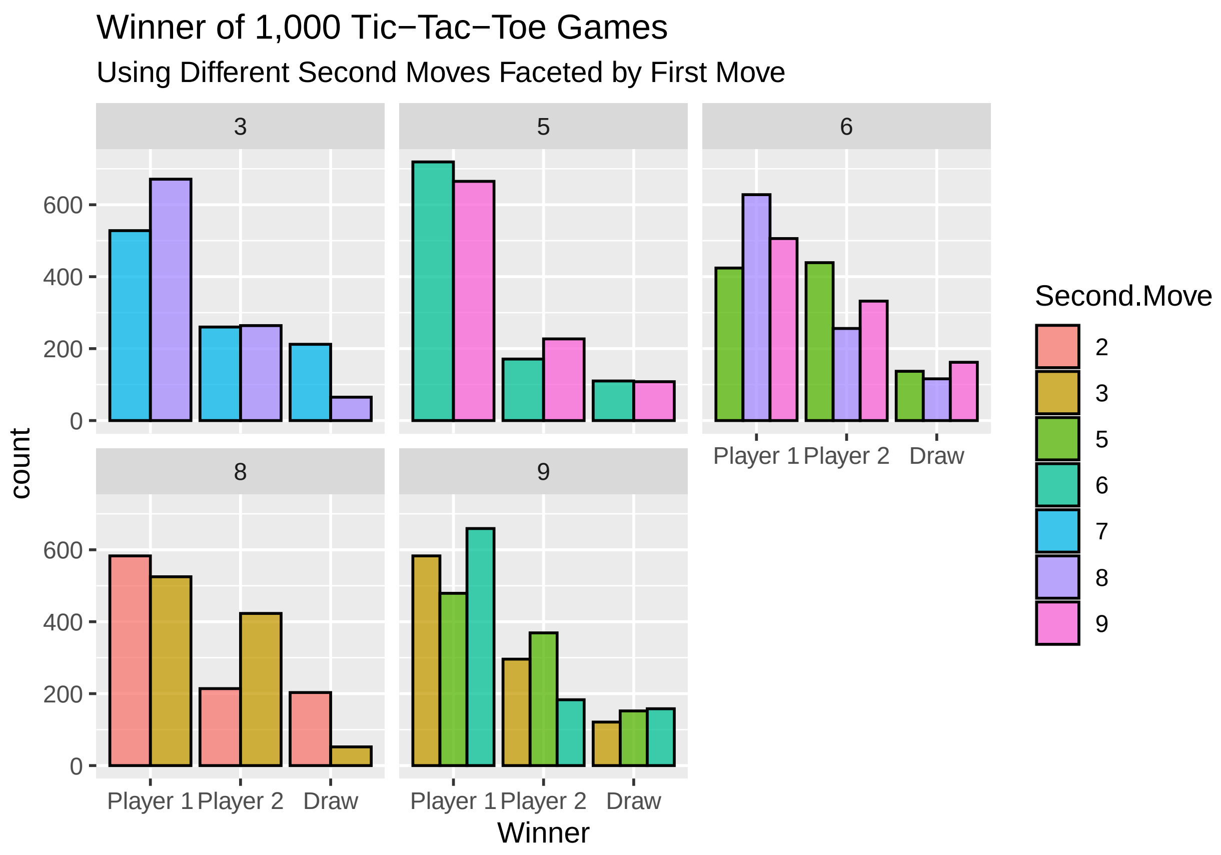
Once more, we can utilize the board's symmetry to reduce the number of boards we have to simulate to 12 different boards (see [5]).

```r
# create boards with different first 2 moves
board1 <- array(c(rep(NA,2), 1, rep(NA, 3), -1, rep(NA, 2)), dim = c(3,3))
board2 <- array(c(rep(NA,2), 1, rep(NA, 4), -1, NA), dim = c(3,3))
board3 <- array(c(rep(NA,2),-1,rep(NA,5), 1), dim = c(3,3))
board4 <- array(c(rep(NA,4),-1,rep(NA,3), 1), dim = c(3,3))
board5 <- array(c(rep(NA,5),-1,rep(NA,2), 1), dim = c(3,3))
board6 <- array(c(rep(NA,2),-1,rep(NA,4), 1, NA), dim = c(3,3))
board7 <- array(c(NA,-1,rep(NA,5), 1, NA), dim = c(3,3))
board8 <- array(c(rep(NA,5),1,NA, -1, NA), dim = c(3,3))
board9 <- array(c(rep(NA,4),-1, 1, rep(NA,3)), dim = c(3,3))
board10 <- array(c(rep(NA,5),1,rep(NA,2), -1), dim = c(3,3))
board11 <- array(c(rep(NA,4),1,rep(NA,3), -1), dim = c(3,3))
board12 <- array(c(rep(NA,4),1, -1, rep(NA, 3)), dim = c(3,3))

# create new DF to store the winning results
df1 <- data.frame(matrix(ncol = 3, nrow = 0))
colnames(df1) <- c('Winner', 'First Move', 'Second Move')

# update the DF with the results of simulations
df1 <- randomSimulation(n, 3, board1, df = df1, seed = 0, 'Second Move')
df1 <- randomSimulation(n, 3, board2, df = df1, seed = 1, 'Second Move')
df1 <- randomSimulation(n, 3, board3, df = df1, seed = 2, 'Second Move')
df1 <- randomSimulation(n, 3, board4, df = df1, seed = 3, 'Second Move')
df1 <- randomSimulation(n, 3, board5, df = df1, seed = 4, 'Second Move')
df1 <- randomSimulation(n, 3, board6, df = df1, seed = 5, 'Second Move')
df1 <- randomSimulation(n, 3, board7, df = df1, seed = 6, 'Second Move')
df1 <- randomSimulation(n, 3, board8, df = df1, seed = 7, 'Second Move')
df1 <- randomSimulation(n, 3, board9, df = df1, seed = 8, 'Second Move')
df1 <- randomSimulation(n, 3, board10, df = df1, seed = 9, 'Second Move')
df1 <- randomSimulation(n, 3, board11, df = df1, seed = 10, 'Second Move')
