Elephant Age-Based Population Model with Management

Therese Donovan

2020-03-18

Overview

This script is a translation of the elephant age-based population model with darting we did with spreadsheets. The model is a *female-only* population model that has a post-breeding census. That is, we count individuals immediately after the birthday (and all animals are assumed to be born on the same day in a birth pulse). Thus, our count includes 0, 1, 2, . . . 59 year olds. We assume that all individuals that are age 12 or older can reproduce. We will be modifying the elephant function we made last week, so let's start with a quick review.

The Elephant Model ——

Our Elephant model will be a function called `popElephant()`. We certainly aren't required to build our model as a function. We could just have a script where you create new objects at the top of your script, and then use those objects later in the script. However, we already know that our model has inputs (shaded green in the spreadsheet) and outputs (shaded blue in the spreadsheet), and that we want to be able to quickly use our model to perform experiments with it. A function is perfect for that.

If you recall, our model has several inputs. Here, we will let the `popElephant()` function have six arguments: *starting.pop* (a vector of starting female population sizes by age class), *ci* (the calving interval; the number of years between subsequent births), *offspring.sex.ratio* (the proportion of the offspring that are female), *s.calf* (the survival rate of calves), *s.adult* (the survival rate of adults), , and *plot.traj* (TRUE or FALSE; should a plot of the population trajectory be displayed). 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 popElephant.
# it returns the vector of total population sizes of Rocs with each time step.
# never forget to look for each newly created object in the environment!

popElephant <- function(starting.pop,
                        ci,
                        offspring.sex.ratio,
                        s.calf,
                        s.adult,
                        plot.traj = TRUE){

  # convert the calving interval to a fecundity rate
  fecundity = 1/ci

  # create a matrix that will store the population size by age
  # the rows are the years
  # the columns give the age classes from 0 to 59
  pop <- matrix(data = NA, nrow = 80, ncol = 60)
```

```
  # add column names to avoid confusion
  colnames(pop) <- as.character(0:59)

  # set the initial population vector
  pop[1,] <- starting.pop

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

      # create a vector of survival rates
      # the first entry is the calf survival rate
      # the next 59 entries are the adult survival rates for age classes 1:60
      # the final adult age class has a survival rate of 0; they drop out of the model
      survival.rate <- c(s.calf, rep(s.adult, times = 58), 0)

      # compute survivors in next time step; notice this is length 61
      survivors <- pop[y - 1, ] * survival.rate

      # add the survivors to the population matrix; notice the offset here
      pop[y, 2:60] <- survivors[1:59]

      # add in  the total births; 27.871
      indices <- match(x = as.character(12:59), table = colnames(pop))
      pop[y, "0" ] <- sum(pop[y, indices]) * fecundity * offspring.sex.ratio

    } # end of loop

  # summarize the results
  total.pop <- rowSums(pop, na.rm = T)

  # get growth rate for each year:  lambda = n(t+1)/n(t)
 lambda <- total.pop[-1]/head(total.pop, -1)

  # plot a line graph
  if (plot.traj == TRUE) {
  plot(x = 1:nrow(pop), y = total.pop,
       type = "b",
       col = "blue",
       xlab = "Year", ylab = "Number of Female Elephants",
       ylim = c(0, max(total.pop) + 10),
       main = "Total Female Elephant Population Size over Time")
 }

