A Field Guide to Programming:
Population Growth in R

Christopher Stieha, Kathryn Sullivan Montovan and Derik Castillo-Guajardo
[stieha@hotmail.com, kmontovan@bennington.edu, derik.cg@gmail.com]

1. Thinking about Programming

Programming is like visiting a big city for the first time. At your hotel, you stop and ask for directions to the museum. The clerk tells you to go outside, turn left, walk straight for a while, then turn right. If you follow those directions, you will get to the museum. However, you decide that you will first go right, then walk straight, and then turn left. You think that since you are following the same directions you will get to the same place. You won't. You may end up at the crematorium. The order of the directions is just as important as the directions themselves.

But within those simple directions, you have many things going on. For example, if you cross a road while walking straight, you need to check to make sure no cars are coming. You will also need to dodge other people. You also need to put your right foot on the ground and pickup your left foot. You move the left foot forward in space a little bit and then place it back on the ground. Lift up the right foot, move it forward in space, and place it back on the ground.

All these parts are required in programming. You need to first envision the larger picture and define your main goal (in this case getting to the museum). You need to break this larger goal into smaller goals (such as going right, walking straight, or turning left), and continue to break these smaller goals into chunks that are easily programmed. It is important to remember that things have to be done in order, as if you turn right first instead of left or you lift up your left foot before putting down your right foot you will have problems.

So once you get a problem, you have to break the problem into its principle components. You write down what you know; you figure out what you need to know. You need to think broadly at first to figure out what you have to do to get between the two. Suppose you want to buy two pads of paper and a pencil and you want to know how much everything will cost. You know that a pad of paper costs a dollar and a pencil costs five dollars (a very nice mechanical one). We can break this down to:

```
Cost_of_paper = 1.00
Cost_of_pencil = 5.00
Number_of_paper = 2
Number_of_pencils = 1
```

What we want to know is:

```
How_much_will_it_cost = ?
```

Looking at this, you immediately say it will cost seven dollars. But think what you did to get that. First you had to take the number of pads you want to buy and multiple it by the cost (2 dollars total). Then you said one pencil costs five dollars (1 times 5 = 5). You then added these together to get seven dollars. You didn't realize you took all those steps, but trust me, you did.

That's a very simple case, but every problem can be broken down in to parts, which can be broken down in to smaller problems, which end up looking very similar to what we just did.

Note that if no one told us how much the things cost, we never could have figured the problem out. Or what if there was a ten percent sale on paper? We would first want to figure out the new price of a pad of paper, then figure out the total price. In this case, we had to figure out a minor (but important) piece of information before we could continue on and answer our big problem.

This is the basic idea of programming: taking the big problem, figuring out what the smaller "problems" are, and solving those in order.

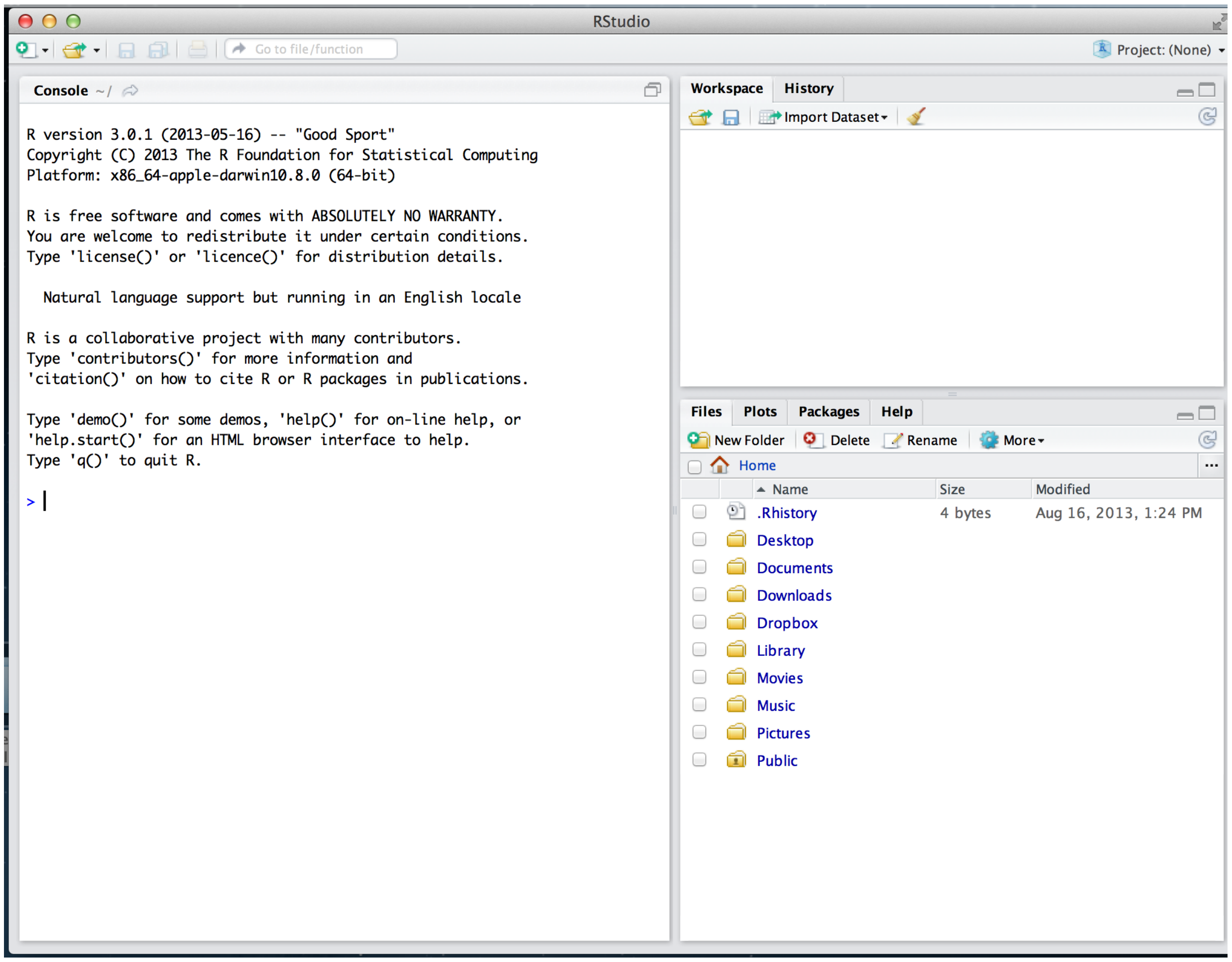
During this discussion, we will be focusing on programming in the R programming language (www.cran.r-project.org) with a slant towards population biology. This is to get you programming with a known function from Ecology, but once the basics are learned, any equations can be used. This discussion does not assume you can program, but instead teaches you how to think about programming and shows you that there are many ways to do the same thing.

2. Variables and Mathematical Operations

Getting and installing R: The main sources for R are CRAN (www.cran.r-project.org) and its mirrors. You can get the source code, but most users will prefer a precompiled version. To get one from CRAN, click on the link for your OS, continue to the folder corresponding to your OS version, and find the appropriate download file for your computer. For windows machines you should read the whole website – it will tell you what base to download. For Windows or OS X, R is installed by launching the downloaded file and following the on-screen instructions. At the end you'll have an R icon on your desktop that can be used to launch the program. Installing versions for LINUX or UNIX is more complicated and idiosyncratic (which will not bother the corresponding users), but many LINUX distributions include a fairly up-to-date version of R. For Windows PCs it is very strongly suggested that you edit the file Rconsole in R's etc folder and change the line MDI=yes to MDI=no, and also edit Rprofile to un-comment the line options (chmhelp=TRUE) by removing the # at the start of the line. These changes allow R's command and graphics windows to move independently on the desktop, select the most powerful version of the help system, and reduce the rate of inexplicable crashes on Windows laptops.

We also recommend that you install and use Rstudio. Rstudio does not change any of the commands or functions for R, but does put all the windows you will need into a nicely formatted window. This may seem silly, but it really helps keep things where you can find them more easily. To download, go to http://www.rstudio.com/ide/download/ after downloading R and download and install the appropriate version.

