Roc Population Model

Therese Donovan

2020-02-07

Overview

This script is a translation of the Roc model we built in class from a spreadsheet model to an R model. 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. I've also elected to build the model as a **function** in R – one that we will create from scratch.

General tips for coding your model:

- Have an idea for what key inputs and outputs are needed. The spreadsheet model gives us a great head start on this.
- Comment your code liberally! This is crucial if you are sharing your code with others, or even sharing it with your future self.
- Within reason, add argument names to your function calls - this adds clarity.
- Follow some style guidelines to keep your code readable.
- Make sure to look in the global environment after each line is executed! Double check that the class or mode is what you were expecting it to be.

Functions in R ——

Functions in R are similar in some ways to functions in Excel:

- they have a name, such as SUM
- they have arguments that are named
- the function does something, and may return a result

To run a function in Excel, you enter the name, open the parentheses, specify the arguments, and close the parentheses. Some examples:

- =SUM(1,2)
- =SUM(A2,F6)
- =IF(A6 = 10, "do this", "do that")
- =MAX(0, 400)
- =COUNTIF(A1:A17, "bananas")

You can create your own functions in both R and Excel, and name it anything you want. You will need to identify what are the inputs to the function (arguments), and you will need to provide the code that will soak in the inputs, and do what is needed to produce an output.

In R, a function looks like this:

```
add.two <- function(x, z){

  answer <- x + z
  return(answer)
}
```

Here, we created a new function called `add.two()`. This function has two arguments, called *x* and *z*. The user of this function will pass in values for these two arguments: the function will add them together to create an object called **answer**, and will return that object. Let's try it:

```
add.two(x = 2, z = 8)
```

```
## [1] 10
```

And one more for kicks:

```
add.two(x = 7, z = 45)
```

```
## [1] 52
```

If we want to store the result in R, we would need to assign the result to an object:

```
my.result <- add.two(x = 7, z = 45)

# now show it
my.result
```

```
## [1] 52
```

The Roc Model ——

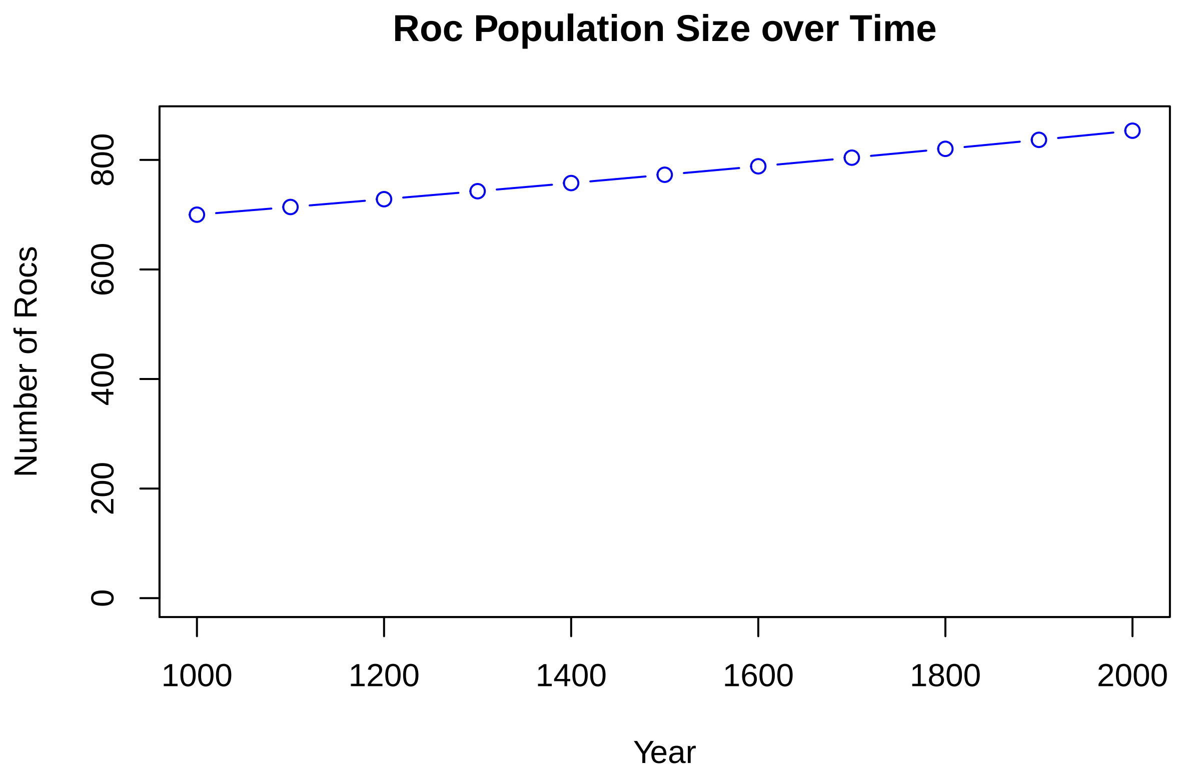
Our Roc model will be a function called `popRoc()`. We didn't need to make a function, but it is clean. Later, I'll show you how to debug your function so that you can study what R is producing in each and every line.