  # return the vector of population sizes
  return(list(total.pop = total.pop, lambda = lambda))

} # end of function
```

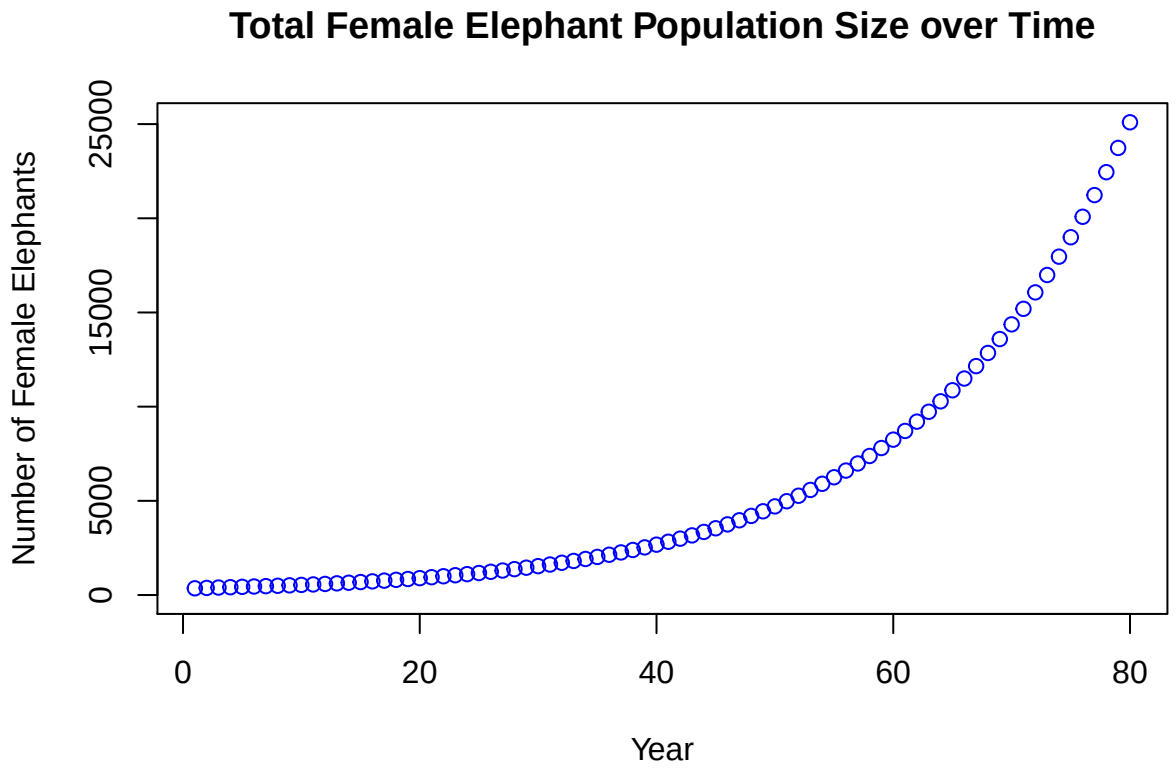
The starting population vector is something that the user of the function (you!) will need to provide. You might recall that we our spreadsheet model started with 40 0-year-olds, 20 individuals for age classes 1-5, and 4 individuals for age classes 6-60. Hand-typing 61 entries can be time-consuming. Here, we'll make use of the `rep()` function to repeat values a certain number of times, like this:

```
c(40, rep(20, times = 5), rep(4, times = 54))
```

```
##  [1] 40 20 20 20 20 20  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4
## [26]  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4
## [51]  4  4  4  4  4  4  4  4  4  4
```

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:
popElephant(starting.pop = c(40, rep(20, times = 5), rep(4, times = 54)),
            ci = 3.1,
            offspring.sex.ratio = 0.5,
            s.calf = 0.9,
            s.adult = 0.99,
            plot.traj = TRUE)
```




```
## $total.pop
##  [1]   356.0000   375.5381   395.4545   414.9351   433.9866   452.6155
##  [7]   470.8284   491.0372   512.9595   536.5373   561.7141   588.4342
## [13]   618.9826   651.8632   687.1009   724.6145   764.3245   806.1530
## [19]   850.0239   896.1752   944.8059   996.1054  1050.2539  1107.4227
## [25]  1168.0782  1232.4878  1300.9193  1373.6276  1450.8545  1532.8292
## [31]  1619.7689  1711.9195  1809.5504  1912.9526  2022.4382  2138.3378
## [37]  2261.0398  2390.9633  2528.5580  2674.3022  2828.7011  2992.2843
## [43]  3165.6047  3349.2412  3543.8024  3749.9282  3968.2931  4199.6075
## [49]  4444.6254  4704.1475  4979.0252  5270.1634  5578.5236  5905.1268
## [55]  6251.0552  6606.7653  6983.0396  7381.2004  7802.6443  8248.8457
## [61]  8710.9643  9205.1166  9728.4551 10282.8360 10870.2218 11492.6872
## [67] 12152.4266 12850.1783 13588.0976 14368.4944 15193.8417 16066.7840
## [73] 16988.6069 17962.7978 18992.4372 20080.8226 21231.4799 22448.1756
## [79] 23734.9301 25095.8000
```

```
##
## $lambda
##  [1] 1.054882 1.053035 1.049261 1.045914 1.042925 1.040239 1.042922 1.044645
##  [9] 1.045964 1.046925 1.047569 1.051915 1.053120 1.054057 1.054597 1.054802
## [17] 1.054726 1.054420 1.054294 1.054265 1.054296 1.054360 1.054433 1.054772
## [25] 1.055141 1.055523 1.055890 1.056221 1.056501 1.056718 1.056891 1.057030
## [33] 1.057143 1.057234 1.057307 1.057382 1.057462 1.057548 1.057639 1.057734
## [41] 1.057830 1.057922 1.058010 1.058091 1.058165 1.058232 1.058291 1.058343
## [49] 1.058390 1.058433 1.058473 1.058511 1.058547 1.058581 1.056904 1.056953
## [57] 1.057018 1.057097 1.057186 1.056022 1.056728 1.056853 1.056986 1.057123
## [65] 1.057263 1.057405 1.057417 1.057425 1.057432 1.057441 1.057454 1.057374
## [73] 1.057344 1.057321 1.057306 1.057301 1.057306 1.057321 1.057336
```

