Roc Flu Population Model

Therese Donovan

2020-02-15

Overview

This script is a translation of the Roc flu spreadsheet model we built in class to R. This model could be a simple R script, but I have chosen to build it in RMarkdown so that we can weave narrative along with calls to R.

We will be incorporating roc flu into the `popRoc()` function that we made last week. So let's begin by reviewing that model.

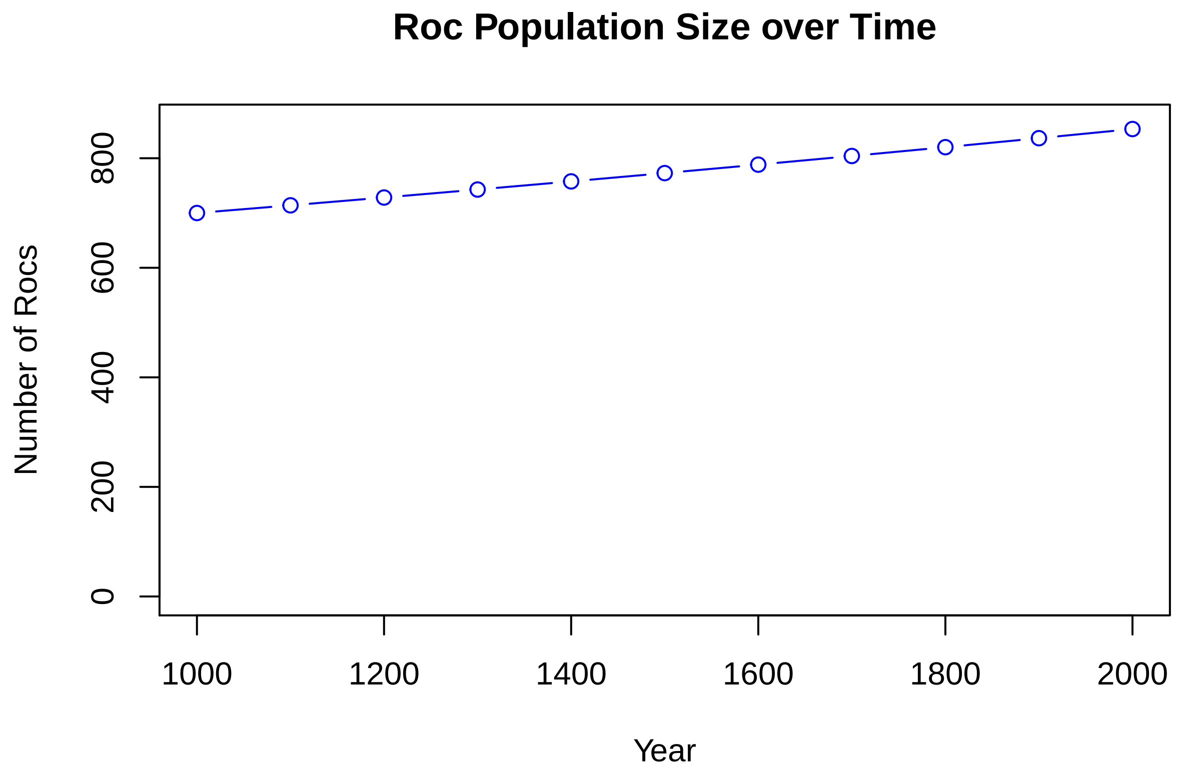
The Roc Model ——

Our Roc model is a function called `popRoc()`. Here's the original function; it has four arguments named *bullets*, *clutch.size*, *starting.pop*, and *plot.traj*. Notice that the *plot.traj* argument has been set to TRUE in the function definition. This means that this argument has a default value of TRUE. The other three arguments do not have default values. The user of the function will need to provide these values.

```
# create a new function call popRoc.
# it has four arguments: bullets, clutch.size, starting.pop, and plot.traj
# it returns the vector of population sizes of Rocs with each time step.

popRoc <- function(bullets, clutch.size, starting.pop, plot.traj = TRUE){

  # set the trajectory with the seq() function
  years <- seq(from = 1000, to = 2000, by = 100)

  # create a vector called rocs that will store our population trajectory
  rocs <- vector(mode = "numeric", length = length(years))

  # set the starting population
  rocs[1] <- starting.pop

  # a loop for generating the population through time
    for (y in 2:length(years)) {

      # this is your main roc model
      # in words: rocs in time step y is the is the number of rocs in the previous
      # time step minus those hit by bullets; this result is multiplied by clutch size.
      rocs[y] <- (rocs[y - 1] - bullets) * clutch.size
    }

  # plot a line graph
  if (plot.traj == TRUE) {
  plot(x = years, y = rocs,
         type = "b",
         col = "blue",
         xlab = "Year", ylab = "Number of Rocs",
         ylim = c(0, max(rocs) + 10),
         main = "Roc Population Size over Time")
  }

  # return the vector of population sizes
  return(rocs)

} # end of function
```