Now that you have a copy of R and Rstudio installed on your computer, let's move on to using the program. Upon opening Rstudio, you will see that the screen is divided into three windows. On the left is the console window. By clicking in this window and typing, whatever you type will appear after the > sign (called prompt). Hitting `ENTER` will cause R to execute whatever you typed. In this introduction, when you, the user, type something, we will use the `Courier type font`. Answers returned from R and items reference from the computer code will also be in `Courier type font`.



```
R version 3.0.1 (2013-05-16) -- "Good Sport"
Copyright (C) 2013 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin10.8.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> 
```

Figure 1: The Rstudio window divided into three parts

In the upper right of Rstudio (see Figure 1), you can choose between the workspace and History. In the lower right you can view the *Files*, *Plots*, *Packages*, and *Help*. We will discuss these in more detail later.

We begin by exploring the simple calculator properties of R.

```
> 1 + 1
```

This is simple addition. After hitting enter, R should give you the result:

```
[1] 2
```

which is telling you that your answer to the input is 2.

```
> 1 - 1
> 1 / 10
> 2 * 2
```

These three operations are subtraction(-), division(/), and multiplication(*) respectively. The first answer returned will be zero, the second answer will be 0.1, and the third will be four.

```
> 2 ^ 3
```

The caret symbol (^) denotes a number raised to the power. In this case 2 is raised to the third power which is `2*2*2` equals eight.

```
> abs(-11)
```

`abs( )` is a function that takes the absolute value of the number inside the parentheses (-11) and returns the positive size of the number: 11.

```
> 10 %% 3
```

The double percent is R's symbol for modulus. It takes the first number, in this case ten, divides it by the second number (three) and returns the remainder. After typing the above operation into R and hitting enter, you should get the value `1`. This operation is useful for truncating values into a numerical framework that cycles. The best example of this is time. Think of a wall clock. At 12 o'clock, the hour hand is at 12. If you look at the clock one hour later, the hour hand is at one o'clock, not thirteen o'clock. If instead of one hour later, you look at the clock 47 hours later, the hour hand will be at `47 %% 12` which will give you 11 o'clock. If you instead look at the clock 48 hours later, the clock will again be at 12 o'clock because 48 %% 12 gives you an answer of zero. When you use modulus, it is important to determine whether zero is a viable number or if you have to treat zero as a special case. In this example, every time you see zero from the modulus operation, it means 12 o'clock.

You can combine operations into a single line. Just like you learned in algebra, R follows the order of operations. Parenthesis and functions (such as abs() from above) are evaluated first, then the caret, followed by division and multiplication, and finally addition and subtraction.
For example:

```
> 4 - 2 / 10
```

gives an answer of 3.8. However,

```
> (4 - 2) / 10
```

gives an answer of 0.2 which are completely different answers. When dealing with any complicated equation, we recommend a liberal use of parentheses to avoid calculation errors.

You can also enter multiple equations on a single line by placing semicolons between expressions. For example:

```
> 3/100; 3^2; abs(-1.2)
[1] 0.3
[1] 9
[1] 1.2
```

The answers to all three inputs are returned in the same order as the expressions were entered. A common error is to forget the semicolons between expressions. Run the above command again, this time forgetting the semicolons. R will output `Error: unexpected numeric constant in "3/100 3"`, telling you that it is confused and doesn't know how to handle the number 100 3 (notice the space in between the 100 and the 3).

As a simple calculator, R is very complicated. Now let's explore something more interesting. Suppose you are studying the growth of an invasive species. After some field work, you determine the growth rate is 2 (the number of individuals doubles every time step). We will focus on the simplest case where nothing eats the invasive species, there is no immigration or emigration, and the generations do not overlap.

On the R command line, type `r = 2` followed by `ENTER`. This should give you

```
> r = 2
```

R returns nothing when defining variables. If you look in the workspace in the upper right corner of the window you will now see

```
r       2
```

This confirms that the variable `r` has correctly been assigned the value `2`. It may seem boring, but for complex algebraic expressions, this is useful. Now in this session of R, you can type in `r` at anytime and R will know to use the value 2. We have defined `r` as a variable that will be used as a parameter in a mathematical function. If we choose to, we can make `r` be a different number and R will automatically use that new number where ever it sees `r`. If you read other people's code, you may see `r <- 2`, which also assigns the number 2 to the variable `r`. For this document, we will use the equal sign, but there are subtle differences between = and `<-` that are beyond the scope of this document.

Please note that `r` is very different than `R` and that R is case sensitive and makes a distinction between the two. In naming your variables, always start with a letter (upper or lowercase). Numbers are allowed after the letter, as well as periods. There are some words that you cannot use as variable names such as exp, log, mod, etc, known as reserved names. The maximum length of a name is 63 characters. Be sure to make your variable names concise and understandable as you will be typing them out many times. Also, do not confuse capital o and zero as well as 1 and lower case L.

To help simplify life and expedite the learning process, we are going to create a R script file. Without a script, if we wanted to run a command multiple times, we would have to input it into the command prompt multiple times. This is easy for small single line commands, but when we want to run many commands one after another, this gets tedious. Script files allow us to write larger programs once and be able to run them multiple times or modify them between runs. To create a script file, go to the *File* tab, click on *New* then *R script*. This will make the Console smaller so that the script window can fit in the upper left corner of the screen. We will refer to this as the script file window. In the script file window, we can type as many commands as we want before we have to run them. This allows us to think about our program and work on programming before R wants to give us answers.

First thing we should do is save our file. While in the script file window, click on File and choose "Save as...". Name the file popgrowth.R, navigate to the desired folder for saving your files, and click Save.

On the first line in our new script file, type in `r = 2`. Even though we have already typed this into the console, this will keep the file whole as opposed to us having to write things on the command prompt before we use our script file each time. Pressing ENTER will put the cursor on the next line. While programming, keep each instruction on its own line. This will greatly increase the legibility of your code.

Next, let's define a variable with the population size obtained during our field work. Write the following command on the second line of your script file. Notice that the first name is not no but "n" with a zero.

```
n0 = 20; # initial population size
```

Two things are important in this statement: the semi-colon and the pound sign. The semicolon is used to separate distinct commands on the same line. In this case, the semi-colon serves no purpose other than to visually separate our command from our comment. You can choose whether to include the semicolon in your code, but we find that it helps keep our code cleaner so we typically include it. The pound sign tells R to ignore the rest of the characters to the right. They are very useful to make comments and remind yourself what you are doing. Try to comment just about everything you do in R. As you are writing your program, you may remember everything you are doing and why you did things how you did it, but trust us, you won't in six months or a year. Anyone reading your model will also appreciate comments to help them understand how each line of code functions and fits into the larger model.

In this situation, the computer treats the above operation as

```
n0 = 20
```

From basic ecology, we can determine the population size in the next generation by multiplying the population size of the current generation by the growth rate. Formally, this is the general equation $N_{t+1} = N_t \times r$, where r is the growth rate of the species, t is generation number (often based on years due to yearly cycles or annual species), N_{t+1} is the population size of the next generation, and N_t is the current population size. As we are just beginning to study our population, we are focused on the specific equation $N_1 = N_0 \times r$, where N_0 is our initial population size of 20 stored in the variable `n0`. We store the population size of the next generation in a variable called `n1`; On the third line of our script file, we type:

```
n1 = n0 * r; # population size at t = 1
```

At this point we can highlight all three lines of code and hit command (or ctrl) and enter at the same time to tell R to execute the highlighted commands.

If we go to the command line, we can see the value of `n1` by simply typing `n1` at the command prompt and hitting enter. The variable `n1` should have the value of 40. If we want to compute the population size in the second generation, we multiply the population size of the first generation by the growth rate.
Going back to our script file, we go to the next empty line and type:

```
n2 = n1 * r; # population size at t = 2
```

and hit command-enter (or ctrl+enter) again (while the cursor is on the line you just typed). This will run only the line defining `n2`. Typing `n2` into the command prompt and hitting `ENTER` also gives us the answer.

We can continue doing this for several generations.

```
n3 = n2 * r; # population size at t = 3
n4 = n3 * r; # population size at t = 4
n5 = n4 * r; # population size at t = 5
```

Using this method, we can obtain population sizes of a few generations. We could have typed in the commands individually at the command prompt, but then exploring different possibilities (such as a different growth rate or a different initial population size) would require us to retype in all the commands. By creating a script file, we can simply go to either the first line or second line and change the value of the growth rate, `r,` or the initial population size, `n0`, highlight all lines, and press command(ctrl)-enter to run the script again but with the new values.