df1 <- randomSimulation(n, 3, board12, df = df1, seed = 11, 'Second Move')

# make each variable factors
```

```r
df1$First.Move <- as.factor(df1$First.Move)
df1$Second.Move <- as.factor(df1$Second.Move)
df1$Winner <- factor(df1$Winner, levels = c(1, -1, 0),
                    labels = c('Player 1', 'Player 2', 'Draw'))

# plot barplot of winners based on Player 2's second move
ggplot(df1, aes(Winner)) +
  geom_bar(aes(fill = Second.Move), position = 'dodge', color = 'black',
           alpha = 0.75) +
  ggtitle('Winner of 1,000 Tic-Tac-Toe Games',
          subtitle = 'Using Different Second Moves')
```



```r
# plot the barplots faceted by Player 1's First Move
ggplot(df1, aes(Winner)) +
  geom_bar(aes(fill = Second.Move), position = 'dodge', color = 'black',
           alpha = 0.75) + facet_wrap(~First.Move) +
  ggtitle('Winner of 1,000 Tic-Tac-Toe Games',
          subtitle = 'Using Different Second Moves Faceted by First Move')
```



```r
# create new DF to store the results
results <- data.frame(matrix(ncol = 4, nrow = 0))
colnames(results) <- c('Winner', 'Second Move', 'Probability')

# calculate probabilities and update the DF
for (i in 1:9){
  df_fmove <- df1 %>% filter(First.Move==i)
  #print(dim(df_fmove))
  for (j in 1:9){
  df_smove <- df_fmove %>% filter(Second.Move == j)
  p1 <- length(df_smove$Winner[df_smove$Winner == 'Player 1']) / dim(df_smove)[1];
  p2 <- length(df_smove$Winner[df_smove$Winner == 'Player 2']) / dim(df_smove)[1];
  draw <- length(df_smove$Winner[df_smove$Winner == 'Draw']) / dim(df_smove)[1];

  res <- data.frame(Winner = c('Player 1', 'Player 2', 'Draw'), 'First Move' = i,
                 'Second Move' = j, Probability = c(p1,p2, draw))
  results <- rbind(results, res)
  }
}

# create and output a pivot table
pt <- pivot_wider(results, id_cols = c(First.Move,Second.Move), names_from = Winner, values_from = Proba
pt <- na.omit(pt);
knitr::kable(pt, align = 'cccc') %>% kable_styling(full_width = F)
```