Adding Darts

Our next step is to modify this function to including darting, a management activity that can be used to prevent females from breeding. We will need to modify our `popElephant()` model to include this. Our new function will be called `popElephantMgt()`, and it will include just two extra arguments. The first argument will be called "prop.dart", which is the proportion of adult females that are darted (non-breeding) each year.

The only major change to the function code for this new argument occurs in the equation for computing the number of 0 year olds in a given time step. The "old" equation is:

```
pop[y, "0" ] <- sum(pop[y, indices]) * fecundity * offspring.sex.ratio
```

This will need to be replaced by

```
pop[y, "0" ] <- sum(pop[y, indices]) * (1 - prop.dart) * fecundity * offspring.sex.ratio
```

In words, the number of 0 year old femals in time step y is the sum of the adult breeding femals in time step y, multiplied by the proportion that were *not* darted, multiplied by the fecundity rate, and finally multiplied by the offspring sex ratio.

We will add one more argument to our function that will allow us to set the number of years for the simulation. This new argument will be called "years", and it will be a number. The default value will be 80, which is of course what value the function will use if the user does not provide a value for this argument.

The only major change to the function code for this new argument occurs in the equation for setting up the matrix that stores the projection results. The "old" equation is:

```
pop <- matrix(data = NA, nrow = 80, ncol = 60)
```

This will need to be replaced by

```
pop <- matrix(data = NA, nrow = years, ncol = 60)
```

And, one more thing! We will let our new function return the matrix that stores the projection, in addition to the vector of population sizes and the vector of lambda values. We will need this later in the script when we explore how the initial population vector affects our results. The "old" equation is:

```
return(list(total.pop = total.pop, lambda = lambda))
```

This will be replaced by

```
return(list(pop = pop, total.pop = total.pop, lambda = lambda))
```

```r
# create a new function call popElephantMgt.
# it returns the vector of total population sizes of Rocs with each time step.
# never forget to look for each newly created object in the environment!

popElephantMgt <- function(starting.pop,
                           ci,
                           offspring.sex.ratio,
                           s.calf,
                           s.adult,
                           prop.dart,
                           years = 80,
                           plot.traj = TRUE){

  # convert the calving interval to a fecundity rate
  fecundity = 1/ci

  # create a matrix that will store the population size by age
  # the rows are the years
  # the columns give the age classes from 0 to 59
  pop <- matrix(data = NA, nrow = years, ncol = 60)

  # add column names to avoid confusion
  colnames(pop) <- as.character(0:59)

  # set the initial population vector
  pop[1,] <- starting.pop

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

      # create a vector of survival rates
      # the first entry is the calf survival rate
      # the next 59 entries are the adult survival rates for age classes 1:60
      # the final adult age class has a survival rate of 0; they drop out of the model
      survival.rate <- c(s.calf, rep(s.adult, times = 58), 0)

      # compute survivors in next time step; notice this is length 61
      survivors <- pop[y - 1, ] * survival.rate

      # add the survivors to the population matrix; notice the offset here
      pop[y, 2:60] <- survivors[1:59]

      # add in  the total births; 27.871
      indices <- match(x = as.character(12:59), table = colnames(pop))
      pop[y, "0" ] <- sum(pop[y, indices]) * (1 - prop.dart) * fecundity * offspring.sex.ratio

    } # end of loop

  # summarize the results
  total.pop <- rowSums(pop, na.rm = T)

  # get growth rate for each year:  lambda = n(t+1)/n(t)
 lambda <- total.pop[-1]/head(total.pop, -1)
```