To test your function, call it (just like Excel) and send in some values for the arguments:

```
# to run this function, do something like this:
popRoc(bullets = 0, clutch.size = 1.02, starting.pop = 700, plot.traj = TRUE)
```



```
##  [1] 700.0000 714.0000 728.2800 742.8456 757.7025 772.8566 788.3137 804.0800
##  [9] 820.1616 836.5648 853.2961
```

Incorpating Roc Flu ——

We will modify our `popRoc()` function to include the flu. Our new function will be called `popRocFlu()` to keep it separate from the original `popRock()` model. Step 1 is to add two new arguments to our function. These will be called *flu.prob* (for probability of a flu in a century) and *flu.mortality* (the proportion of the Roc population that dies in the event of a flu).

```
popRocFlu <- function(bullets,
                      clutch.size,
                      starting.pop,
                      plot.traj = TRUE,
                      flu.prob,
                      flu.mortality
                      ){

       # code that takes the input arguments and creates outputs go here

} # end of function
```

Next, we can borrow code from our original `popRoc()` model and copy it into the function definition. The code below runs the original roc model and simply returns a vector of population sizes.

```
popRocFlu <- function(bullets,
                   clutch.size,
                   starting.pop,
                   plot.traj = TRUE,
                   flu.prob,
                   flu.mortality
                   ){
  # set the trajectory
  years <- seq(from = 1000, to = 2000, by = 100)

  # create the trajectory
  rocs <- vector(mode = "numeric", length = length(years))

  # set the starting population
  rocs[1] <- starting.pop

  # a loop for generating the population through time
    for (y in 2:length(years)) {

      # this is your main roc model
      rocs[y] <- max(0, (rocs[y - 1] - bullets) * clutch.size)
    }

  # return the vector of rocs
  return(rocs)

      }
```

All of this shoud be familiar. Notice that the main model has been wrapped in a `max()` function to stop the population from going negative. Let's test it out so far, adding some reasonable inputs:

```
popRocFlu(bullets = 20,
          clutch.size = 1.02,
          starting.pop = 700,
          plot.traj = TRUE,
          flu.prob = 0,
          flu.mortality = 0)
```

```
##  [1] 700.0000 693.6000 687.0720 680.4134 673.6217 666.6941 659.6280 652.4206
##  [9] 645.0690 637.5704 629.9218
```

So far, so good. The function returns a vector of population sizes, given the inputs. But we haven't done anything with the arguments *flu.prob* or *flu.mortality* yet. Where would we add these? They belong within the time step loop, where we compute our total roc population size. Our current model is:

```
# this is your main roc model
      rocs[y] <- (rocs[y - 1] - bullets) * clutch.size
```

How can we incorporate the flu here? We need a firm idea of where we are headed before we start to code, and our spreadsheet version gives us a great starting point. In the spreadsheet, we had a column that generated a random number between 0 and 1, another column that returned a 1 if the century had a flu outbreak, and a final column that calculated flu deaths in the event of an outbreak. We'll follow these steps below, making use of the `runif()` and `ifelse()` functions. Make sure to use `help()` to scan the help pages whenever you use a new function!

Focusing on the loop portion of our `popRocFlu()` function, here are the adjustments:

```
# a loop for generating the population through time
    for (y in 2:length(years)) {

      # generate a random number with the runif function
      rand <- runif(n = 1, min = 0, max = 1)

     # determine if a flu occurred in this time step
      flu <- ifelse(test = rand < flu.prob, yes = 1, no = 0)

     # calculate the mortality rate booster
      flu.mortality.rate <- flu.mortality * flu

      # and now incorporate this into our main model equation
      rocs[y] <- (rocs[y - 1] * (1 - flu.mortality.rate) - bullets) * clutch.size
    }
```

Our loop, which does the calculations for a single time step, now does several things:

- We use the `runif()` function to generate a random number. This is similar to =RAND() in Excel. We store that random number as an object called **rand**. It is a vector of length 1, and it stores a number.
- We use the `ifelse()` function to return a 1 if flu occurred, and 0 if it did not. This is similar to Excel's =IF() function. We store that result as an object called **flu**. It is a vector of length 1, and stores a number.
- We then calculate the flu mortality rate for the timestep by multiplying *flu.mortality* (an input to the function) by **flu**. If there is no flu, the flu mortality rate will be 0. And if there is a flu that century, the flu mortality rate will equal the *flu.mortality* input.
- Finally, we adjust our main model equation so that the number of Rocs in this time step is the number of Rocs in the previous time step, multiplied by the proportion of Rocs that escaped the flu (Tony called this "q" in the video). From this, we further subtract off individuals that were shot by silver bullets. This is the total number of Rocs that can breed, and we multiply by the clutch size to get the final number.