If you have saved the script file, you can also run the full script in the console window by going to the File tab and choosing Source File... (in linux this was under the Code tab) In the pop-up window navigate to the saved file (in our case popgrowth.R). This will run the script file without ever opening it. When using, saving, and running script files, make sure R is looking in the correct place for the file.

For a few generations, typing in the specific solution (the `n5 = n4*r;` from above) is tedious but doable. What happens if we want to study a population for 100 years? We would then have to type in equations up to n100, which is doable but not fun. What about 1000 years? There are easier ways to make the computer do all that for us.

3. Vectors

Vectors are collections of elements, such as numbers, arranged in rows or columns. Instead of assigning each year of our population to a new variable (`n0, n1, n2, n3`, and so on), we can actually store all those numbers in one place (such as in the variable `popsize`). In the case of our growing population, we can put all six population sizes in a vector. Even though vectors are variables, we will specifically describe a variable as a vector when it contains more than one number. The convenience of a vector for population size does not stop in having only one variable for population size. Plotting a vector is much easier, which we will discuss later.

Row vectors are entered in R as a collection of numbers separated by commas (or blank spaces) inside of the function `c( )`. They can be given names just like variables with only one element. After running the previously created script file from above, go back to the command prompt and type

```
> popsize = c(n0, n1, n2, n3, n4, n5); popsize; # builds vector popsize with
6 values
```

The vector named `popsize` has six elements. We can see that also by asking the size of a vector or by looking at the workspace window.

```
> length(popsize)
[1] 6
```

The output indicates the number of elements in the vector.

Elements within vectors can be displayed or changed, erased or added. For example, to display the second element of the vector `popsize`, we use square brackets notation to indicate the element we want displayed.

```
> popsize[2]
[1] 40
```

To see all the elements in a vector at once, type `popsize`. R will print off all the population sizes from n0 to n5. To see the second through the fourth elements, type `popsize[2:4]`. Vectors do not have to be constant in size throughout your entire program. R easily adds and deletes elements from vectors. Given the nature of our examples and questions, we will focus more on adding elements to vectors.

For example, if we want to add the population size at time 6, using the same growth equation, we write:

```
popsize[7] = popsize[6]*r; # appending n6 at the end of a row vector
containing n0 through n5.
```

Immediately you should realize we built `popsize` using only six numbers and now we are writing something to the 7th position in the vector `popsize`. R will automatically adjust the size of the vector. Another thing to note is that elements are called by their location in the vector. The first element is at location 1, the second is at location 2, and so on. Because we stored the initial population size (`n0`) in location 1, all the years are shifted up in the vector. For example, to look at the third year, we actually call the fourth element. In the above example, we are retrieving the population size of the fifth year (which is element number 6 in the `popsize` vector) and multiply it by `r`. Referring to the vector on the right side of the equals sign tells R to use that value in the equation (in this case the population size at time 5, `n5`). Referring to the vector on the left side of the equals sign assigns the values on the right side to that spot in the vector (in this case, the seventh spot). If you are not careful, you can accidentally write over previous values. Or you can choose to write over previous values.

By combining all of our population sizes in a single vector, this allows us to store values in a way that is easily manipulated by the computer itself. We can now automate the process and determine what happens to the population in 1000 years and on!

4. Flow Control: For Loops

In designing our programs, we usually have to do multiple things and repeat these things many times. By using loops, we only have to write the code once and we cycle over it until the job is completed. The best example is our current project. In exponential growth, we take the population size and multiply it by the growth rate. This gives us the population size at the next time. Then we multiply the new population size by the growth rate to determine the population size at the next time step. We then want to repeat this process for as long as we want.

For loops can be used for this purpose. For loops are sets of instructions that will be repeated a definite number of times. The following expression summarizes the basic structure of a for loop in R. Open up a new R script file and type in the following commands, saving the file as vectorgrowth.

```
r=2; # growth rate
 popsize = 20; #defining initial conditions
 for(time in 2:4){ #defining the start and end of the loop
      popsize[time]=popsize[time-1]*r  # the instruction to be repeated
 } #closing the loop
```

We will now go through the code and describe what each line accomplishes.

```
r=2; # growth rate
popsize = 20; #defining initial conditions
```

A for loop needs to be initialized in terms of initial population size and growth rate *outside* of the loop. The name used for initializing the variable and the name used in the instruction to be repeated *must match*. We are using the vector popsize to store the size of the population at different times. By default, a variable created in this way is a vector (an ordered list of numbers), in this case, a vector of length 1. This is the same as typing popsize[1] = 20; .

The next three lines reference the loop. For loops are run a specific number of times.

```
for (time in 2:4){
```

First we must tell R that we are going to use a for loop. The variable time is the iteration the for loop is currently on. In this case it refers to the time step we are currently working on. With for loops, the choice of start and endpoint for the loop needs special planning. We already have a value stored in the first element of popsize, which we can think of as the value for the first time step, which is the value for the zeroth generation. Therefore we start at the second time step (the 2), which would correspond to the first generation. And we want to only go through time step four. This is all coded in the 2:4 which can be read as start at two and finish at four increasing the variable time by one each time (default). This will run the code 3 times (time = 2, time = 3, time = 4). The open curvy bracket defines the start of the instructions to be repeated. The line

```
popsize[time]=popsize[time-1]*r # the instruction to be repeated
```

is the generalized form of the growth equation we have been working with ($N_{t+1} = N_t \times r$). Again, the generalized equation is that the population size at the next time step is the population size at the current time step multiplied by the growth rate. Another way to think about it, and how we need to code it, is that the population at the current time step is the population of the previous time step times the growth rate. Since we are storing population size in the vector popsize, r is the growth rate, and time refers to the time step, we get the code above. Note that this line of code is indented. When you are using loops, you need to indent the lines contained within the loop. This isn't for the computer, but for you. This allows you to quickly determine what is to be repeated. This may not sound important when we are working on something this simple, but when you have for loops nested inside of other loops and those are nested in other loops, you will have problems if you do not indent.

The final line of code is

```
} # closing the loop
```

This curvy bracket tells R that the `for` loop is finished. When R gets to this line, it causes the computer to go back up to the `for (time in 2:4)` and add one to `time`. If `time` is 4 or less, the cycle repeats, but if it is greater than four, R stops the `for` loop and continues with the code after the `} # closing the loop` line.

The best way to show how a `for` loop works is to work out what the computer is doing. First the computer stores `r = 2.` Whenever the computer sees `r` on the right hand side of an equals sign, it will replace it with the number 2. R then initializes the vector `popsize` with the value 20. The `for` loop is then started. The variable `time` is given the value 2. Two is less than or equal to four, so the computer continues on. The computer comes across the line of code

```
popsize[time]=popsize[time-1]*r
```

Using the values of `r=2` and `time=2,` it plugs those into the formula to solve, which becomes

```
popsize[2]=popsize[2-1]*2
```

which boils down to

```
popsize[2] = popsize[1] * 2
```

The computer then grabs the first element of `popsize` to give the solvable equation

```
popsize[2] = 20 * 2
```

which equals 40. The vector `popsize` now has two values in it: `[20 40]`. The computer then reads the next line of code and reads `}`. It goes back up the line for `(time in 2:4)` and adds one to `time` to make time equal 3. Since three is less than or equal to four, the computer proceeds with the code inside of the `for` loop. Now `time` equals three while `r` still equals 2. The code

```
popsize[time]=popsize[time-1]*r
```

becomes

```
popsize[3]=popsize[2]*2
```

where `popsize[2]` equals 40. This sets the third element in `popsize` to 80. Again the computer reads the code `}` and goes back to the for statement. The computer adds one to time, making it four. The line of code

```
popsize[time]=popsize[time-1]*r
```

now becomes

```
popsize[4]=popsize[3]*2
```

which sets the fourth element of popsize to 80×2 or 160. The vector `popsize` now has four values in it: `[20 40 80 160]`. Again the computer reads the end line and goes back up to the for loop. This time, when one is added to `time`, `time` is equal to five. Five is greater than four which tells the computer that it has run the for loop as many times as it is supposed to. The computer goes back to the code `} #closing the loop` and picks up the rest of the program. In this case, there is no code after that, therefore the program is done. Wheew! Aren't you glad that computers can do all this for you?.