The faceted bar plot is easiest to read to answer this question; however, we can also select Player 2's optimal move using the pivot table of probabilities of each outcome. We summarize the optimal move for Player 2

First.Move	Second.Move	Player 1	Player 2	Draw
3	7	0.528	0.260	0.212
3	8	0.671	0.264	0.065
5	6	0.719	0.171	0.110
5	9	0.665	0.227	0.108
6	5	0.424	0.439	0.137
6	8	0.628	0.256	0.116
6	9	0.506	0.332	0.162
8	2	0.583	0.214	0.203
8	3	0.525	0.423	0.052
9	3	0.583	0.296	0.121
9	5	0.479	0.369	0.152
9	6	0.659	0.183	0.158

given Player 1's first move below.

Player 1's First Move	Player 2's Optimal Move
3	8
5	9
6	5
8	3
9	5

We note that despite these being Player 2's optimal moves, Player 1 still wins many of the games. The only time Player 2 has an approximately equal probability of winning to Player 1 is when Player 1's first move is 6 and Player 2's second move is the center box 5. Given this set up, Player 1's probability of winning is 42.4% and Player 2's is 43.9%.

Monte Carlo Search Tree

MCTS vs. Random Simulation:

Comparing MCTS with a different number of iterations for each move

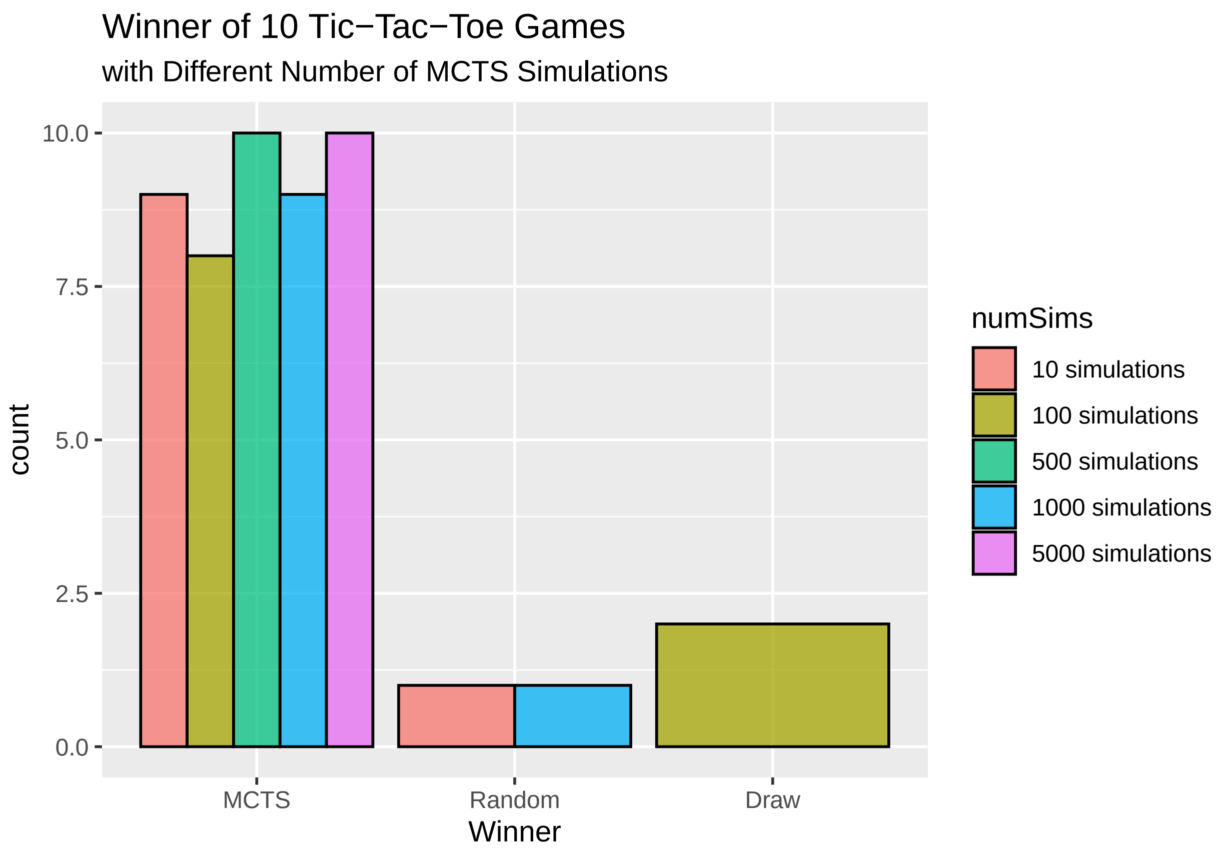
The number of iterations to run the MCTS algorithm to determine the next optimal move is a parameter that is normally selected depending on the amount of computation power and memory a system has. In this section, we compare different numbers of iterations to estimate how many iterations of MCTS are required for the MCTS player to win or tie all of the games. The number of games we simulate from now on is also dependent on how many iterations of the MCTS algorithm we decide to run. Due to computational limits, the higher the number of iterations of MCTS run per move, the lower the number of total games that can be simulated and vice versa. These simulations are run such that the first player is the MCTS player.

```r
#set seed for reproducibility
set.seed(0)

start.time <- Sys.time()
# simulate 10 games for differing numbers of iterations
winner10 = MCTSvsRandomSim(numSims =10, first = 'MCTS', iterations = 10)
winner100 = MCTSvsRandomSim(numSims =10, first = 'MCTS', iterations = 100)
winner500 = MCTSvsRandomSim(numSims =10, first = 'MCTS', iterations = 500)
winner1000= MCTSvsRandomSim(numSims =10, first = 'MCTS', iterations = 1000)
winner5000= MCTSvsRandomSim(numSims =10, first = 'MCTS', iterations = 5000)
#winner200 = MCTSvsRandomSim(numSims =1000, first = 'MCTS', iterations = 200)
end.time <- Sys.time()
```

```r
time.taken <- end.time - start.time; time.taken
```

```
## Time difference of 1.019231 mins
```

```r
# create DF so that each game is a row with a column containing number of iterations
dfIter = data.frame(winner10, winner100, winner500, winner1000, winner5000)
winnerIterations = pivot_longer(dfIter, cols = colnames(dfIter))
colnames(winnerIterations) <- c('numSims','Winner')
winnerIterations$Winner <- factor(winnerIterations$Winner,
                                  levels = c(1, -1, 0),
                                  labels = c('MCTS', 'Random', 'Draw'))
winnerIterations$numSims =
  factor(winnerIterations$numSims, levels = c('winner10', 'winner100',
                                              'winner500', 'winner1000', 'winner5000'),
                               labels = c('10 simulations', '100 simulations',
                                          '500 simulations', '1000 simulations',
                                          '5000 simulations'))

# create a bar plot of the results
ggplot(winnerIterations) +
  geom_bar(aes(Winner, fill = numSims), position = 'dodge', color = 'black',
           alpha = 0.75) +
  ggtitle('Winner of 10 Tic-Tac-Toe Games',
          subtitle = 'with Different Number of MCTS Simulations')
```