Here's the 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.

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

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

  # 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] <- (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 the arguments

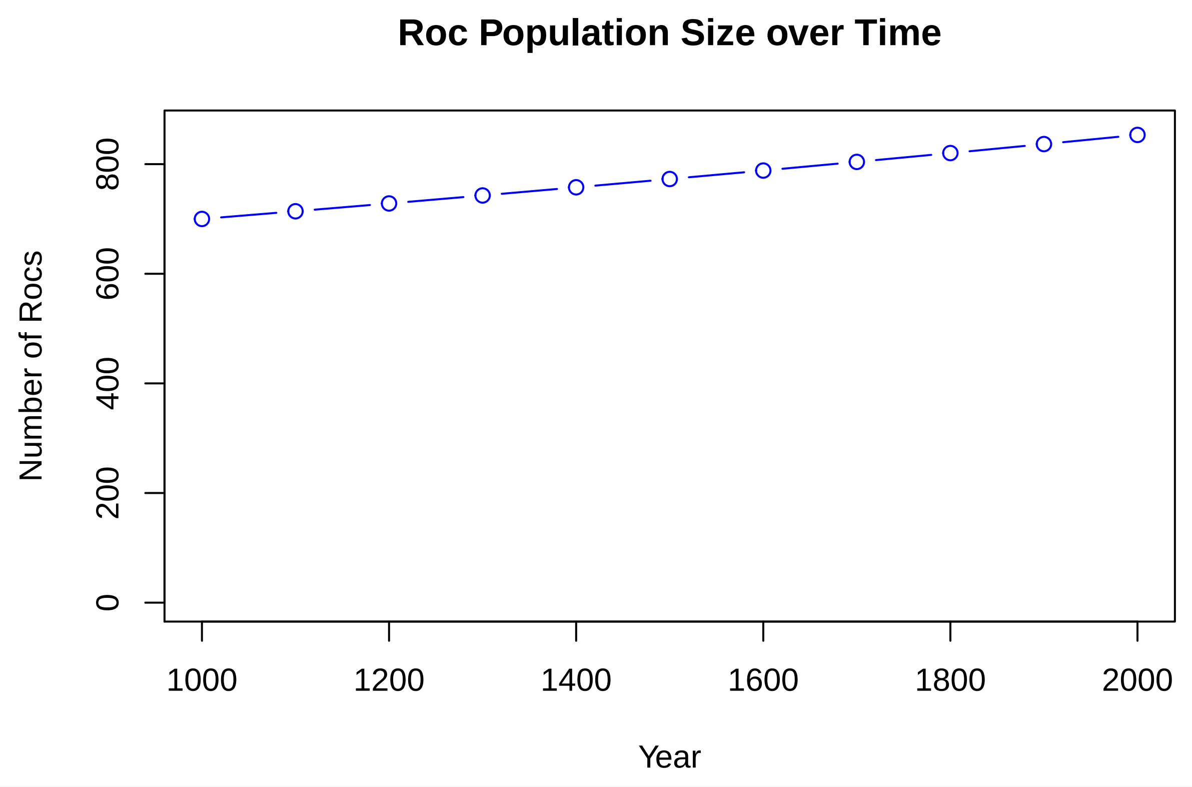
```
# to run this function, do something like this:
popRoc(bullets = 0, clutch.size = 1.02, starting.pop = 700)
```