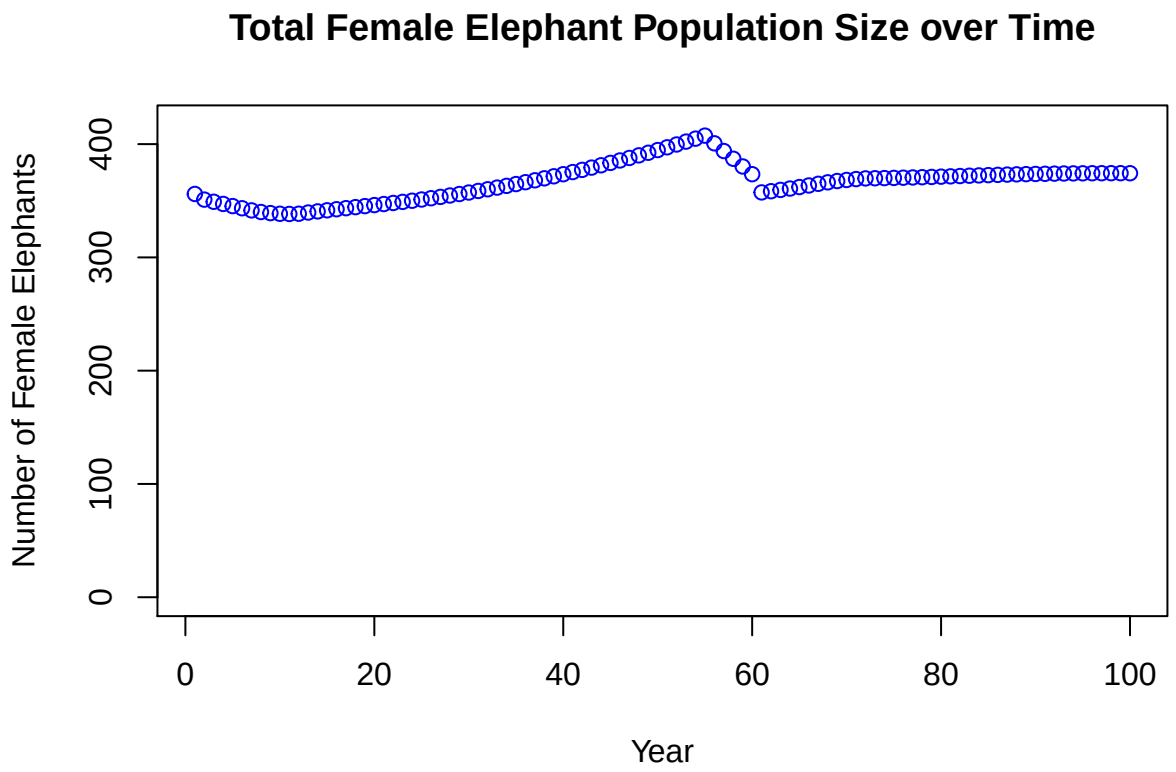
```
  # plot a line graph
  if (plot.traj == TRUE) {
  plot(x = 1:nrow(pop), y = total.pop,
       type = "b",
       col = "blue",
       xlab = "Year", ylab = "Number of Female Elephants",
       ylim = c(0, max(total.pop) + 10),
       main = "Total Female Elephant Population Size over Time")
 }

  # return the results as a named list
  return(list(pop = pop, total.pop = total.pop, lambda = lambda))

} # 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:
test <- popElephantMgt(starting.pop = c(40, rep(20, times = 5), rep(4, times = 54)),
            ci = 3.1,
            offspring.sex.ratio = 0.5,
            s.calf = 0.9,
            s.adult = 0.99,
            prop.dart = 0.8,
            years = 100,
            plot.traj = TRUE)
```




```
# look at the structure of the returned result
str(test)
```

```
## List of 3
##  $ pop      : num [1:100, 1:60] 40 6.13 6.07 6.01 5.95 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : NULL
##   .. ..$ : chr [1:60] "0" "1" "2" "3" ...
##  $ total.pop: num [1:100] 356 351 349 347 345 ...
##  $ lambda   : num [1:99] 0.986 0.995 0.995 0.995 0.994 ...
```