Based on the bar chart, we decide to try to run 500 iterations of MCTS per move since this is the lowest number of iterations such that the MCTS player completely wins all 10 games.

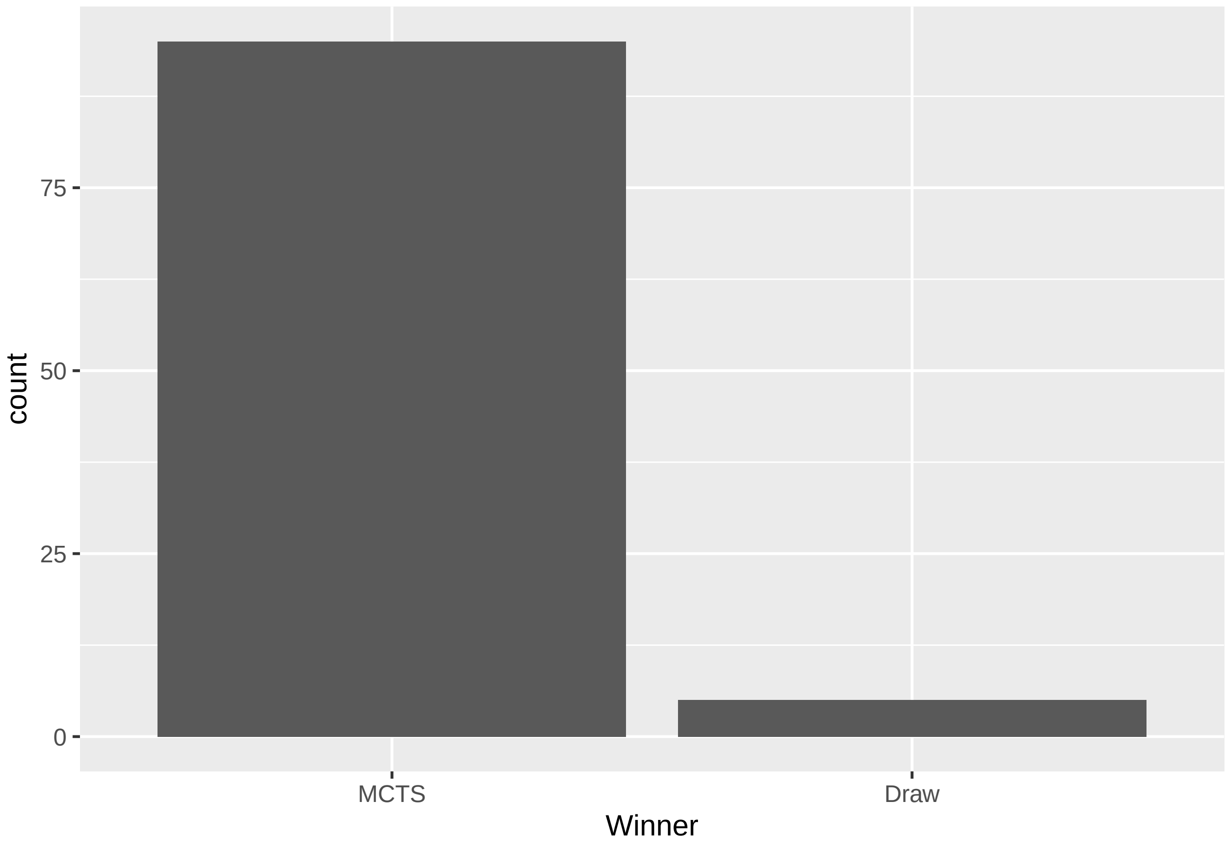
Test how many games and iterations to simulate

In this subsection, given that we want to run 500 iterations of MCTS per move, we experiment with how many tic-tac-toe games to simulate.

```r
set.seed(0)
start.time <- Sys.time()
winnerParam = MCTSvsRandomSim(numSims =100, first = 'MCTS', iterations = 500)
end.time <- Sys.time()
time.taken <- end.time - start.time; time.taken
```

```
## Time difference of 59.03402 secs
```

```r
dftest = data.frame(winnerParam)
dftest$Winner = factor(dftest$winnerParam, levels = c(1,-1,0),
                       labels = c('MCTS','Random', 'Draw'))
ggplot(dftest) + geom_bar(aes(Winner))
```