While programming, you are going to mistype or even forget to type things. R is not forgiving, but tries to help you correct the errors as fast as possible. It is common to forget to type `}` to complete a `for` loop. Remove the `closing curly bracket` in your file and attempt to run your program. You will see that in the command line you have a + instead of the normal prompt > this tells you that a loop is still open that needs to be closed before you can proceed. You will see the + sign whenever you have a left parenthesis that does not have a corresponding right parenthesis.

If you are having a hard time finding the error in your code, start at the beginning and run each line of code individually (by clicking on the line and pressing command(ctrl)-enter). Stop when you hit an error or an unexpected result and try to figure out what the mistake is.

Another common error occurs by referencing vectors incorrectly. Because of the way the population size is stored in the vector `popize`, it is necessary to start at `time = 2`. If `time = 1` is used, then the right hand side of the instruction will produce an error, since by definition, no vector has an element in the zeroth location. Actually, change the `for` loop to go from 1 to 4 and run the code to see what happens (after you put the closing bracket `}`back into your program!). In the R console you will see:

```
Error in popsize[time] = popsize[time - 1] * r :
  replacement has length zero
```

The first line of the above code tells you the command that is causing problems for R. In this case, your program is attempting to call a value that does not exist in a vector, specifically you are trying to get the zeroth element of a vector, which does not exist. The second line is trying to explain the problem that has been encountered. When you get an error, take a breath and read what R is telling you. Many people get frustrated by the error and annoyed by the redness. Don't. Reread what R writes to your screen and correct the error. Whenever you see this error, you can type `time` at the command prompt and see exactly what value of `time` that R is trying to use.

Another common error is not matching the endpoints of the loop to the instruction within the loop. That is, you need to match how you increment `time` to the way that the elements within the vector `popsize` are chosen. For example, consider the following alternative way of writing the same loop. Notice in this example, `time` goes from 1 to 3 and not 2 to 4 like in the previous example.

```
r=2; # growth rate
popsize = 20; #defining initial conditions
for (time in 1:3){ # defining the start and end of the loop
  popsize[time+1]=popsize[time]*r # the instruction to be repeated
} # closing the loop
```

The output will be exactly the same in both versions of the loop, but the computation is different. Compare the equations to compute `popsize` in both of the scripts to see the subtle but important difference.

5. Plotting Vectors

Simply clicking on the command prompt, typing in `popsize` and browsing through numbers of population sizes would give us information but not in an easily understandable format. To obtain a plot of the population sizes through time, we can use the `plot` command. The `plot` command usually takes two vectors, one for the values along the x axis and one for the values along the y axis where the first x value is matched with the first y value, the second x value with the second y value and so on. If only the y values are supplied, the plot command will automatically create an x vector with the position of each element in the y vector. The vector `popsize` contains the values along the vertical axes. In this case, the statement

```
> plot(popsize)
```

will produce the graph in the figure window (one of the tabs in the lower right window in Rstudio).

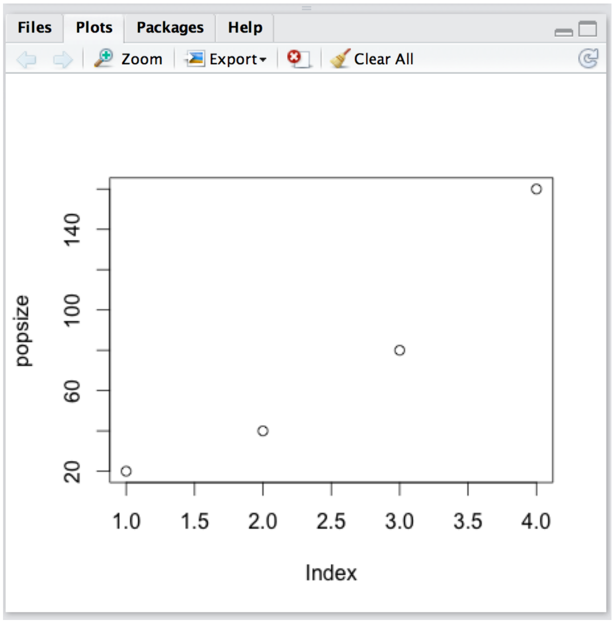


Figure 2: Population size plotted against its position in the `popsize` vector

Notice that the graph starts with the initial population size being plotted at 1 along the x axis. This does not correspond to the initial time. To change this, we will have to tell R exactly what we want plotted. Close the figure window to get rid of the bad graph. To draw the correct graph, we can either define a vector of x values outside of the `plot` command or within the `plot` command. As `popsize` from the previous script has 4 values, we can specifically build it with only four values:

```
> xvalues = c(0, 1, 2, 3);
```

Since our first value in `popsize` is the initial population size, we want that plotted on the x-axis at zero. To build a vector like this for thousands of time iterations would be tedious. Luckily there is a short cut. If we want to make a vector with 1001 values starting with 0 and incrementing by 1 each time, we write

```
> example = 0:1000
```

For our current program, we only need it to go to three.

```
> xvalues = 0:3
```

This creates the vector `xvalues` that contains the values [0 1 2 3]. We can now plot correctly (Figure 3).

```
>plot(xvalues, popsize)
```

As an alternative, we can instead produce the vector inside of the plot command.

```
>plot(0:3, popsize);
```

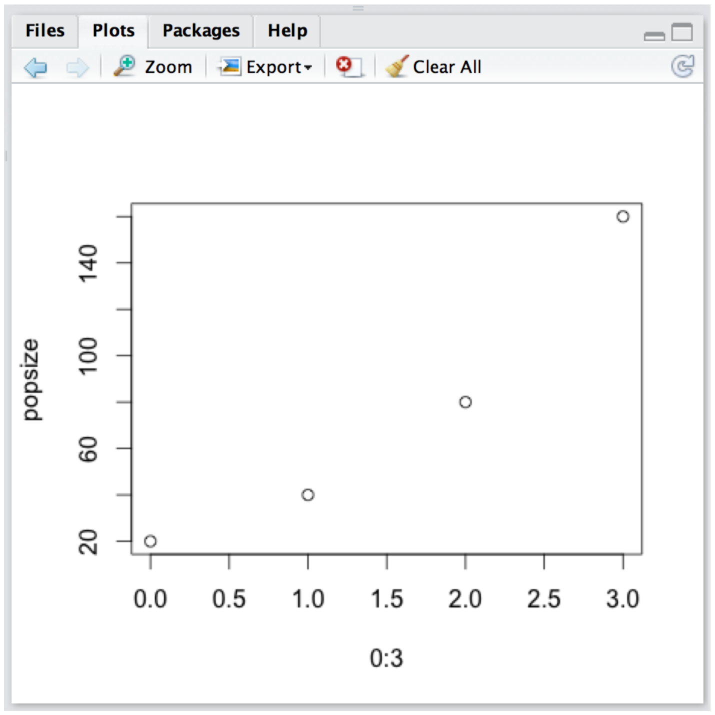


Figure 3. Plot of population size against time step.

This corrects the plotting problem and will produce a graph where the initial value is plotted at zero on the x axis. The plot command can be included at the bottom of our R script so that a graph is automatically made each time the script file is run. In this case, you want to generalize the plot command to ignore "magic numbers." For example, in our above plot commands, we knew that `popsize` had four values. What happens if you change the for loop in our script to run ten times? The vector `popsize` will contain 11 values (the initial value and the 10 times through the for loop), but our plot command will plot only the first four values. This code generalizes the plot function and makes it change depending on the size of `popsize`.

```
numiterations = length(popsize);     # number of values in popsize
numiterations = numiterations-1;     # because we start with an initial
                                     # condition at popsize(1), we actually
                                       # run one less year than popsize
plot([0:numiterations], popsize)     # this will plot everything nicely
```

We first need to know how large `popsize` is. Using the `length` command will give us the number of elements in a vector. We save the size of the vector `popsize` as `numiterations`. Since we stored the initial condition in the first element of the vector `popsize` and defined this as time zero, we actually ran the program for one less year than the number of values stored in `popsize`. Therefore we need to subtract 1 from `numiterations`. If you notice, a variable can be on both the left hand side and right hand side of an equals sign. In this situation, the right hand side is computed first (`numiterations - 1`) using the current value of `numiterations`. Once this new value is computed, `numiterations` is then reassigned this new value. For example, if `numiterations` is 10, the computer first computes 10 - 1 which is 9. This nine is then assigned to `numiterations`. Now every time the computer uses `numiterations`, it will use the value nine and not ten. The `plot` statement will now plot using `popsize` as the y values and the x values will come from the vector `0:numiterations`. All graphics properties can also be changed using written instructions. How to do this is outside of the scope of this little manual. Please consult the relevant documentation, starting with `help(plot)`.