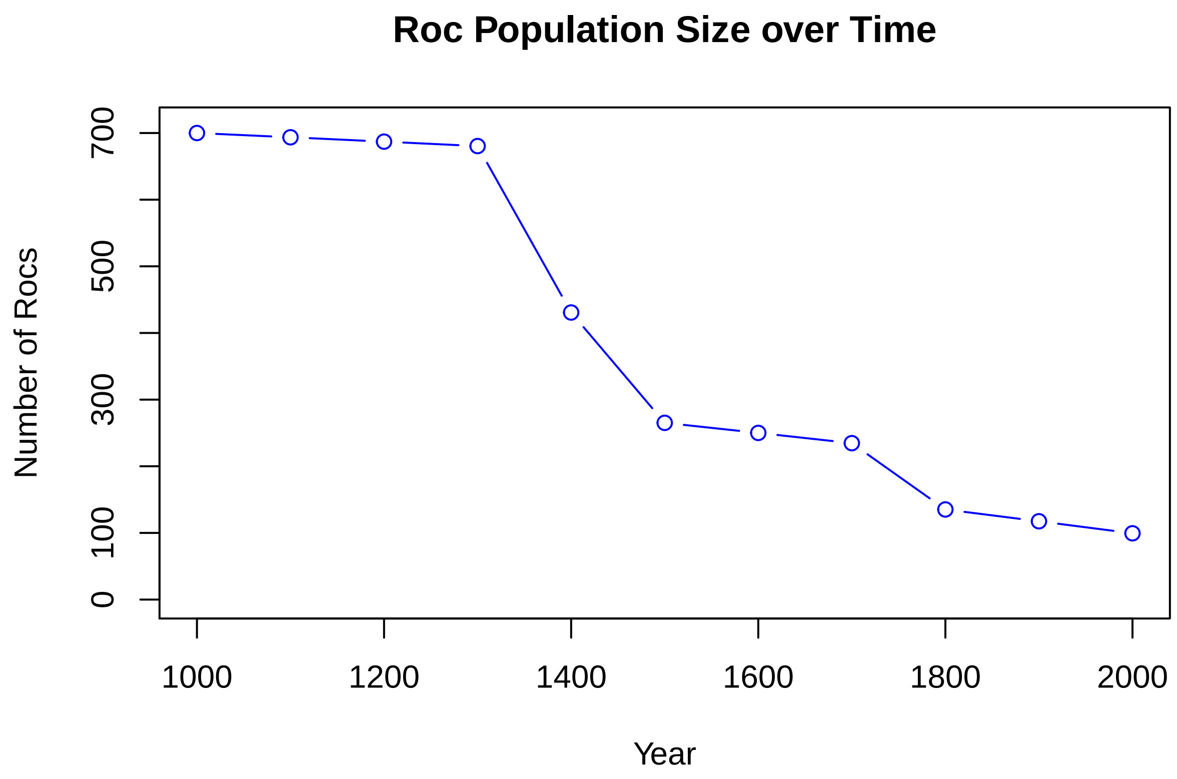
Our full `popRocFlu()` function is shown below. Note that I added a vector called **outbreaks** before the loop so that we can track flu outbreaks over the simulation. I also wrapped our key population result in a `max()` function to stop the population from going negative. Toward the end of the function, I also added the code from `popRoc()` that produces a graph of the population trajectory.

The function returns a list. The first part of the list is named "rocs" and it contains the population trajectory. The second part of the list is named "outbreaks" and it contains the outbreak vector.

```
popRocFlu <- function(bullets,
                     clutch.size,
                     starting.pop,
                     plot.traj = TRUE,
                   flu.prob,
                   flu.mortality
                   ){

  # set the trajectory
  years <- seq(from = 1000, to = 2000, by = 100)

  # create a vector that will store the population trajectory
  rocs <- vector(mode = "numeric", length = length(years))

  # create a vector that will store flu outbreaks
  outbreaks <- vector(mode = "numeric", length = length(years))

  # set the starting population
  rocs[1] <- starting.pop

  # a loop for generating the population through time
    for (y in 2:length(years)) {

      # generate a random number
      rand <- runif(n = 1, min = 0, max = 1)

      # determine if a flu occurred in this time step
      flu <- ifelse(test = rand < flu.prob, yes = 1, no = 0)

      # add the result to the outbreaks vector
      outbreaks[y] <- flu

      # calculate the mortality rate booster
      flu.mortality.rate <- flu.mortality * flu

       # and now incorporate this into our main model equation
      rocs[y] <- max(0, (rocs[y - 1] * (1 - flu.mortality.rate) - bullets) * clutch.size)
    }

  # plot a line graph
  if (plot.traj == TRUE) {
  plot(x = years, y = rocs,
       type = "b",
       col = "blue",
       xlab = "Year", ylab = "Number of Rocs",
       ylim = c(0, max(rocs) + 10),
       main = "Roc Population Size over Time")
  }

  # return the vector of rocs
  return(list(rocs = rocs,
              outbreaks = outbreaks))

      }
```

Let's try the `popRocFlu()` function now, using the same inputs that we used in the class video.

```
simulation1 <- popRocFlu(bullets = 20,
                         clutch.size = 1.02,
                         starting.pop = 700,
                         plot.traj = TRUE,
                         flu.prob = 0.25,
                         flu.mortality = 0.35)
```