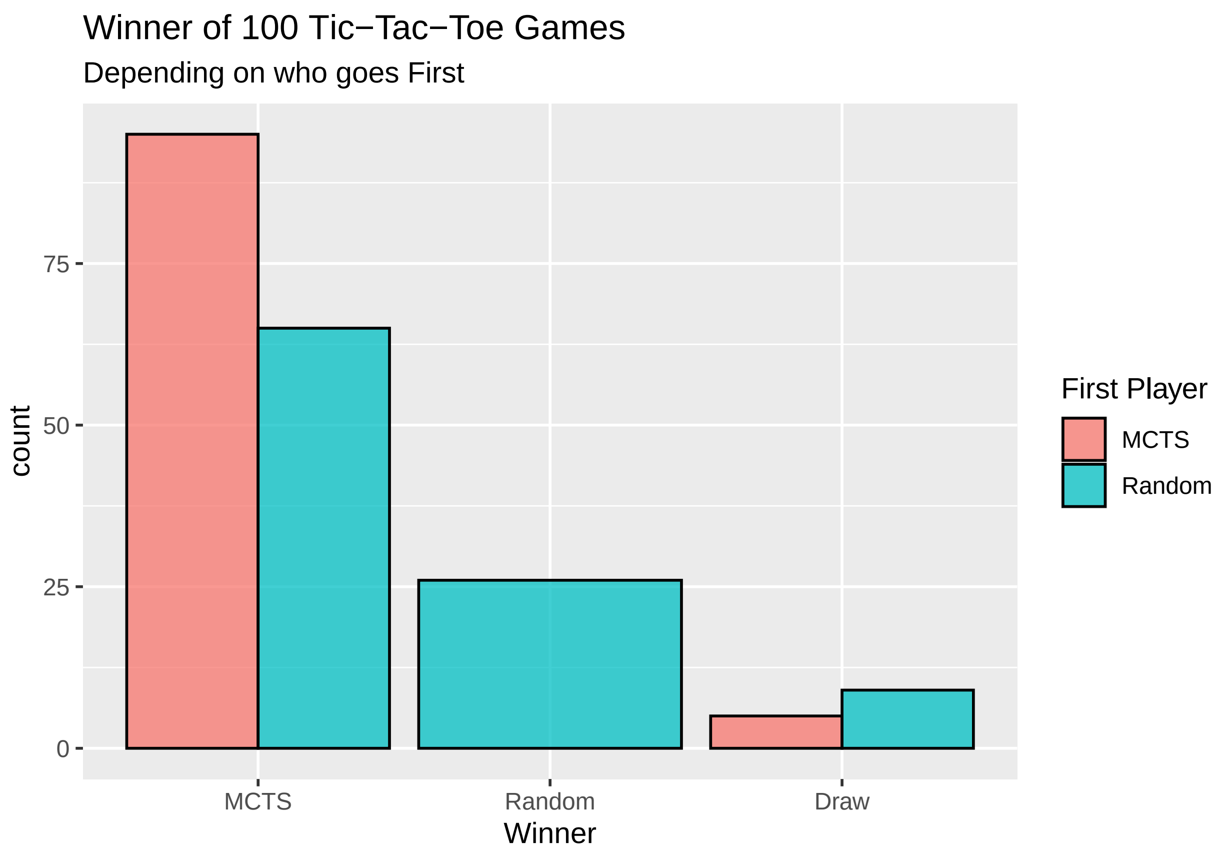
We choose 100 games to simulate in total since the runtime is consistently between 1-1.3 minutes. Additionally, of the games won, all of them result in either MCTS winning or a draw. This is what we expect since with enough iterations, the MCTS converges to the minimax tree, and similarly, a game played between a random player and a minimax player should result in either a draw or the minimax player winning.

How does who goes first effect the probability of each outcome?

```r
#set seed for reproducibility
set.seed(0)

# simulate 100 games, MCTS plays first
```

```r
winnerMCTS = MCTSvsRandomSim(numSims =100, first = 'MCTS', iterations = 500)
# simulate 100 games, Random Player goes first
winnerRandom = MCTSvsRandomSim(numSims = 100, first = 'random', iterations = 500)
```

```r
# create DF so that each game is a row with a column containing the who goes first
dfWinners <- pivot_longer(data.frame(winnerMCTS, winnerRandom),
                          cols = c('winnerMCTS', 'winnerRandom'))
colnames(dfWinners) <- c('First Player', 'Winner')
dfWinners$`First Player` = factor(dfWinners$`First Player`,
                                  levels = c('winnerMCTS', 'winnerRandom'),
                                  labels = c('MCTS', 'Random'))
dfWinners$Winner = factor(dfWinners$Winner,
                                  levels = c(1, -1, 0),
                                  labels = c('MCTS', 'Random', 'Draw'))
# create a bar plot of the results
ggplot(dfWinners) +
  geom_bar(aes(Winner, fill = `First Player`), position = 'dodge', color = 'black',
           alpha = 0.75) +
  ggtitle('Winner of 100 Tic-Tac-Toe Games',
          subtitle = 'Depending on who goes First')
```



```r
# create new DF to store the previous results and their probabilities
results <- data.frame(matrix(ncol = 3, nrow = 0))
colnames(results) <- c('Winner', 'First Player', 'Probability')

# update the new DF with the probabilities of each player winning on each board size
for (i in unique(dfWinners$`First Player`)){
```