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

To save the population trajectory, you need to save the results as an object:

```
rocs <- popRoc(bullets = 0, clutch.size = 1.02, starting.pop = 700)
```

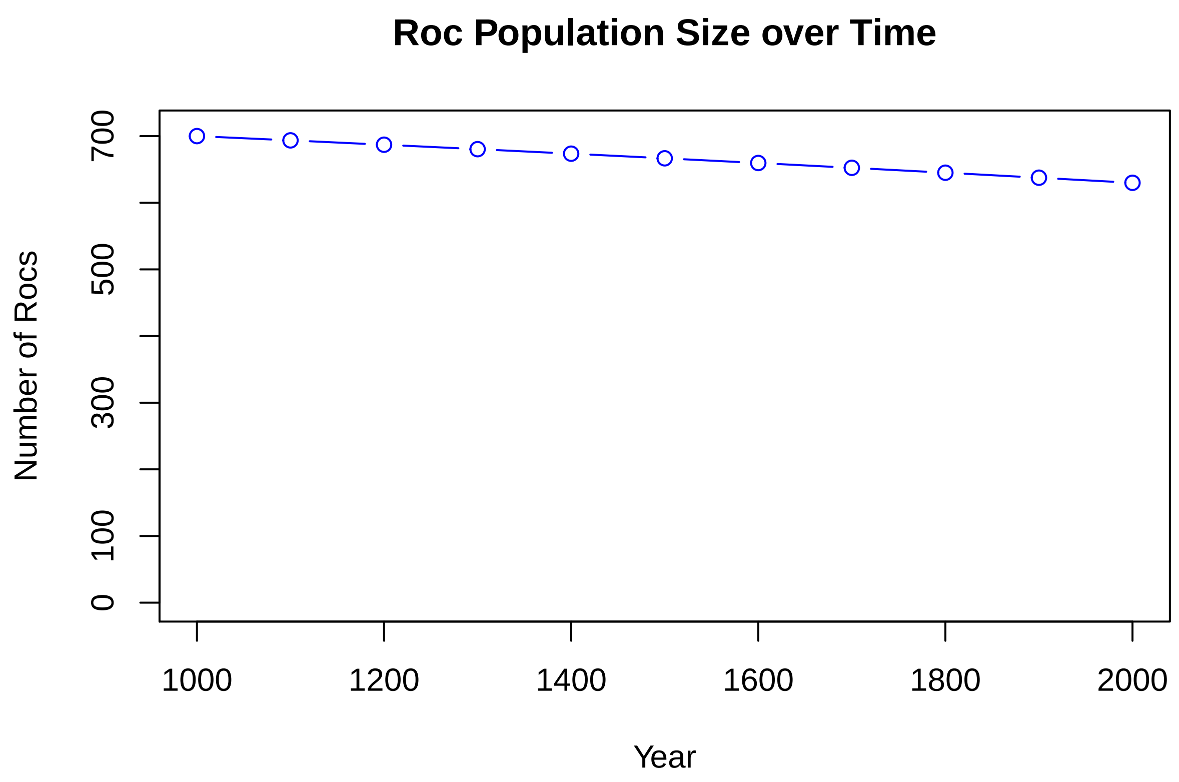


```r
# show the returned object
rocs
```

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

Let's try it again with new inputs, and have the results immediately displayed by wrapping everything in parentheses.

```r
# let's try it with new inputs, and return the result immediately
(rocs <- popRoc(bullets = 20, clutch.size = 1.02, starting.pop = 700))
```