6. Flow control: If else statements

If else statements are forks in the flow of scripts. Up to now, the flow of the program is from the first line to the last line, completely in order and not skipping a line. What if we want some code to run sometimes and other code to run at other times? Consider now the following situation: We are interested in population growth in a species where it experiences four years of low population growth followed by a high resource year where population growth increases greatly for that year. In our previous examples, population growth (`r`) has stayed constant across the years, but now it varies. We focus on determining whether it is an ok year or a great year and use the correct growth rate for that year. If else statements have a comparison. If that comparison is true, then you do something. If the comparison is not true, you find an `else` statement and do whatever the else statement tells you to do. In our case, the if else statement determines if it is a high growth year. If it is, the high growth rate is used. If it is not, the low growth rate is used.

Before, population growth was defined outside of the `for` loop. Now, we need to define it inside of the `for` loop as it changes depending on the year that we are currently in. Going back to the `vectorgrowth` file, we need to make some changes. Notice that we are making `time` go from 2 to 10, not 2 to 4.

```
#notice that we no longer define r here
popsize = 20; #defining initial conditions
for (time in 2:10){ # defining the start and end of the loop
  #this section modifies the growth rate depending on the year
  if( (time%%5) == 0){#this determines whether it is a great year
          r = 4
  } else { # or just an ok year
          r = 2
  }
  popsize[time]=popsize[time-1]*r # the population growth equation
} # closing the loop
```

Notice that the if else statement is inside of the `for` loop and for easier reading should be indented. We also indent the statements inside of the `if ... else` statement for easier reading. This simple program boils down to: if it is every fifth year, then r = 4. If it is not a fifth year, then the growth rate equals 2. Remember that `time %% 5` will give you a zero when `time` is divisible by five (as there is no remainder). One thing to point out is the double equals sign after the `time %% 5` (translation: time modulus 5). The double equal sign is NOT to be confused with the single equal sign. The single equal sign says to assign whatever is on the right side of the equal sign to the variable on the left side. The double equal sign is a logical operator. It compares the value on the left side to the value on the right side. If they are the same value, you get a 1 or a TRUE returned. If they are not the same value, then you will get a 0 or a FALSE returned. If you are using `if ... else` statements and the program is giving you errors or not working as it should (based on hand calculations), check you equal signs and make sure you are using the correct one given the situation. R will give you all sorts of error messages if the run the loop with only one equal sign in the if statement. The first error statement returned is usually the most helpful:

```
Error: unexpected '=' in: "  #this section modifies the growth rate depending
on the year  if(time%%5 ="
```

This is R's way of saying that there is an error in the command `if(time%%5 =` and that the equal sign is probably the problem. For any error you might encounter we recommend looking at the first error message returned as this is usually the most helpful. Be patient and re-check the section of code affected.

Another inequality we can test is whether two things are not equal to one another. Using this test, we get TRUE when two things are not equal to one another. We can rewrite the above example using the symbol !=, which tests for inequality. By using not equal to, we want the growth rate to equal two whenever it is not the fifth year, else the growth rate equals 4 because it is the fifth year. Focusing on just the `if ... else` statement from the above example gives us

```
if ( (time%%5) != 0){
    r = 2
} else{
    r = 4
}
```

With `if ... else` statements, we can check for more than just equality. If we want to determine how a normal population growth rate of 2 for five years followed by a growth rate of 3 for every year after that affects population size, we can.

```
if (time <= 6){
    r = 2
} else{
    r = 3
}
```

In this situation, we have to remember that we stored the initial population size as the first element in the vector `popsize`. Therefore, the fifth year is actually when the variable `time` equals 6. To read the statement, we say if `time` is less than or equal to six then r = 2, else r is equal to three (as `time` is year six and above). The symbols <= is less than or equal to. We could use only the less than symbol (<) but our `if` statement then becomes `if (time < 7)`, as we also want r = 2 when `time` is equal to 6.

We can also rearrange this example to use >= which is greater than or equal to.

```
if (time >= 7){
# remember, we want years 6 and above to have a growth rate of three
    r = 3
} else{
    r = 2
}
```

Again, we do not have to use the greater than or equal to symbols, but could use simply greater than (>). Our `if` statement would then be `if (time > 6)` as we want to include when time equals 7. For clarity in the program and in thinking, one version may be better to use than the other.

`If ... else` statements can also be used with more than one comparison. Consider a situation where the growth rate is 2 for the first five years, 0.5 for the next three years, and then goes back to 2 for all the years after that. In this case, we want the growth rate to be two if the variable `time` is less than or equal to six or greater than or equal to ten. There are two ways to do this. The first way uses two `if ... else` statements, the second uses a single `if ... else` statement.

We can nest `if ... else` statements inside one another. To test for two different values, we would need two if else statements. In this case, we can test whether or not the variable `time` is six or less. If we have computed more than five years of population sizes (the variable `time` is greater than six), we then need to test whether it is past the ninth year (the variable `time` is greater than or equal to ten) or between five years and nine years . Again, remember that the initial population size is in the first element of the vector, therefore all of our values of time will actually be one value above the actual year that we want.

```
if (time <= 6){     # for the first five years, use r = 2
    r = 2
} else {
    # We have been going for more than five years,
    # we need to see if we are on year nine or above
    if (time >= 10){
          r = 2
    } else{
          # we are on year 6,7, or 8
          # therefore use the low growth rate
          r = 0.5
      } # this ends the (time >= 10) if else statement
}    # this ends the (time <= 6) if else statement
```

Note: If you accidently delete the contents of your script file (which is easy to do if you highlight it to run it, and then hit a random letter), don't worry, just press ctrl+Z, (or command+Z on a mac) to undo your typing and restore the script file.

We have nested an `if ... else` statement inside of another `if ... else` statement. In English, this program first determines whether `time` is less than or equal to 6. If it is, then our growth rate is 2. If it is not, we go to the `else` statement of the first `if ... else` statement. Inside that `else` statement, we have another `if ... else` statement. If `time` is greater than or equal to 10, the growth rate is 2, else the growth rate is the low value (0.5). In this situation where you have nested statements, be sure to include an } for each `if ... else` statement! This advice holds true for when you also nest different types of statements such as `if ... else` statements, `while` loops (discussed later) and `for` loops! A comment telling you which loop you are closing with the } will help you keep everything in order. Despite getting the job done, our example is more confusing than it needs to be. R allows you to test multiple comparisons in the same `if ... else` statement in two ways.

```
if (time <= 6){ # for the first five years, use r = 2
    r = 2
} else if(time >= 10){
      # We have been going for more than five years,
      # so we need to see whether it has been more than eight years
    r = 2
} else{
    # we are inbetween the fifth year and the ninth year,
    # therefore use the low growth rate
    r = 0.5;
} # this ends the if elseif else statement
```

We have kept the comments the same between the two different methods to show how they compare. Obviously, this method is easier to read from a human perspective. In this case, the computer first checks to see if `time` <= 6. If it is, it sets `r = 2` then exits the `if ... else` statement. If it is not, the computer then checks to determine whether `time` >= 10. If it is, `r = 2` and the computer exits the `if ... else` statement. If `time` is not greater than or equal to 10, then `r = 0.5`. The computer checks the equalities in the order that they appear in the program.

Because the growth rate is the same whether `time` is less than or equal to six as well as if it is greater than or equal to ten, we can actually combine those into a single `if` statement.

```
if ((time <= 6) | (time >= 10)){
      r = 2
} else {
    r = 0.5
}
```

In this case, our two `if` statements were combined into one using the ***or*** connector (|) and by enclosing both requirements in one set of parentheses. To type the *or* connector, hold down shift and hit the backslash button. The *or* connector states that if either of the comparisons are true, then you set `r` to 2. When using or, only one of the comparisons has to be true. There is also an *and* connector (&) that only runs if both the comparisons are true. Instead of thinking that `time` is less than or equal to 6 or greater than or equal to ten, we can instead ask whether `time` is less than ten and greater than six. In this case, `r` would be set to the low growth rate.

```
if ((time <  10) & (time > 6)) {
    r = 0.5;
} else {
    r = 2;
}
```

When using comparisons, make sure you are actually testing for the value that you want to test for. Take into consideration what your variables mean, for example that `time` is actually the year plus one. Also, determine if you actually want the specific year included or only everything above that year. Many errors come from bad comparison statements. Take your time and make sure you do it right. As we wrote this, even we made a couple mistakes.