The function appears to be working as expected (given that we know what the results should be from our spreadsheet exercise). The function returns a named list. The first element is called "pop" and contains a matrix (here, with 100 rows and 60 columns). The second element is named "total.pop" and is a vector of length 100 that stores numbers. The third element is named "lambda" and is a vector of length 100 that stores numbers.

Examining the Effects on Age Structure

We've mentioned several times that the initial age structure (in the first year of simulation) can dramatically affect λ values in the short term. Eventually, with enough years, the long term λ values stabilize to a given value (known as the *asymptotic lambda*). With a Leslie matrix, you can actually compute the stable age distribution analytically (with linear algebra). Here, though, we will use simulation. We'll run the model *without* darting for 500 years, and then compute the proportion of individuals in each age class in the 500th year.

```
# run the model with no darting for 500 years; save the output
no.darts <- popElephantMgt(starting.pop = c(40, rep(20, times = 5), rep(4, times = 54)),
            ci = 3.1,
            offspring.sex.ratio = 0.5,
            s.calf = 0.9,
            s.adult = 0.99,
            prop.dart = 0,
            years = 500,
            plot.traj = FALSE)

# look at the structure of the returned result
str(no.darts)
```

```
## List of 3
##  $ pop      : num [1:500, 1:60] 40 30.7 30.4 30 29.7 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : NULL
##   .. ..$ : chr [1:60] "0" "1" "2" "3" ...
##  $ total.pop: num [1:500] 356 376 395 415 434 ...
##  $ lambda   : num [1:499] 1.05 1.05 1.05 1.05 1.04 ...
```

To compute the stable age distribution (knowing the population growth rate without darting stabilizes in 500 years), we can do something like shown below. The `tail()` function will grab the end of a vector, matrix, or dataframe, returning the number of observations you specify with the "n" argument:

```
# extract the last record of the matrix with the tail function
final.pop <- tail(x = no.darts$pop, n = 1)

# divide final.pop by total pop in the la
age.distribution <- final.pop/tail(x = no.darts$total.pop, n = 1)

# look the stable age distribution for no darts
age.distribution
```

```
##                 0          1          2          3          4          5
## [500,] 0.07100508 0.06043701 0.05658603 0.05298043 0.04960457 0.04644382
##                 6          7          8          9         10         11
## [500,] 0.04348447 0.04071368 0.03811945 0.03569052 0.03341635 0.0312871
##                12         13         14         15         16         17
## [500,] 0.02929352 0.02742697 0.02567935 0.02404309 0.02251109 0.0210767
##                18         19         20         21         22         23
## [500,] 0.01973372 0.01847631 0.01729902 0.01619674 0.0151647 0.01419842
##                24         25         26         27         28          29
## [500,] 0.01329371 0.01244665 0.01165356 0.01091101 0.01021577 0.009564835
##                30          31          32          33          34          35
## [500,] 0.008955373 0.008384746 0.007850479 0.007350255 0.006881904 0.006443396
##                36          37          38          39          40          41
## [500,] 0.00603283 0.005648424 0.005288513 0.004951534 0.004636028 0.004340625
##                42          43          44          45          46          47
## [500,] 0.004064045 0.003805088 0.003562632 0.003335625 0.003123082 0.002924083
##                48          49          50          51          52          53
## [500,] 0.002737763 0.002563316 0.002399984 0.002247059 0.002103879 0.001969822
##                54         55          56          57          58         59
## [500,] 0.001844307 0.00172679 0.001616761 0.001513742 0.001417288 0.00132698
```