Let's look at the new object we created called **simulation1**. You can never over-use the `str()` function to learn more about the structure of an R object. You can also look for the object in the global environment.

```
str(simulation1)
```

```
## List of 2
##  $ rocs     : num [1:11] 700 694 687 680 431 ...
##  $ outbreaks: num [1:11] 0 0 0 0 1 1 0 0 1 0 ...
```

A Word About Reproducibility ——

When you have a script or function that involves some random draw, you will get new results each time you run the function. For instance, let's re-run our `popRocFlu()` function with identical inputs, and save the result as **simulation2**:

```
simulation2 <- popRocFlu(bullets = 10,
                         clutch.size = 1.2,
                         starting.pop = 700,
                         plot.traj = FALSE,
                         flu.prob = 0.25,
                         flu.mortality = 0.35)
```

Let's compare our two simulations. Remeber, our outputs are provided in a **list**, and we only want to retrieve that portion of the list that is named "rocs":

```
simulation1$rocs
```

```
##  [1] 700.00000 693.60000 687.07200 680.41344 430.71411 265.16346 250.06672
##  [8] 234.66806 135.18492 117.48862  99.43839
```

```
simulation2$rocs
```

```
##  [1]  700.000  828.000  981.600 1165.920 1387.104 1652.525 1971.030 2353.236
##  [9] 2811.883 3362.259 4022.711
```

This is just like pressing F9 to generate new simulations in Excel. When you run simulations in Excel, you generate a sequence of random numbers. Unless you store those numbers somehow, it is difficult to perfectly reproduce a single simulation.

But R has a very cool trick that allows you to generate the same simulation results time and time again, and that is the `set.seed()` function. Here is some code taken from the `set.seed()` helpfile:

```
# set the seed to 42, then generate 30 random uniform values
set.seed(42)
runif(30)
```

```
##  [1] 0.91480604 0.93707541 0.28613953 0.83044763 0.64174552 0.51909595
##  [7] 0.73658831 0.13466660 0.65699229 0.70506478 0.45774178 0.71911225
## [13] 0.93467225 0.25542882 0.46229282 0.94001452 0.97822643 0.11748736
## [19] 0.47499708 0.56033275 0.90403139 0.13871017 0.98889173 0.94666823
## [25] 0.08243756 0.51421178 0.39020347 0.90573813 0.44696963 0.83600426
```

```
# set the seed to 42, then generate 30 random uniform values
set.seed(42)
runif(30)
```

```
##  [1] 0.91480604 0.93707541 0.28613953 0.83044763 0.64174552 0.51909595
##  [7] 0.73658831 0.13466660 0.65699229 0.70506478 0.45774178 0.71911225
## [13] 0.93467225 0.25542882 0.46229282 0.94001452 0.97822643 0.11748736
## [19] 0.47499708 0.56033275 0.90403139 0.13871017 0.98889173 0.94666823
## [25] 0.08243756 0.51421178 0.39020347 0.90573813 0.44696963 0.83600426
```

When you set a random number generator seed, you can reproduce the exact same sequence even though random numbers are involved, as long as you use the same seed that is. Let's confirm this with our `popRocFlu()` function.

```
# set the seed to 100, and run your projection
set.seed(100)
simulation1 <- popRocFlu(bullets = 10,
                         clutch.size = 1.2,
                         starting.pop = 700,
                         plot.traj = FALSE,
                         flu.prob = 0.25,
                         flu.mortality = 0.35)


# reproduce those exact same results by setting the seed again
set.seed(100)
simulation2 <- popRocFlu(bullets = 10,
                         clutch.size = 1.2,
                         starting.pop = 700,
                         plot.traj = FALSE,
                         flu.prob = 0.25,
                         flu.mortality = 0.35)

# compare the two simulations; note they are identical
simulation1$rocs
```

```
##  [1]  700.0000  828.0000  981.6000 1165.9200  897.4176 1064.9011 1265.8813
##  [8] 1507.0576 1796.4691 2143.7630 1660.1351
```

```
simulation2$rocs
```

```
##  [1]  700.0000  828.0000  981.6000 1165.9200  897.4176 1064.9011 1265.8813
##  [8] 1507.0576 1796.4691 2143.7630 1660.1351
```

The two trials produce the same results. The bottom line: if you have stochasticity in your model, the `set.seed()` function in invaluable for reproducing results.

Looping through Simulations ——

Now that our `popRocFlu()` function is working, we can use it to get to the heart of what the Caliph wants to know: What is the probability that the Roc population will dip below a particular value? To answer that, we will run the `popRocFlu()` function 1,000 times, with an end goal of computing these probabilities.