Most importantly, we don't want you to get bogged down into being able to change your programs to specifically show that you can write your code in many different ways, but use the code that is the easiest to read and most intuitive.

7. Flow Control: While loops

Let's continue with an example of an invasive species with an annual life cycle. Suppose that damage will occur to the native ecosystem if the population size of the invasive species is larger than 1200. The question to answer is how much time will it take until the population size is greater than 1200 individuals? The best way to answer this question is to take the growth equation described in section 2, Variables and Mathematical Operations, and solve it analytically. We want to answer this question using R to illustrate `while` loops.

The main idea is to run the equation in section 2 until the population size reaches 1200. Then we look at the size of the vector `popsize` to figure out the time. The situation is different from the for loop because we do not know in advance how many times to iterate the equation. In this case we can use the while loop. The while loop iterates the equation an indefinite amount of times until a condition is satisfied. In our case the condition will be when the population size is greater than 1200 (`popsize` > 1200). Open up a new script file and type in the following code:

```
r=2; # growth rate
popsize = 20; #defining initial conditions
time = 0; # initializing time
while (popsize[time+1] <= 1200){#defining the condition for finishing the loop
   time = time +1 ; # make time advance in one unit
  popsize[time + 1]=popsize[time]*r; # the equation of population growth
} # closing the loop
```

The first three lines define the variables `r`, `popsize`, and `time` which are needed inside the while loop. The new variable in this section is `time`, which we will use to store the number of generations we have advanced. Unlike the `for` loop, we have to explicitly define the start point, the step size, and how our focal variable `time` is modified. We define the starting point outside of the `while` loop and the step size within the `while` loop. Since we want to know how many years have to pass before a population reaches more than 1200 individuals, we initialize `time` with 0. Consider if the population size is initially 1200 or more individuals. In this case, `time` would never be incremented, meaning that it takes 0 years for the population to be greater than 1200 individuals.

Inside the while loop, we first increase the `time` variable by one unit (in this case, a year). We do this to use `time` as both an index for the `popsize` vector and a counter. Population growth is handled as described in the section devoted to for loops. The first time through the `while` loop, `time = 1` and a second value is added to the vector `popsize` (in this case 40). Then the } is reached. The computer goes back to the beginning of the `while` loop and determines whether `popsize` <= 1200. All of the values in the vector `popsize` is compared to 1200. The while loop will continue if *all* of the values in the `popsize` vector are smaller or equal to 1200. Since both 20 and 40 are smaller than 1200, the cycle repeats. In the second round, `time = 2` and a third value is added to `popsize` (the value of 80). The vector `popsize` now contains the values [20 40 60]. Since all of the values of `popsize` are less than 1200, the `while` loop continues. This goes on until the sixth round, when `time` = 6 and a population size of 1280 is added to the seventh position in `popsize`. This time when the computer determines whether `popsize` is less than or equal to 1200, the computer will find the value 1280 and no longer run the `while` loop.

The size of the `popsize` vector (number of entries) can then be used to find out how long it will take the species to become a pest. At the command line, type in the statement:

```
>length(popsize)

       ans =
              7
```

says that the vector has 7 entries, but since the initial time is zero, the last entry corresponds to `time` = 6. The population size during the sixth year is

```
>popsize(7)

       ans =
           1280
```

Notice that the discrete model of population growth assumes the population size jumps from one value to another and therefore the program does not return the exact time when the population turned 1200.

Rerun the script and notice how long it takes before the computer stops and gives you an answer. Now, run the code with the population growth rate less than or equal to 1 (the population is stable). Do you get an answer? In this situation, the population size will never be greater than 1200. You know this, but the computer does not know this. The computer is going to continue running your code until a value stored in `popsize` is greater than or equal to 1200, which will never happen. This is what is known as an infinite loop. In the console window in Rstudio, click on the little stop sign in the right hand corner of the window to stop your code. This should cause R to immediately quit whatever code it is running. While writing code, you need to think about infinite loops and prevent them from occurring. In this case, a simple `if ... else` statement testing whether the growth rate is greater than 1 will prevent this.

8. Functions

Currently, we have been focusing on writing script files. Sometimes, as we write script files, we find ourselves typing in the same code over and over again or find ourselves doing the same computations but with different numbers. We could type that code in new each time we need it, or we could only type it in once and use it many times. I understand your love of typing things in multiple times, but that is really something you should get over. A programming function is not to be confused with a mathematical function (even though a programming function could code a mathematical function). In papers, a reference to mathematical function refers to an equation. A programming function is code that does a specific job, but does not have to be an equation. Using our basic discrete population growth program, we are going to turn it into a function. This could be useful if you want to run many runs over the course of a day and analyze the results without having to modify the script file itself.

If you have been following this guide, you have already used functions. The `length` command, `plot` command, and `abs` command are all functions. You type in the name of the command and tell the computer what values or vectors you want the function to use, and the computer does the rest.

First, we need to type the code we will be modifying. Open up another editing window like you do with normal script files. Type this code in:

```
r=2; # growth rate
popsize = 20; #defining initial conditions
for (time in 2:4){ # defining the start and end of the loop
  popsize[time]=popsize[time-1]*r # the instruction to be repeated
} # closing the loop

numiterations = length(popsize);      # the number of values in popsize
numiterations = numiterations - 1;    # we start with an initial condition
                                      # at popsize(1), we actually run
                                      # one less year than popsize
plot(0:numiterations, popsize)        # this will plot everything nicely
```

All of this code should look familiar to you. It is a basic discrete growth program that then plots the population size versus generations on a graph. Looking through this, notice that there are three values that you will want to modify: growth rate `r`, initial population size, and how long the `for` loop runs. In this case, these are the three values that we need to generalize.

Functions start with a `function` declaration. In this case, we are going to create a function named `discretegrowth`. As stated before, we will modify three variables (parameters for our growth equation) within our function. We need to tell R what variables we are manipulating and how we will refer to them within the file. Type in the below line just above the code that you want to make into a function.

```
discretegrowth <- function(r, inital, howlong){
```

This line tells R to create a function called `discretegrowth` that will use three variables that will be defined by the user (`r`, `initial`, `howlong`). Just after the last line of our code type:

```
return(d)
}
```

This tells R that after the function is complete, it should return whatever is in variable `d`. Now we need to modify the current code to be a function. We do this by changing the program to use these new variable names where we need the growth rate (`r`), the initial population size (`initial`), and the number of years to run (`howlong`). Modifications are given in bold.

```
discretegrowth <- function(r,initial,howlong){
#Notice that the line below is commented out.
#r=2; # growth rate

popsize = initial; #defining initial conditions
for (time in 2:howlong){ # defining the start and end of the loop
  popsize[time]=popsize[time-1]*r # the instruction to be repeated
} # closing the loop

numiterations = length(popsize);      # the number of values in popsize
numiterations = numiterations - 1;    # we start with an initial condition
                                      # at popsize(1), we actually run
                                      # one less year than popsize
plot(0:numiterations, popsize)         # this will plot everything nicely

d = popsize                            # function returns popsize vector
return(d)
}
```

Since we used the same variable name for the population growth rate, we did not have to change it in the `for` loop. We did have to remove it from being initialized as the value of 2, so we commented it out with the # symbol. If we had not commented out that line, the function would have used a growth rate of 2 every time you used the function. Select all the code from the script file and run it in your R console. This will produce the function `discretegrowth`, that we can now call from the command line or from another script file.
If you want to run the same model we have been running, go to the command line and type in

```
>discretegrowth(2, 20, 4);
```

A graph will pop up that should look awfully familiar. Congratulations, you just created and used your own function. What you typed in was the name of the function and the three values the function needs to use to do its job. The first value is the population growth. The second and third are the initial population size and the number of times you want the `for` loop to run, respectively. Notice that you have to put in the variables in the exact order that they appear on the first line of the function file. If you typed in `discretegrowth(20,2,4)`, the computer would assign 20 to the population growth (`r`), 2 to the initial population size (`initial`), and 4 to `howlong`, which is not what you want. If you want to look directly at the values that are being plotted, you need to type in

```
> popsizetemp = discretegrowth(2, 20, 4);
```