This vector gives the proportion of total population that is assigned to each age class. Let's see how this changes with darting. Here, we'll run the model again for 500 years, and set the proportion of darted animals to 0.8 (80% of females are unavailable to reproduce each year):

```
# run the model with no darting for 500 years; save the output
with.darts <- popElephantMgt(starting.pop = c(40, rep(20, times = 5), rep(4, times = 54)),
             ci = 3.1,
             offspring.sex.ratio = 0.5,
             s.calf = 0.9,
             s.adult = 0.99,
             prop.dart = 0.8,
             years = 500,
             plot.traj = FALSE)

# extract the last record of the matrix with the tail function
dart.final.pop <- tail(x = with.darts$pop, n = 1)

# divide final.pop by total pop in the la
dart.age.distribution <- dart.final.pop/tail(x = with.darts$total.pop, n = 1)

# look the stable age distribution for no darts
dart.age.distribution
```

```
##                 0          1          2          3          4         5
## [500,] 0.02413816 0.02172771 0.02151377 0.02130194 0.02109219 0.0208845
##                 6          7          8          9         10         11
## [500,] 0.02067887 0.02047525 0.02027364 0.02007402 0.01987636 0.01968065
##                12         13       14         15         16         17
## [500,] 0.01948686 0.01929499 0.019105 0.01891688 0.01873062 0.01854619
##                18         19         20         21         22         23
## [500,] 0.01836357 0.01818276 0.01800372 0.01782645 0.01765092 0.01747712
##                24         25         26         27         28         29
## [500,] 0.01730503 0.01713464 0.01696592 0.01679887 0.01663346 0.01646968
##                30         31         32         33         34         35
```

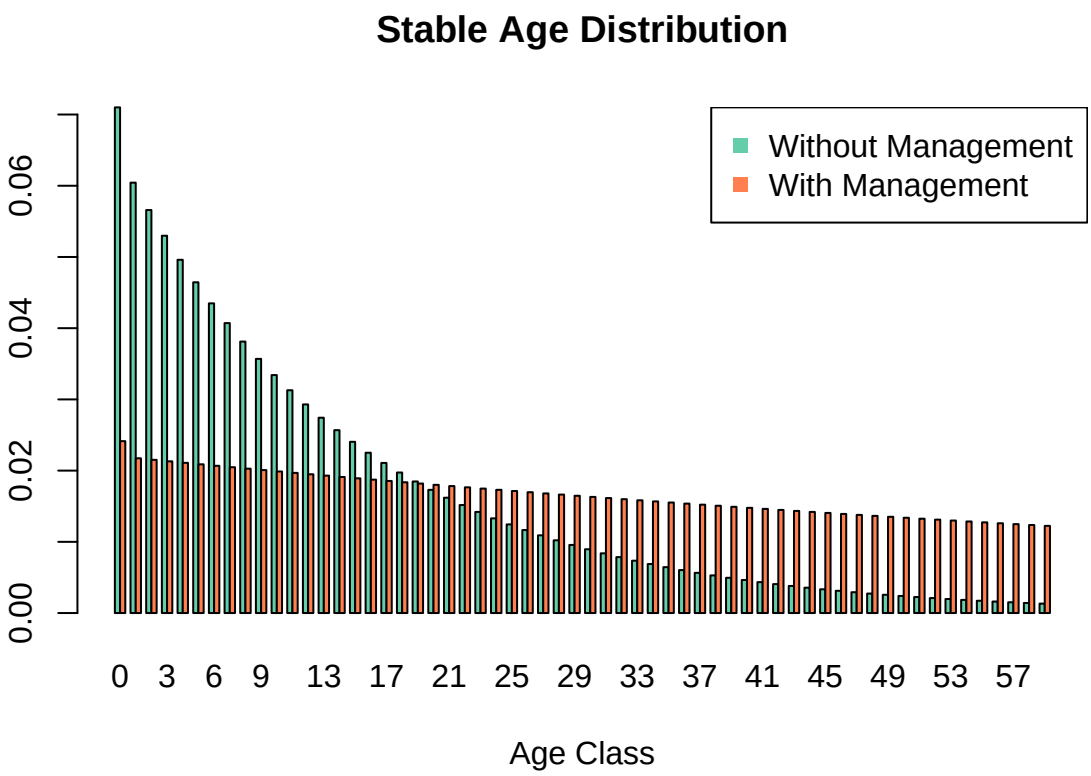
```
## [500,] 0.01630751 0.01614694 0.01598795 0.01583053 0.01567465 0.01552031
##                36         37         38       39         40         41
## [500,] 0.01536749 0.01521617 0.01506635 0.014918 0.01477111 0.01462567
##                42         43         44         45         46         47
## [500,] 0.01448165 0.01433906 0.01419787 0.01405807 0.01391965 0.01378259
##                48         49         50         51         52         53
## [500,] 0.01364688 0.01351251 0.01337946 0.01324772 0.01311727 0.01298812
##                54         55         56         57         58         59
## [500,] 0.01286023 0.0127336 0.01260822 0.01248407 0.01236115 0.01223944
```