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

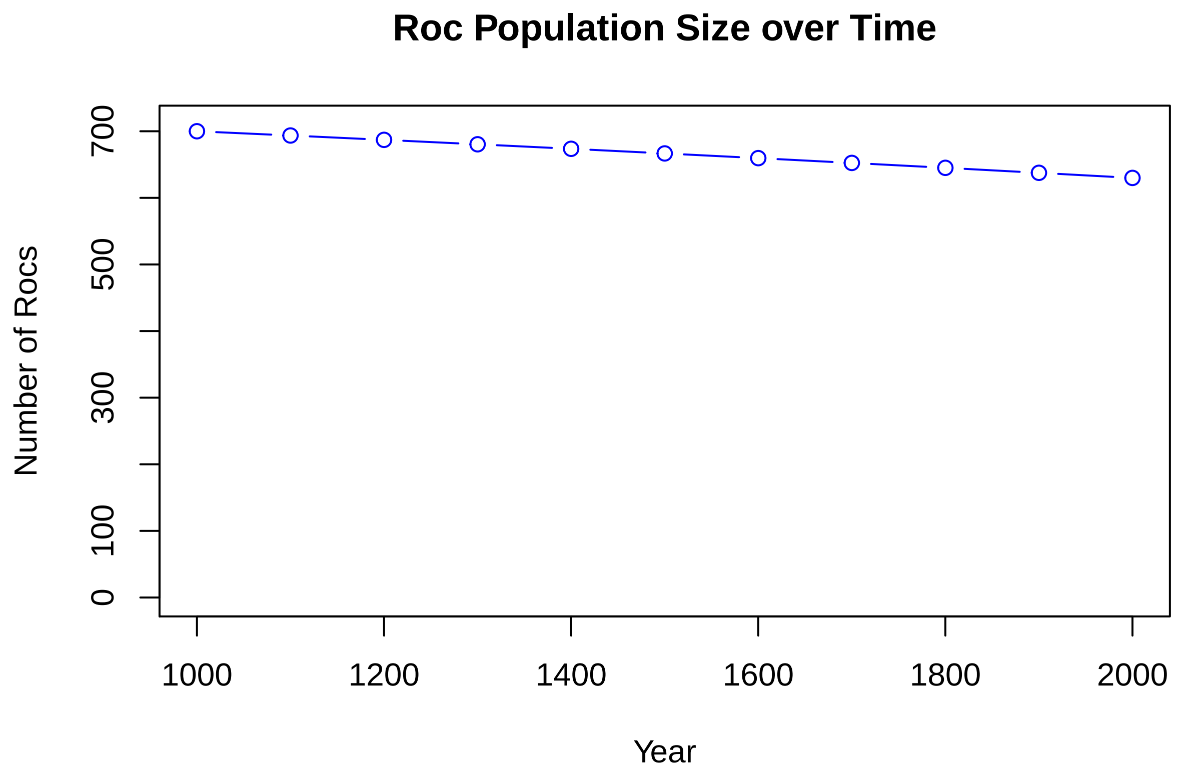
Some Questions:

1. How can you modify your code to keep the population from going negative?
   - Wrap your pop sizes in a max function. E.g., max(0, rocs[11])
2. Are there options besides writing a loop?
   - Maybe! There are 100 different ways this function could have been written. A different instructor would surely have written a different function.
3. Is there a way to debug your loop easily?
   - Yes, put a `browser()` call near the top of the function, like shown below.

```r
# add a browser() call to the popRoc function to allow debugging.

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

  browser()

  # 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 model
      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
```

When the function is run, it will pop you into debug mode. This is an incredibly useful tool in R (I live in browser mode). Try it by running the code below!

```
rocs <- popRoc(bullets = 20, clutch.size = 1.02, starting.pop = 700)
```



The "debug" mode allows you step through each line of your function one at a time. After you run a line, look in R Studio's environment panel, and notice a few things:

1. You are in the *function's* environment. This shows all of the objects that the function can see.
2. As you progress through each line, look into this environment and see the objects that have been

created, and what their values are.
3. You can actually change these objects in your console (while still in the function!). This allows you to figure out how to fix bugs and make your changes.

Sensitivity Analysis ——

Our key model output is Roc population size in the year 2000. How does changing each of these inputs affect the key model output? In other words, how sensitive is the model output with respect to a change in a model input?

Now that your model is running, you can use it to find out how changes in the inputs affect the outputs (and your report to the Caliph).