This creates a new variable in your workspace called `popsizetemp` that contains all the values of `popsize` from the function. If you have been following this guide and have not restarted R, you have another variable called `popsize` in your workspace. The `popsize` in your workspace and the `popsize` referred to in your function are two different variables. Don't believe me, type this at the command prompt:

```
> popsize = discretegrowth(2, 20, 4);
> popsizetemp = discretegrowth(2,50,50);
```

Do these one at a time. Rstudio overwrites the current plot with the new one, so you will need to run the commands separately to see the results of each. You can look at them and make sure they are different. One should only have four points on it while the other should have fifty. Go back to your command line. Type in

```
> length(popsize)
> length(popsizetemp)
```

The first command should tell you that `popsize` is a vector of size 4 . The second command should tell you that `popsizetemp` is a vector of size 50. But if you think about it, `popsize` was called in the function `discretegrowth` when we called it. Therefore, `popsize` should be the same as `popsizetemp`, but that is not what happens. The variables in the parentheses when we call the function are the only variables in the function that can be modified by us (in our case, the growth rate, initial population size, and how long to run the program). The variables created in the function are *only* seen by the function and by no one else. This allows your function to be independent of your other code and prevents your program from mysteriously breaking if you happened to accidentally use the same variable name twice in different contexts. Also, the only way to get a variable back from a function is by having the function return a value. We have `return(d)` in the last line of our function. This says that the only variable that is being returned from this function is whatever is assigned to variable `d`. If you look in our function file, we added a `d = popsize;` line. This assigns the values of the vector `popsize` to the variable `d`. Therefore, we can only get the values of the vector `popsize` without modifying the function file.

Functions can range from full blown detailed programs (such as ours and more advanced) to small mathematical functions or simple comparisons.

9. Ordinary Differential Equations

We have been focusing on the discrete population growth equation, where time changes in very large steps (years or generations). In ecology, we also have the continuous population growth equation where time changes in very small increments that are very close to zero. The continuous population growth equation is defined as the ordinary differential equation (ODE)

```
dN
--- = rN
dt
```

where `r` is the growth rate, `N` is the population size, and `dN/dt` is the rate of change of the population size. With this equation, we continue to work with exponential growth.

R provides one tool for simulating (not solving) ordinary differential equations (ODEs). It is important to know the difference: solving an equation means to transform the ODE into another equation called a solution. Simulating an ODE means to obtain a collection of numbers that correspond to a solution (without solving the equation). Solving in general can be done for certain equations, whereas a much wider range of ODEs can be simulated. This section will be concerned only with simulation.

By its very nature, an ODE cannot be iterated in the same manner we used in previous sections. Numerical methods have been devised to obtain simulations. The basic idea behind simulation is to start with an initial condition and set of parameter values. Using the ODE, the rate of change of the population size evaluated at the initial condition can be obtained. The next time point is chosen and the corresponding simulation value is computed using the slope and the size of the time step. The same procedure is repeated over and over to simulate the given ODE.

Numerical methods for simulating ODEs have been implemented in a single R function, making our lives easier. Typically, the library `deSolve` is not included in R, so we must install this library.

Go to the lower right part of Rstudio and click the `Packages` menu. This will show you the available packages in your installation. If `deSolve` is missing from the list, click the install packages button. A new window pops up, where you can type the name of the library, and click on the install button. `deSolve` should now show up in the list of available packages. The last step is to load the package, which is done by checking the corresponding box. If you prefer to use the command line, you can install the `deSolve` package with the command `install.packages('deSolve')` and then load the package with the command `library(deSolve).`

To simulate an ODE we need one script with several required elements: a vector of parameters, initial population density, and a time vector. We will discuss then in order.

First, we will define the vector of parameters. We will use a "named vector", so we can extract the elements without error. If we call this vector `pars` we can write

```
pars <- c(r=2)
```

which defines the growth rate `r` as 2 (see the code for section 9 for an example with two parameters).

Together with the parameter values, we will also need our initial population density (mathematicians like to call this an initial condition). In this case, we don't need a named vector, we can relax a bit and use a simple vector as follows

```
ic <- c(0.4)
```

Remember that a vector can hold any number of elements, including just one element.

The last accessory for our simulation is a time vector. The package will compute population densities for a lot of time steps, but we may not be interested in all of them. We can define the time points where the population size will be returned. If we want an evenly spaced time vector, the function `seq` comes handy.

```
time <- seq(from=0, to = 10, by = 0.5)
```

The last command produces a vector `time` with a sequence of numbers starting at zero, ending at 10 with increments of 0.5 (so 0, 0.5, 1, 1.5 and so on until 10).

At last we can focus on the heart of the simulation – our equation. We need to translate our equation into R code as a function. Create a new function named `expgrowth`. Remember that functions in R may occur at any line of the script.

```
exp.gr<-function(t,x,p)
 {
 }
```

We are not only defining the function, but also defining the three parameters that our function needs: `t` refers to time, `x` to the previous solution, and `p` is the parameter (the growth rate). The braces will enclose the differential equation and the return values. Other letters can be used, but we will stick to this notation, as it is found in the R (`deSolve`) documentation. To understand how to convert our differential equation into R code, it maybe be easier to see a 'flat' notation of the equation:

```
dN/dt = r * N
```

The first thing to notice is that the differential equation uses the notation dN/dt and the letter N for the population density, while our function uses the letter x. It will be necessary to change the notation as follows

```
dx/dt = r * x
```

As for the dx/dt, we can use only one letter: x. We can then rewrite the differential equation in R code as

```
x <- r * x
```

R requires `x` to be a vector. In our case, `x` is a vector with only one element, but `x` could contain many elements. By including many elements, R could simulate multiple ODEs simultaneously that interact with one another. These are called coupled ODEs with a good example being predator-prey dynamics where there is an equation describing the prey dynamics and an equation describing the predator dynamics. Because `x` is a vector, we should denote this by appending a number at the left hand side of the ODE and use vector notation each time we refer to the state variable as follows

```
x1 <- r * x[1]
```

Please notice that the ODE requires one parameter called `r`, whereas the function we defined above will look for a vector of parameter values called `p`. If we define the parameter values through a named vector (see above), then we need to change our notation to be able to extract the required values from the named vector `p`.

```
x1 <- p[“r”] * x[1]
```

Instead of using position within the vector to tell R which element of the vector `p` to use, we are using the name of the element (in this case `r`) found in the vector `p`. Finally, we have the R version of our differential equation and can now include it between the braces of our `exp.gr` function.

```
exp.gr<-function(t,x,p)
  {
  x1 <- p[“r”] * x[1]
  }
```

Remember we need to tell a function what values to return to us. Because we store the results of our simulation in `x1`, we need to tell the function to return the values of `x1`. The function `lsoda` requires the function to return a list, so we make `x1` a list with the `list()` function. In a function this simple, defining what is returned may seem redundant, but it ensures clarity and paves the way for habitats that are useful in complex functions.

```
exp.gr<-function(t,x,p)
{
x1 <- p[“r”] * x[1]
list(c(x1)) #this is the same as return(list(c(x1)))
}
```

The function `exp.gr` will return a list, composed of a vector with one entry, corresponding to the population size `x1`, the state variable. You may be wondering why the parameter values vector was defined as `pars`, and the function expects the parameters to be called `p`. Good question! We are not finished.

To simulate the ODE we need to call the function `lsoda` (part of the `deSolve` package), and feed it the initial condition (`ic`), the time vector (`time`), the ODE coded as a function (`exp.gr`), and our parameter values (`pars`).

```
exp.gr.out <- lsoda(ic,time,exp.gr,pars)
```

We store the output of the `lsoda` function in a vector called `exp.gr.out`. The `lsoda` function takes our ODE function and passes the parameters `pars`, initial condition `ic`, and `time` vector into the ODE function and resolves the name discrepancies. As a simple conceptual example, `lsoda` contains a vector named `p`. The function `lsoda` assigns our vector `pars` to this vector `p`. The vector `p` is then used in the `exp.gr` function.