There are several ways we could do this. For example, we could write a new function call `fluSim()`. We'll do that now, and assume the function has an argument called *trials* (which allows the user to set the number of replicate runs.) But, we also want our user to pass arguments that are sent to `popRocFlu()`, because after all, that is the function that we'll be running over and over again. Here's one approach:

```
fluSim <- function(trials, bullets, clutch.size,
                  starting.pop, plot.traj,
                  flu.prob, flu.mortality){

  # set up a vector to store final population size of each trial
  rocs <- vector(mode = "numeric", length = trials)

  # set up a vector to total number of outbreaks for trial
  outbreaks <- vector(mode = "numeric", length = trials)

  # set up a loop and loop through trials
  for (i in 1:trials) {

    # run the popRocFlu() function, given the user's inputs
    # bullets is an argument name in the popRocFlu function,
    # but it  is also an argument in the fluSim function
    results <- popRocFlu(bullets = bullets,
                         clutch.size = clutch.size,
                         starting.pop = starting.pop,
                         plot.traj = FALSE,
                         flu.prob = flu.prob,
                         flu.mortality = flu.mortality)

    # store the final population size in the rocs vector
    rocs[i] <- tail(results$rocs, n = 1)

    # store the total outbreaks in the outbreaks vector
    outbreaks[i] <- sum(results$outbreaks)
  } # end of trials


  return(list(rocs = rocs, outbreaks = outbreaks))

}
```

Let's now run this function, setting *trials* to 10 (to test the function) and storing our output as a new object called **sim.results**. This will be a list. Can you see why?

```
# first set the random seed so that you can reproduce results later
set.seed(400)

# now run the fluSim() function
sim.results <- fluSim(trials = 10, bullets = 20, clutch.size = 1.02,
                  starting.pop = 700,
                  flu.prob = 0.25, flu.mortality = 0.35)

# look at the structure of the results
str(sim.results)
```

```
## List of 2
##  $ rocs     : num [1:10] 65.3 331.3 55.4 395 348.2 ...
##  $ outbreaks: num [1:10] 3 1 4 1 1 0 0 1 4 3
```

As thought, **sim.results** is a list (because we returned a list to our user in the `fluSim()` function). The first part of this list is named "rocs", and is a vector of length 10. It is storing the final Roc population size in the year 2000 for each of our 10 trials. The second element of our list is called "outbreaks", and it stores the total outbreaks for any given simulation.

We can do a lot with these results. For example, we can compute some common summary statistics with the functions below (make sure to read their helpfiles!).

```
# compute the min
min(sim.results$rocs)
```

```
## [1] 32.3165
```

```
# compute the max
max(sim.results$rocs)
```

```
## [1] 629.9218
```

```
# compute the mean
mean(sim.results$rocs)
```

```
## [1] 291.5115
```

```
# compute the standard deviation
sd(sim.results$rocs)
```

```
## [1] 227.7317
```

```
# get the quartiles
quantile(sim.results$rocs)
```

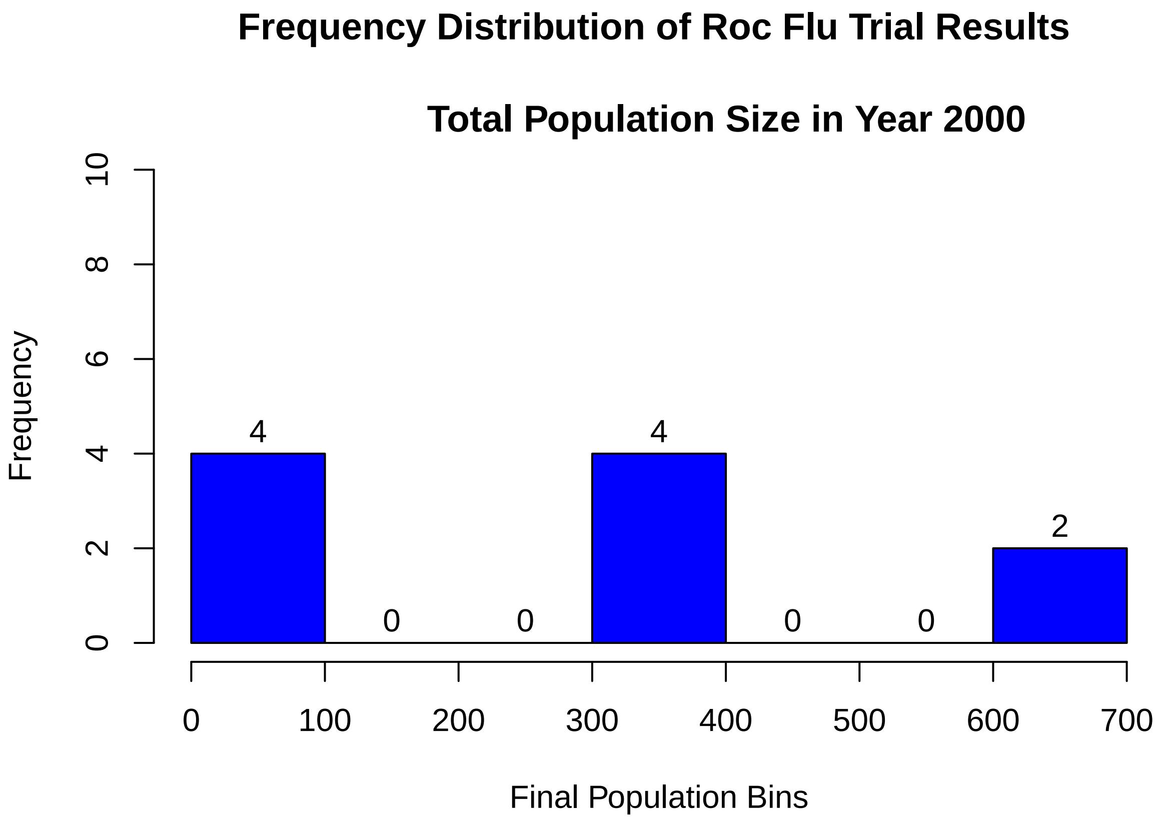
```
##        0%       25%       50%       75%      100%
##  32.31650  68.85476 339.71746 383.31146 629.92179
```