1. Census. Do we need a better census?
2. Nesting sites. Do we need to more adequately count nesting sites?

3. Clutch size. Do we need a better estimate of clutch size?
4. Bullets. What about the number of silver bullets?

To do this in R, we need to set up a series of options for numbers of bullets, and a series of options for clutch size. We'll create a matrix that will store these options as row and column headers to begin:

```r
# create a vector of bullet sizes
bullet.opts <- seq(from = 0, to = 30, by = 10)

# create a vector of clutch sizes
clutch.opts <- seq(from = 0.8, to = 1.3, by = 0.1)

# set up a matrix to hold the results
results <- matrix(data = 0,
                  nrow = length(clutch.opts),
                  ncol = length(bullet.opts),
                  dimnames = list(clutch.opts, bullet.opts))

# look at this so far
results
```

```
##     0 10 20 30
## 0.8 0  0  0  0
## 0.9 0  0  0  0
## 1   0  0  0  0
## 1.1 0  0  0  0
## 1.2 0  0  0  0
## 1.3 0  0  0  0
```

```r
# look at the rownames (characters, not numbers)
rownames(results)
```

```
## [1] "0.8" "0.9" "1"   "1.1" "1.2" "1.3"
```

```r
# look at the column names (characters, not numbers)
colnames(results)
```

```
## [1] "0"  "10" "20" "30"
```

Let's quickly review what a matrix is in R. It is a two-dimensional object that stores just one type of data. Here, our matrix will store numbers. The columns represent alternative number of bullets, and the rows represent alternative clutch sizes. To work with this matrix, we will need be able to locate a particular part of the matrix, identified by the matrix row and column in bracket notation with row number first, followed by a comma, then the column number. I always think of **R**ussell **C**row to help remember the order. **R**osemary **C**looney, **RC** Cola, **R**ay **C**harles, and **R**achel **C**arson also work.

matrix.name[r, c]

```
# turn row 3, column 2 to 100 rocs
results[3, 2] <- 100

# look at results
results
```

```
##     0  10 20 30
## 0.8 0   0  0  0
## 0.9 0   0  0  0
## 1   0 100  0  0
## 1.1 0   0  0  0
## 1.2 0   0  0  0
## 1.3 0   0  0  0
```

You can also do this by name if you'd like:

```
# turn row 1, column 4 to 10 rocs
results["0.8", "30"] <- 10

# look at results
results
```

```
##     0  10 20 30
## 0.8 0   0  0 10
## 0.9 0   0  0  0
## 1   0 100  0  0
## 1.1 0   0  0  0
## 1.2 0   0  0  0
## 1.3 0   0  0  0
```

These will be over-written shortly.

Next, we'll call our `popRoc()` function many times to fill in the grid. We'll do this with two loops. The outer loop will focus on changing the clutch size, and the inner loop will focus on changing the number of bullets.

```
# run the model under each of these conditions.

for (i in 1:length(clutch.opts)) {
  for (j in 1:length(bullet.opts)) {

    # run the model and pass in the bullet and clutch as inputs
    rocs <- popRoc(bullets = bullet.opts[j],
                   clutch.size = clutch.opts[i],
                   starting.pop = 700,
                   plot.traj = FALSE)

    # add the last population size to our results matrix
    results[i, j] <- max(0, tail(rocs, n = 1))

  } # end of bullet option
} # end of clutch option
```

```
# look at result matrix (similar to Excel's data table output)
results
```