Shall we compare the two stable age distributions with a graph? Let's do it with the `barplot()` function. The two distributions can be plotted side by side by setting the "beside" argument to TRUE:

```
help(barplot)
```

```
barplot(rbind(age.distribution, dart.age.distribution),
        beside = TRUE,
        col = c("aquamarine3","coral"),
        names.arg = as.character(0:59),
        main = "Stable Age Distribution",
        xlab = "Age Class",
        ylab = "")

# add the legend
legend("topright",
       legend = c("Without Management","With Management"),
       pch = 15,
       col = c("aquamarine3","coral"))
```



The bottom line is that management activities can dramatically change the population's age structure, both in the short and in the long term. As Tony mentioned, this might affect elephant behavior and have social

implications as well.

Goal Seek in R?

In our spreadsheet model of elephant population management, we used Goal Seek to find the proportion of females that need to be darted in order to keep the population's growth rate at 1.

Is there a way to do this in R? A quick Google search shows us an example:

https://stackoverflow.com/questions/28575395/back-solving-a-function-or-goal-seek-in-r

Take a moment to read this thread now (Stack Overflow in general is a great resource for R users). We'll follow the thread that talks about the `optimize()` function in R.

```
help(optimize)
```

To illustrate this, the Stack Overflow writer provides this example (modified slightly here for clarity):

Let's create a function named "f". In this function, the user inputs an argument called "x". The function will then compute

```
f <- function(x) {

  # create a vector that uses the input, x
  my.vector <- c(100, 30 * c(x, 2:50))

  # compute the cumulative sum of the vector
  cum.sum <- cumsum(my.vector)

  # return the last entry of the cum.sum object
  tail(cum.sum, n = 1)
}
```

Let's test this function:

```
# set x to 2
f(x = 2)
```

```
## [1] 38380
```

```
# set x to 100
f(x = 100)
```

```
## [1] 41320
```

OK, suppose we want to find the value of "x" such that the result will be **48010**. How do we do this in R? Another Stack Overflow user on this same thread provides us with a golden nugget:

> You can optimize any function by rewriting y = f(x) to f(x)-y and finding the root of that. f(x) returns some value. You want that value to equal target_value. Write f2 <- function(x) f(x) - target_value and run optim() on that. – Carl Witthoft

That's the answer! Thank you Carl, whoever you are! We will need to create a new function (here, we'll call it "new.f"). This function will include a single argument (x), that will in turn feed into our "f" function, and then subtract the value we wish to seek (48010). The absolute value is returned:

```
# create a new function called new.f
new.f <- function(x) {
  abs(f(x) - 48010)
}
```

Next, we use the `optimize()` function. Definitely look at this function's help page before you plow ahead.

We need to specify the name of the function that we wish to optimize (here, new.f), and provide the upper and lower bounds for the function's single input (x). We also specifiy whether we want to minimize the function or maximize it. Here, we wish to find a value for x such that the result from our function, "f" - 48010 is a minimum (close to 0).

```
# run the optimize function to
optimize(new.f, lower = -1000, upper = 1000, maximum = FALSE)
```

```
## $minimum
## [1] 323
##
## $objective
## [1] 0.0007025907
```

Whoa! `optimize()` returned a list with two elements. The first is named "minimimum" and this provides the solution (i.e., the goal seek solution). The second is named "objective" and this provides you with an indication of how far you are from your desired answer of 48010.