The quantiles are really helpful (in my opinion). If you take the trials, and line them up from the smallest to the largest *in order*, the `quantile()` function will tell you what percentage of the data lie above and below a given point. For example, the 25th percentile is 68.85476. This means that 25% of our simulations had final population sizes that were *lower* than this value, and 75% of our simulations were *higher* than this number. The 50th percentile is also called the median.

To present our results as a histogram, we can use R's `hist()` function. This function has many arguments that let you customize a histogram. Definitely check out the help page for this function. Unless you provide the arguments explicitly, `hist()` will try to create a default histogram to the best of its ability using the default argument values (of which there are many).

Let's run the function, and save our outputs as a new object called **h** (for **h**istogram). We'll look at this object soon.

```r
# https://www.datamentor.io/r-programming/histogram/
h <- hist(x = sim.results$rocs,
          breaks = seq(from = 0, to = 700, by = 100),
          freq = TRUE,
          col = "blue",
          main = "Frequency Distribution of Roc Flu Trial Results \n
             Total Population Size in Year 2000",
          xlab = "Final Population Bins", ylab = "Frequency",
          ylim = c(0,10),
          labels = TRUE)
```



For now, see if you can match up each argument with the appearance on the graph.

We only ran 10 trials, but the histogram is showing how the 10 final population sizes fall by "bin". Let's now have a look at our **h** object – this is so very instructive!

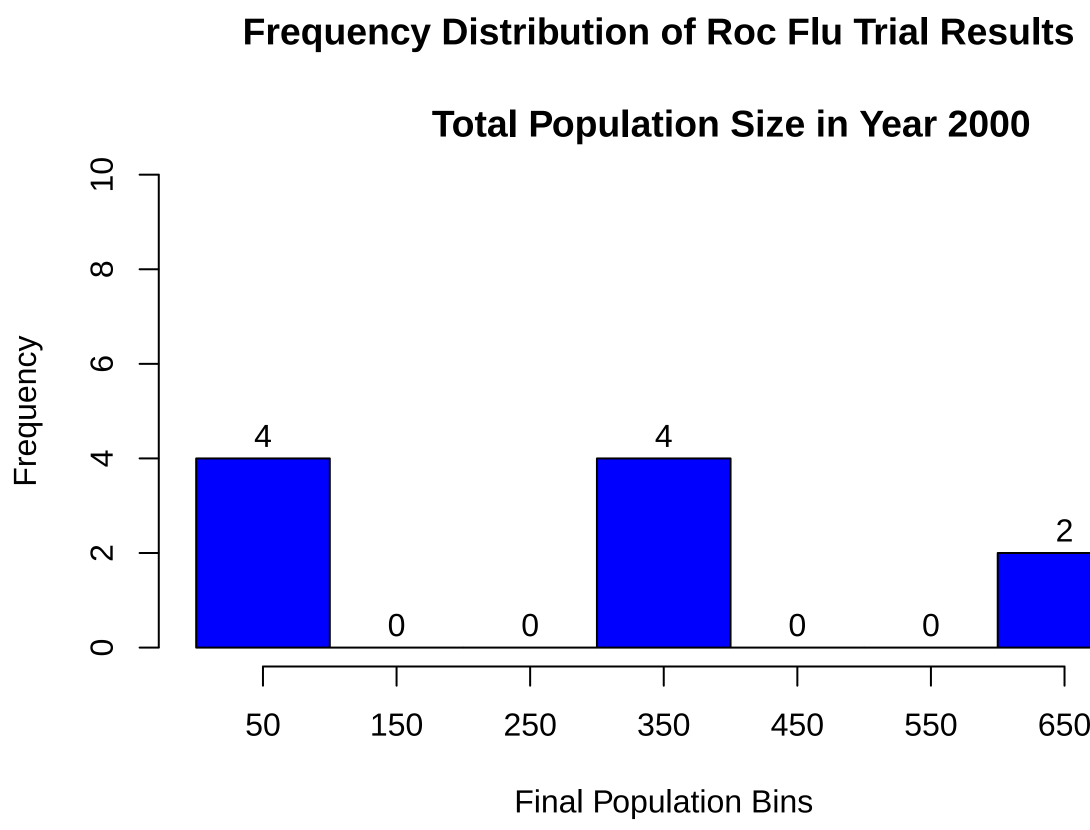
```r
# look at the object called h
h
```

```
## $breaks
## [1]   0 100 200 300 400 500 600 700
##
## $counts
## [1] 4 0 0 4 0 0 2
##
## $density
## [1] 0.004 0.000 0.000 0.004 0.000 0.000 0.002
##
## $mids
## [1]  50 150 250 350 450 550 650
##
## $xname
## [1] "sim.results$rocs"
##
## $equidist
## [1] TRUE
##
## attr(,"class")
## [1] "histogram"
```