Type in `exp.gr.out` to see the output. If everything is right, the output should be a matrix with two columns, the first is the time vector, and the second is the population density for each time step. We can in turn take this output and plot it for analysis.

```
plot(exp.gr.out[,1],exp.gr.out[,2],type="l",xlab="Time",ylab="Population
size")
```

To plot the output, remember we need to provide two vectors to the `plot` function: one for the x values and the other for the y values. This is done by extracting the first and second columns of `exp.gr.out` as separate vectors. Because `exp.gr.out` contains both rows and columns, you extract values by referencing both the row and the column, separated by a comma. For example, to extract the value from the first row (the first time step) and first column (the population density at the first time step) of the `exp.gr.out` matrix, you would write `exp.gr.out[1,2]`. Because you want a vector of all the population densities through time, you want to extract all the rows for the second column. To do this in R, you would write `exp.gr.out[,2]`. Notice that there is nothing before the comma, which R interprets as "use all of the rows." You can go ahead and run the code and be amazed at your new skills (Fig 4).

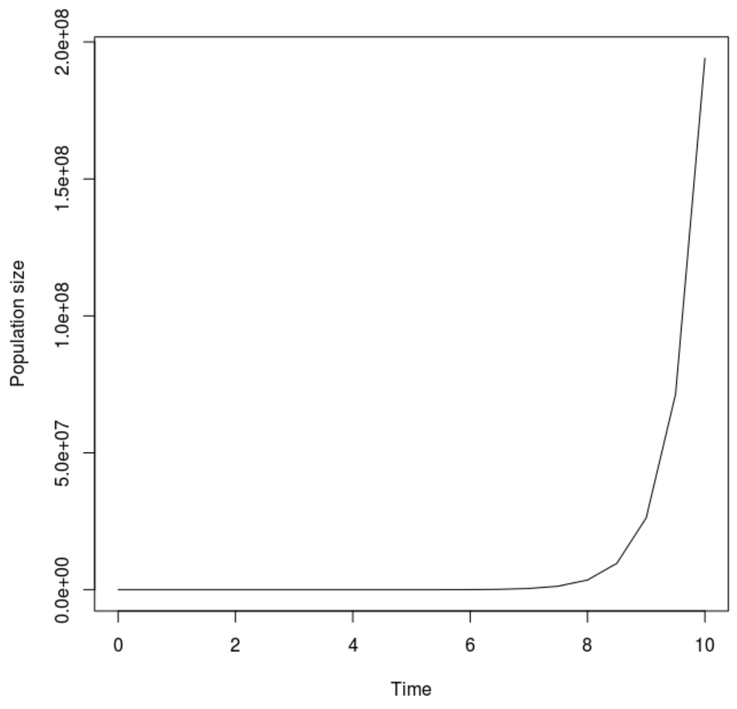


Figure 4. Population density through time of the continuous population growth equation. Solved using an ODE solver.

Common errors when writing scripts to solve ODEs are: changing the order of the parameters listed in the function, dropping parameters when writing the ODE function, and forgetting to define an input for the simulation, such as the initial condition or the time vector.

An error due to changing the order of the parameters can be tricky to figure out . For example, let's go to the definition of the ODE function and swap two parameters. Our function now looks like

```
exp.gr<-function(t,p,x) # as opposed to exp.gr<-function(t,x,p)
 {
 x1 <- p["r"]*x[1]
 list(c(x1))
 }
```

The change seems minor, but the consequences are not. When running the script, you will not be notified of any error. R just keeps working as usual. When you plot the results, you will be given a blank graph. Once the `plot` command fails, the next obvious step will be to check the output `exp.gr.out`, only to find

```
     time   1
1    0.0  0.4
2    0.5  NA
3    1.0  NA
4    1.5  NA
…
```

This output means two things: 1) the initial condition at time 0.0 is 0.4 and 2) that `lsoda` was unable to compute the any time step beyond time 0.0 as denoted by the NAs in the second column. In this case, R does not complain because it is able to find the correct initial condition, but the wrong order of parameters in the function makes it impossible to compute valid data for the simulation.

A second common error is to drop one letter in the function. Let's go back to the script, and delete the `x` of the population size. The function now looks like

```
exp.gr<-function(t,p)
 {
 x1 <- p["r"]*x[1]
 list(c(x1))
 }
```

When you run the script, you will get the following error:

```
> exp.gr.out <- lsoda(ic,time,exp.gr,pars)
Error in func(time, state, parms, ...) : unused argument(s) (parms)
```

This error is a bit confusing. The confusion comes from the fact that `lsoda` expects the function describing the ODE have a specific number of inputs into the function (arguments). In this case, `lsoda` expects `exp.gr` to have three arguments but we only listed two (`t` and `p`) because we forgot `x`. The error is basically saying that R has three arguments that it needs to assign values, but we only gave it two, so `parm` cannot be assigned anything. In this case, `time` corresponds to `t` in the function `exp.gr`. The next input that R expects is `state`, which should be our population size. R takes the next input and assigns it to `state`, but the next input in our bad `exp.gr` function is `p`, which is actually our parameters such as growth rate!. Then R tries to assign something to `parms`, but when R looks at the function `exp.gr`, R finds nothing. This is why it reports that the parms is unused.

10. Conclusion and Future Direction

And this is the end as of now. Kind of a let down as there is so much more to talk about. This guide is only meant to get you started thinking like a programmer and put basic programming techniques into a conceptual framework. You will use these basic concepts whether you continue to develop models or simply develop a program to manipulate your data into a useful format (such as converting ten years worth of hourly measurements of rainfall into monthly measurements of drought). One can focus on the logic of programming or focus on learning the ins and outs of R. With respect to learning about programming, we recommend thinking about coding, writing lots of code, and reading other people's code. We support many of the ideas presented in Peter Norvig's "Teach Yourself Programming in Ten Years" (http://norvig.com/21-days.html). If you really get into programming, a book on algorithms will be useful and could potentially save you time (especially when a simple code change can make a program that takes 4 days to run only take a day or less).

If you want to continue to develop models in Biology, you can find books that are overviews of the field or delve into specific techniques. We presented simple population growth models, but there are many other population models as well as other types of models. *Modelling for Field Biologists and Other Interesting People* by Hanna Kokko is a great starting point to learn about the many different types of models in ecology. Roughgarden's *Primer of Ecological Theory* goes into greater depths of programming and mathematics of diet choice, quantitative genetics, and population biology. Gotelli's `A Primer of Ecology` is a great introduction to much of the math and theory in ecology, but will require you to convert these to programs yourself (which is a great programming exercise and develops your understanding of the theory). Depending on your interest, a book specific to your question and type of model will be required. For example, Caswell's *`Matrix Population Models: Construction, Analysis, and Interpretation`* is the definitive guide to using matrix models to study population growth, as opposed to the discrete time equations we used in this example. Behaviorists may be interested in Dynamic Programming to solve decision problems. In this case, either Mangel and Clark's *`Dynamic Modeling in Behavioral Ecology`* or their *`Dynamic State Variable Models in Ecology`* are required guides. Specific references can also be found at the end of the respective chapter in Kokko's book. Obviously, we are missing many of the references out there for many of the topics we couldn't discuss. Nothing that couldn't be remedied by a simple literature search or Internet search or asking other modelers.

The best piece of advice about programming we (ok, Chris specifically) have ever received is this: The most important thing in programming is to make sure your code is correct and that it is doing what it is supposed to.

11. References

Caswell, H. 2000. Matrix Population Models: Construction, Analysis, and interpretation. Sinauer, Sunderland, MA. ISBN: 978-0878930968

Clark, C.W. and Mangel, M. 2000. Dynamic State Variable Models in Ecology. Oxford University Press, Oxford.

Gotelli, N.J. 2008. A Primer of Ecology. 4th edition. Sinauer Associates, Inc., Sunderland, MA.

Kokko, H. 2007. Modelling for Field Biologists and Other Interesting People. Cambridge University Press, Cambridge. ISBN: 9780521538565

Mangel, M. and Clark, C.W. 1989. Dynamic Modeling in Behavioral Ecology. Princeton University Press, NJ. ISBN: 978-0691085067

Norvig, P. 2001. Teach Yourself Programming in Ten Year. http://norvig.com/21-days.html

Notepad++ http://notepad-plus-plus.org/

R Core Team. 2013. R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing. Vienna, Austria. http://www.R-project.org/

RStudio, Inc. 2014. RStudio. https://www.rstudio.com/