Table 3: Probabilities of Winning Based on who Moves First

First.Player	MCTS	Random	Draw
MCTS	0.95	0.00	0.05
Random	0.65	0.26	0.09

```
  df <- dfWinners %>% filter(`First Player` == i)
  p1 <- length(df$Winner[df$Winner == 'MCTS']) / 100; p1
  p2 <- length(df$Winner[df$Winner == 'Random']) / 100; p2
  draw <- length(df$Winner[df$Winner == 'Draw']) / 100; draw

  res <- data.frame(Winner = c('MCTS', 'Random', 'Draw'),
                    'First Player' = i, Probability = c(p1,p2, draw))
  results <- rbind(results, res)
}

# create and output a pivot table of the results
pt <- pivot_wider(results, id_cols = First.Player, names_from = Winner, values_from = Probability);
knitr::kable(pt, align = 'cccc',
             caption = 'Probabilities of Winning Based on who Moves First') %>%
  kable_styling(full_width = F)
```

Our results show that if the MCTS player moves first, there is no probability that the Random player can win. The best case that can occur for Player 2 at this point is to tie with Player 1, and even then, the probability of ending in a draw is only 5%. If a Random Player moves first, even though MCTS is the second player, the MCTS player still wins 65% of the games. This is more than double probability of Player 2 winning if they play randomly (28%). Moreover, this shows that the MCTS method is extremely effective at determining the optimal move and beating a random player. Thus, we have evidence that the MCTS algorithm can improve Additionally, the probability of a draw occurring when either the MCTS or the random player moves first is lower than the probability of a draw occurring when both players play randomly.

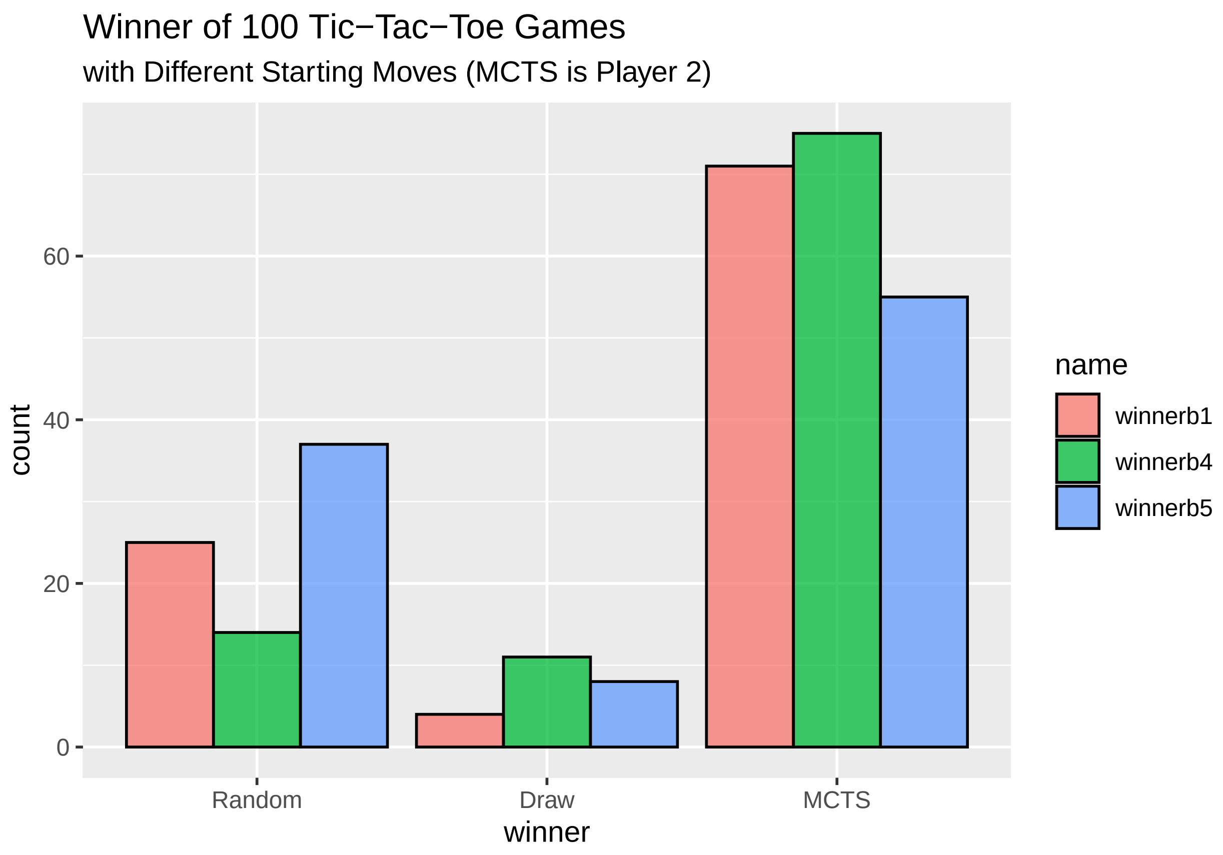
Given a Random Player's first move, what are the probabilities of each outcome?