We can see that *h* is a list, and this list provides all of the information needed to create a histogram, such as where the "breaks" are and the "counts" associated with each bin. The "density" element of this list gives the something along the lines of probability, but not exactly. You can see that these don't add to 1. What gives? Let's take it bar by bar. The first bar in our histogram has a count of 4. This is the height of this bar. What is the width of this bar? It's 100. This means that this bar has an area of 400 units. Across the graph, we have a total area of 1000 units. Now, what is the density of the first bar? It's the count of the bar (4) divided by the total area (1000), which is 0.004.

If you want to change something in your histogram, such as the breaks, just set the *breaks* argument to something else. I'm fine with the breaks, but I'd like to center my bars over the midpoints to make the histogram more readable. To do that, we need to set the *xaxt* graphical argument to "n" (which presumably stands for x-axis text) – this will suppress the x axis labels. Then we'll need to provide our home-made labels with the `axis()` function, where we can make use of the "mids" information stored in **h**:

```
h <- hist(x = sim.results$rocs,
     freq = TRUE,
     col = "blue",
     main = "Frequency Distribution of Roc Flu Trial Results \n
             Total Population Size in Year 2000",
     xlab = "Final Population Bins", ylab = "Frequency",
     ylim = c(0,10),
     labels = TRUE,
     xaxt = "n")

axis(side = 1, at = h$mids)
```



Now that we know a bit more about the **fluSim()** output, we can run our simulation with 1000 trials, just like we did with a spreadsheet macro.

```
# first set the random seed so that you can reproduce results later
set.seed(400)

# now run the fluSim() function
sim.results <- fluSim(trials = 1000, bullets = 20, clutch.size = 1.02,
                      starting.pop = 700,
                      flu.prob = 0.25, flu.mortality = 0.35)

# look at the structure of the results
str(sim.results)
```

```
## List of 2
##  $ rocs     : num [1:1000] 65.3 331.3 55.4 395 348.2 ...
##  $ outbreaks: num [1:1000] 3 1 4 1 1 0 0 1 4 3 ...
```

Now you can see that our **sim.results** object stores 1000 elements in each vector. A frequency histogram is often useful to illustrate the results, but in this case the Caliph wants to know the **probability** that the Roc population will dip below 100. I could not find an easy way to do this with `hist()` . . . if you know of a way, please let me know!

One way to create the probability graphic this is to use the `cut()` function to "cut" the population sizes into bins directly.

```
# look at the first 20 records of our simulation
head(sim.results$rocs, n = 20)
```

```
##  [1]  65.26128 331.26815  55.43030 395.02636 348.16676 629.92179 629.92179
##  [8] 348.16676  32.31650  79.63518  18.50076  58.10872 208.18430 118.61748
## [15]   0.00000  98.98116 356.36838  29.97314 331.26815 209.78819
```

```r
# get the maximum pop size and add a buffer
max.pop <- max(sim.results$rocs) + 100

# cut the population sizes into bins with the cut function
cut.pop <- cut(sim.results$rocs,
               breaks = seq(from = 0, to = max.pop, by = 100),
               include.lowest = TRUE)

# look at the first 20 records of this new object
head(cut.pop, n = 20)
```

```
##  [1] [0,100]   (300,400] [0,100]   (300,400] (300,400] (600,700] (600,700]
##  [8] (300,400] [0,100]   [0,100]   [0,100]   [0,100]   (200,300] (100,200]
## [15] [0,100]   [0,100]   (300,400] [0,100]   (300,400] (200,300]
## 7 Levels: [0,100] (100,200] (200,300] (300,400] (400,500] ... (600,700]
```

What did the `cut()` function do? It produces a vector, where each element was binned into a particular category (i.e., a factor). The first simulation result (trial = 1) resulted in a final population size of 65. It was placed into a "bin" of [0,100]. Trial 2 resulted in a final population size of 331, and was placed in a "bin" of (300,400]. See the rounded and square brackets? Which type you use identify whether the interval is "open" or "closed".