Let's see if this is the correct answer:

```
# plug in 323 to our function, f
f(x = 323)
```

```
## [1] 48010
```

Voila! The Stack Overflow post indicates what is is happening (modified to fit our example):

> "R will search [-1000, 1000] in an attempt to find a value, x, that minimizes the absolute value of f(x) - 48010. In this case, it finds 323, for which f(323) returns 42010 and so abs(f(x) - 48010) returns 0.

How can we apply this to elephant management? To keep things as simple as possible, we will first re-write our `popElephantMgt()` function so that it contains only one user input, "prop.dart". All of the other entries will be fixed, so we'll just hard-code them into the function. We'll name this new function `dartMgt()`, and this function will return only one thing: the final lambda value of the projection.

```
# create a new function call dartMgt

dartMgt <- function(prop.dart){

  # create a matrix that will store the population size by age
  pop <- matrix(data = NA, nrow = 100, ncol = 60)

  # add column names to avoid confusion
  colnames(pop) <- as.character(0:59)

  # set the initial population vector
  pop[1,] <- c(40, rep(20, times = 5), rep(4, times = 54))

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

      # create a vector of survival rates
      survival.rate <- c(0.9, rep(0.99, times = 58), 0)

      # compute survivors in next time step; notice this is length 61
      survivors <- pop[y - 1, ] * survival.rate
```

```
      # add the survivors to the population matrix; notice the offset here
      pop[y, 2:60] <- survivors[1:59]

      # add in  the total births; 27.871
      indices <- match(x = as.character(12:59), table = colnames(pop))
      pop[y, "0" ] <- sum(pop[y, indices]) * (1 - prop.dart) * 1/3.1 * 0.5

    } # end of loop

  # summarize the results
  total.pop <- rowSums(pop, na.rm = T)

  # get growth rate for each year:  lambda = n(t+1)/n(t)
  lambda <- total.pop[-1]/head(total.pop, -1)

  # identify your target value
  final.lambda <- tail(lambda, n = 1)

  # return the results as a named list
  return(final.lambda)

} # end of function
```

OK, now we have our function called `dartMgt()`. This is like our function, "f" in the example above. Let's test it:

```
dartMgt(prop.dart = 0.4)
```

```
## [1] 1.036753
```

```
dartMgt(prop.dart = 0.6)
```

```
## [1] 1.022105
```

Remember, the function returns the lambda value (λ) for the final year of the simulation. We wish to find a value for "prop.dart" such that the final lambda value from our function, `dartMgt()` is 1. That is, we want the population growth rate to be exactly 1 (or as close to 1 as possible). Following the Stack Overflow example, we will create a new function called `goalDart()`, which will have a single input (the parameter we wish to fiddle with). This function will pass its input to the `dartMgt()` function, compute the output, and then find the absolute value of the difference between this output and our desired target (the number 1):

```
# create a new function called goalDart
goalDart <- function(prop.dart) {
  abs(dartMgt(prop.dart) - 1)
}
```

Next, we use the `optimize()` function. We need to specify the name of the function that we wish to optimize (here, `goalDart()`), and provide the upper and lower bounds for the function's single input ("prop.dart"). Remember that "prop.dart" is a proportion, so the lower value should be 0 and the upper value should be 1. We also specifiy whether we want to minimize the function or maximize it. Here, we wish to find a value for "prop.dart" such that the result from our function, "dartMgt" - 1 is a minimum (close to 0).

```
# run the optimize function to find the value of "prop.dart" that renders lambda = 1
optimize(goalDart, lower = 0, upper = 1, maximum = FALSE)
```

```
## $minimum
## [1] 0.799427
```

```
##
## $objective
## [1] 6.073374e-08
```

Whoa again! `optimize()` returned a list with two elements. The first is named "minimimum" and this provides the solution (i.e., the goal seek solution). The second is named "objective" and this provides you with an indication of how far you are from your desired answer of 1.

Let's see if this is the correct answer:

```
# plug in 0.799427 to our function, dartMgt
dartMgt(prop.dart = 0.799427)
```

```
## [1] 0.9999999
```

Voila! In summary, you can use the `optimize()` function in R if you are looking for an R version of Excel's Goal Seek tool. There are other alternatives, too. We will use `optimize()` in our decision models later in the course.