We have already shown that despite being the second player, the probability that the MCTS player wins is higher than that of a random player. Thus, we break down the question further and estimate the probability of each outcome given a Random Player's first move.

```
# create the boards
board1 <- array(c(-1, rep(NA, 8)), dim = c(3,3))
board4 <- array(c(rep(NA, 3), -1, rep(NA, 4)), dim = c(3,3))
board5 <- array(c(rep(NA,4),-1,rep(NA,4)), dim = c(3,3))

set.seed(0)
start.time <- Sys.time()
# simulate 100 games with 500 iterations per move for each board
winnerb1= MCTSvsRandomSim(numSims =100, first = 'MCTS', iterations = 500, board = board1, player = -1)
winnerb4= MCTSvsRandomSim(numSims =100, first = 'MCTS', iterations = 500, board = board4, player = -1)
winnerb5= MCTSvsRandomSim(numSims =100, first = 'MCTS', iterations = 500, board = board5, player = -1)
end.time <- Sys.time()
time.taken <- end.time - start.time; time.taken
```

```
## Time difference of 8.585567 mins
```

```r
# create a DF to store all of the results and pivot so each row contains a game
winnerFirst <- data.frame(winnerb1, winnerb4, winnerb5)
winnerFirstLong = pivot_longer(winnerFirst, cols =colnames(winnerFirst))
winnerFirstLong$winner = factor(winnerFirstLong$value, levels = c(-1,0,1),
                                labels = c('Random', 'Draw', 'MCTS'))

# plot the barchart
ggplot(winnerFirstLong) +
  geom_bar(aes(winner, fill = name), position = 'dodge', color = 'black',
           alpha = 0.75) +
  ggtitle('Winner of 100 Tic-Tac-Toe Games',
          subtitle = 'with Different Starting Moves (MCTS is Player 2)')
```



```r
# create new DF to store the previous results and their probabilities
results <- data.frame(matrix(ncol = 3, nrow = 0))
colnames(results) <- c('Winner', 'First Move', 'Probability')
winnerFirstLong$name = factor(winnerFirstLong$name,
                              levels = c("winnerb1", "winnerb4", "winnerb5"),
                              labels = c(1, 4, 5))

# update the new DF with the probabilities of each player winning on each board size
for (i in unique(winnerFirstLong$name)){
  df <- winnerFirstLong %>% filter(name == i)
  p1 <- length(df$winner[df$winner == 'MCTS']) / 100; p1
  p2 <- length(df$winner[df$winner == 'Random']) / 100; p2
  draw <- length(df$winner[df$winner == 'Draw']) / 100; draw
```

Table 4: Probabilities of Winning Based on Random Player's First Move

First.Move	MCTS	Random	Draw
1	0.71	0.25	0.04
4	0.75	0.14	0.11
5	0.55	0.37	0.08

```r
  res <- data.frame(Winner = c('MCTS', 'Random', 'Draw'),
                    'First Move' = i, Probability = c(p1,p2, draw))
  results <- rbind(results, res)
}

# create and output a pivot table of the results
pt <- pivot_wider(results, id_cols = First.Move, names_from = Winner, values_from = Probability);
knitr::kable(pt, align = 'cccc',
             caption = "Probabilities of Winning Based on Random Player's First Move") %>%
  kable_styling(full_width = F)
```

The MCTS player has the highest probabilities of winning for games where the Random player first selects a box that is *not* the center box. This aligns with our intuitive understanding of tic-tac-toe since the center box is often believed to be the optimal first move. Our results support this notion since the Random player has the highest probability of winning the game against an opponent who strategizes when choosing the center box. When the random player selects the center box, the probability that the MCTS player wins is little better than flipping a coin. However, we still note that the MCTS is the second player, and this probability is still much higher than that of a second player that plays randomly.