> From Wikipedia: "An open interval does not include its endpoints, and is indicated with parentheses. For example, (0,1) means greater than 0 and less than 1. A closed interval is an interval which includes all its limit points, and is denoted with square brackets. For example, [0,1] means greater than or equal to 0 and less than or equal to 1. A half-open interval includes only one of its endpoints, and is denoted by mixing the notations for open and closed intervals. (0,1] means greater than 0 and less than or equal to 1, while [0,1) means greater than or equal to 0 and less than 1." Source: https://en.wikipedia.org/wiki/Interval_(mathematics)

Now, we have a vector that identifies what bin each trial was put into. Next, we can use the `table()` in R to count the number of times , which is very much like Excel's FREQUENCY function:

```r
# use the table() function to obtain frequency counts
(tally <- table(cut.pop))
```

```
## cut.pop
##   [0,100] (100,200] (200,300] (300,400] (400,500] (500,600] (600,700]
##       324       235       166       189        23         0        63
```

So, out of 1000 trials, 324 trails resulted in a final Roc population size that fell in the interval [0,100].

To get to probability, we divide each of our tally elements by 1000 (our total trials):

```r
# divide each tally element by 1000; this is an example of "vectorization" in R
(probs <- tally/1000)
```

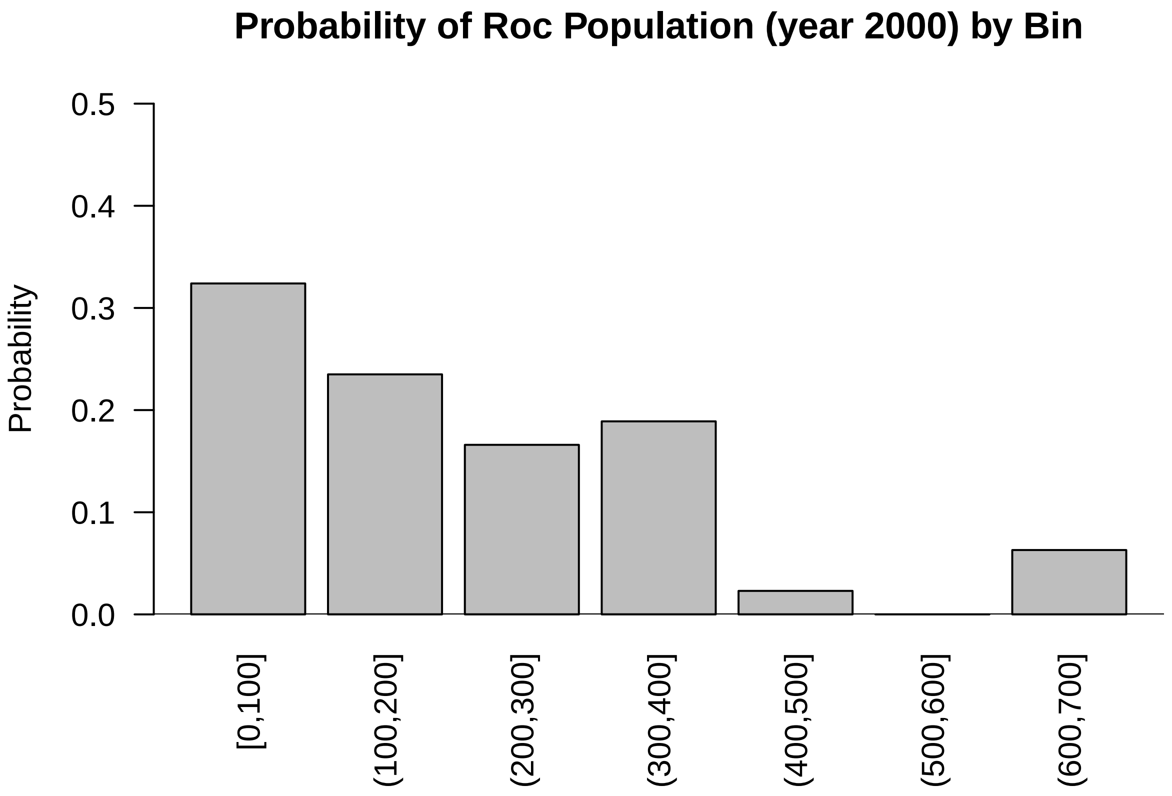
```
## cut.pop
##   [0,100] (100,200] (200,300] (300,400] (400,500] (500,600] (600,700]
##     0.324     0.235     0.166     0.189     0.023     0.000     0.063
```

```r
# double check that the probabilities sum to 1
sum(probs)
```

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

Finally, we can plot the histogram with a `barplot()` function.

```r
barplot(height = probs,
        ylab = "Probability",
        main = "Probability of Roc Population (year 2000) by Bin",
        ylim = c(0,0.5),
        las = 2)
abline(h = 0)
```



Now we're talking. This graph summarizes the results of our 1000 Roc Flu trials in a way that can be conveyed to the Caliph.

Reflection

Now that you have an R version of Roc flu model, ponder these questions:

- What information should you present to the Caliph?
- Why is it important to run multiple trials?
- How do you get from your model world back to the real world?