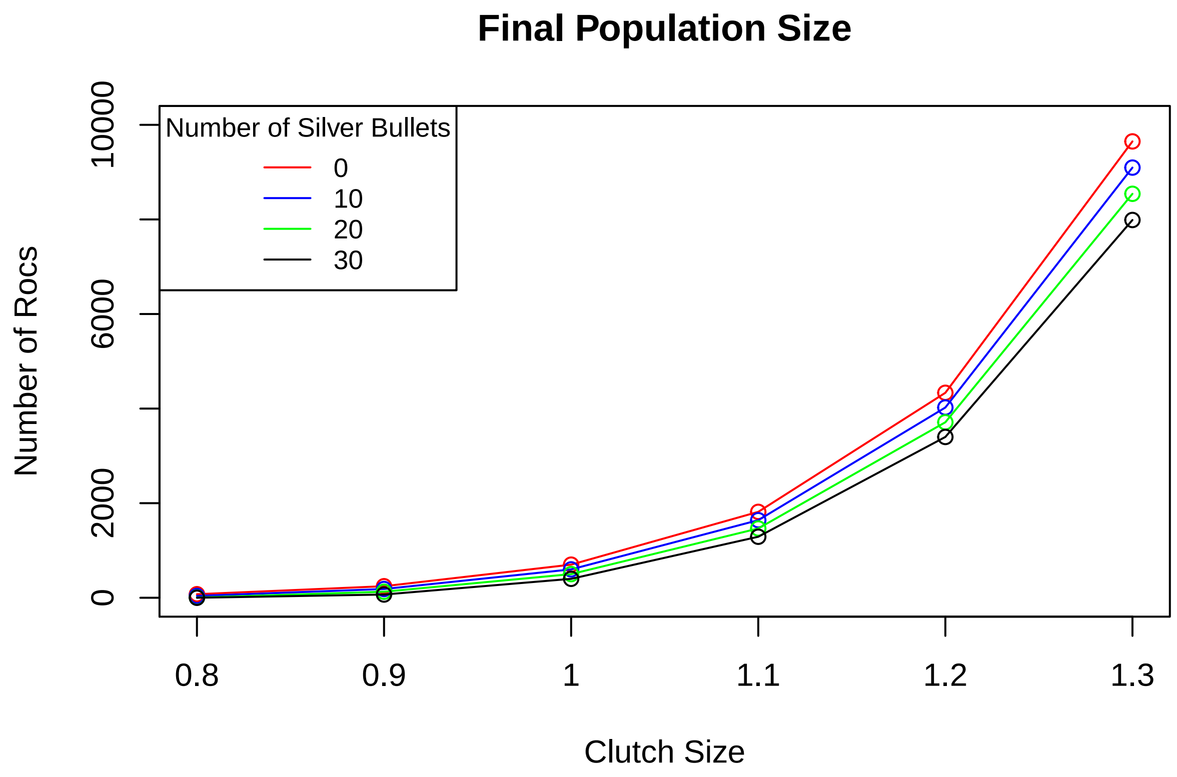
```
##               0          10          20         30
## 0.8   75.16193   39.45689    3.751862    0.00000
## 0.9  244.07491  185.45597  126.837027   68.21809
## 1    700.00000  600.00000  500.000000  400.00000
## 1.1 1815.61972 1640.30805 1464.996381 1289.68471
## 1.2 4334.21550 4022.71131 3711.207125 3399.70294
## 1.3 9650.09443 9096.04096 8541.987500 7987.93404
```

```
# now you can round the results (since the rounding wasn't used in calculations)
(results <- round(results, digits = 0))
```

```
##        0   10   20   30
## 0.8   75   39    4    0
## 0.9  244  185  127   68
## 1    700  600  500  400
## 1.1 1816 1640 1465 1290
## 1.2 4334 4023 3711 3400
## 1.3 9650 9096 8542 7988
```

There a many ways to plot results in R. Many prefer the packages **lattice** or **ggplot2**. Here, I'll use the plain vanilla `plot()` function in R to create the sensitivity results that mimic the spreadsheet chart.

```
# https://www.tutorialspoint.com/r/r_line_graphs.htm
plot(results[,1],type = "o", col = "red",
     xlab = "Clutch Size", ylab = "Number of Rocs",
     ylim = c(0, 10000),
     xaxt = "n",   # suppress the default x axis label
     main = "Final Population Size")

# add in the x axis labels
axis(1, at = 1:6, labels = clutch.opts)

# add in other lines
lines(results[,2], type = "o", col = "blue")
lines(results[,3], type = "o", col = "green")
lines(results[,4], type = "o", col = "black")

# Add a legend to the plot
legend("topleft", title = "Number of Silver Bullets",
       legend = as.character(bullet.opts),
       col = c("red", "blue", "green", "black"),
       lty = 1, cex = 0.8)
```



The next portion of our Excel model is the threshold graph, which we filled in with Excel's Goal Seek function. In R, we can use the `optim()` function to do this. We will introduce this function in a different exercise.

Reflection

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

- What is modeling?
- What role does the modeling vehicle play?
- How do you get from your model world back to the real world?