Conclusion

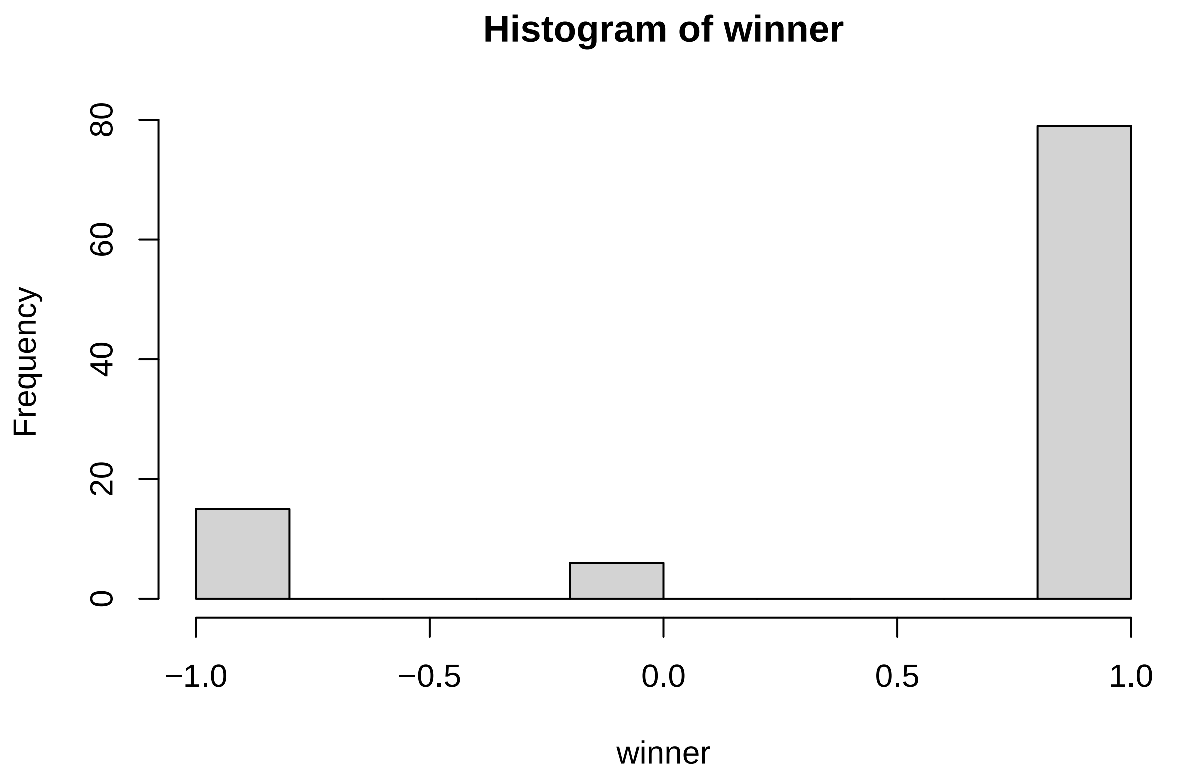
In this project, we have shown that using a Monte Carlo Search Tree to determine the next optimal move can effectively increase a player's probability of winning. So much so, that if the MCTS plays first against a Random player, that it will always win or tie against that player. We have implemented a MCTS demonstration using tic-tac-toe and we have done some experimenting to fine tune the tree's parameters. We have also answered which opening move is the best for a random player and the optimal move for a second random player.

Future Research

Further research includes implementing a minimax tree to compare with the MCTS method. It has been proven that the MCTS tree converges to the minimax with enough iterations, and so, we would like to investigate how many iterations are required to see that the difference in root probabilities is less than some ε. Additionally, we would like to tune the exploration parameter c used in the Upper Confidence Bound formula and see how changing the parameter changes the MCTS and game results. Furthermore, although this may be overkill for this problem, parallel computing is a technique commonly used with MCTS, and so we would like to look into this technique and the different ways to implement it. And, to connect MCTS to our first Random vs Random question (How does the size of the board effect the probability of winning?), we would like to generalize the MCTS implementation to any board size and see if it is still possible for the algorithm to win against a random player on larger boards where a draw is more likely.

Lastly, since the MCTS converges to the minimax tree, we expect both algorithms when playing against themselves to always result in a draw. However, we note that in our implementation, this is not the case; there are still players winning. Thus, we would like to revisit this project and fix our implementation of a MCTS player vs. MCTS player.

```
winner = replicate(100, MCTSvsMCTS(iterations = 500, player = -1))
hist(winner)
```



References

1. C. B. Browne et al., “A Survey of Monte Carlo Tree Search Methods,” in IEEE Transactions on Computational Intelligence and AI in Games, vol. 4, no. 1, pp. 1-43, March 2012, doi: 10.1109/TCIAIG.2012.2186810.
2. https://www.r-bloggers.com/2022/07/programming-a-simple-minimax-chess-engine-in-r/
3. https://towardsdatascience.com/reinforcement-learning-and-deep-reinforcement-learning-with-tic-tac-toe-588d09c41dda
4. https://en.wikipedia.org/wiki/Monte_Carlo_tree_search#Pure_Monte_Carlo_game_search
5. https://martin-ueding.de/posts/tic-tac-toe-with-monte-carlo-tree-search/
6. https://web.eecs.umich.edu/~weimerw/2014-4610/lectures/weimer-game-theory-intro.pdf
7. https://faculty.cc.gatech.edu/~surban6/2019fa-gameAI/lectures/2019-11-20%20minimax%20MCTS%20and%20CBR.pdf
8. https://nestedsoftware.com/2019/08/07/tic-tac-toe-with-mcts-2h5k.152104.html

GitHub Repository:

https://github.com/dkinsman/stat327project