A Field Guide to Programming:
Population Growth in MATLAB

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 MATLAB programming language (www.mathworks.com) with a slant towards population biology. The free programming language Octave (www.octave.org) can also be used, but compatibility between MATLAB and Octave is not guaranteed. 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

We assume that you already have a copy of MATLAB (any version) or Octave installed on your computer. Upon opening MATLAB, you will see multiple windows forming the graphical user interface. The largest window is known as the command window (denoted by a 1 in Figure 1). By clicking in this window and typing, whatever you type will appear after the >> sign (called prompt). Hitting `ENTER` will cause MATLAB to execute whatever you had typed. The basic installation of Octave has the command line only. A graphical user interface can be installed, but is beyond the scope of this manual. In this introduction, when something is typed by you the user, we will use the `Courier type font`. Answers returned from MATLAB and items reference from the computer code will also be in `Courier type font`.

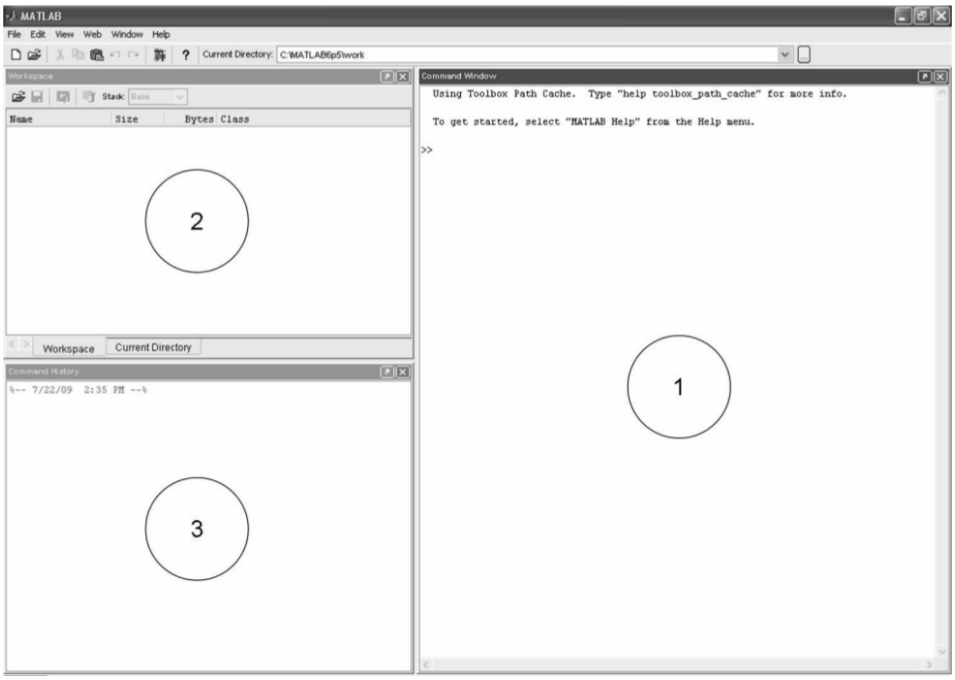

Figure 1. Layout of the MATLAB User Interface. Frame 1 is the Command Window. Frame 2 and 3 are the Workspace/Current Directory and the Command History, respectively.

We begin by exploring the simple calculator properties of MATLAB.

```
>>1 + 1
```

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

```
ans =
     2
```

which is telling you that your answer (`ans`) 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.

```
>>mod(10, 3)
```

Mod 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 MATLAB 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 `mod(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 `mod(48,12)` gives you an answer of zero. When you use `mod`, 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 `mod` operation, it means 12 o'clock.

You can combine operations into a single line. Just like you learned in algebra, MATLAB follows the order of operations. Parenthesis and functions (such as mod(x,y) 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. As you type these in, you should notice what you typed in appears in the command history (marked by 3 in Figure 1). If you want to reuse a command, you can double click it in the command history and it will run in the command window. You can also drag and drop commands from the command history into the command window. This copies the command but does not run it until you press enter.

As a simple calculator, MATLAB is very expensive. 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 MATLAB command line, type `r = 2` followed by `ENTER`. This should give you

```
>>r = 2
            r =
                        2
```

This is MATLAB's way of saying that it was capable of reading and executing one instruction. It may seem boring, but for complex algebraic expressions, this is useful. Now in this session of MATLAB, you can type in `r` at anytime and MATLAB 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 MATLAB will automatically use that new number where ever it sees `r`. Please note that `r` is very different than `R` and that MATLAB is case sensitive and makes a distinction between the two. Notice that your new variable appears in another window, called the workspace (marked by 2 in Figure 1). In naming your variables, always start with a letter (upper or lowercase). Numbers are allowed after the letter, as well as underscores. 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 low case L.

To help simplify life and expedite the learning process, we are going to create a MATLAB script file. Without a script, if we wanted to run a command multiple times, we would have to input the command 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* and choose *M-file*. A new window will open up, which we will refer to 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 MATLAB wants to give us answers. If you are using Octave, you need to open up Notepad or some other simple text editor. One author uses Notepad++ (http://notepad-plus-plus.org/) as a high powered simple text editor, but Windows Notepad works just as well for these simple exercises.

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 and click Save. If you are using Octave, you need to click "Save as..." in your text editor and name the file "popgrowth.m" without the quotation marks.

On the first line in our new script file, type in `r = 2`. This will keep the file whole as opposed to us having to write things on the command prompt before focusing on our script file. 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 the same field trip. 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 percent sign and the semicolon. If you type this into the command line and hit enter, you will not see any results. Before, MATLAB would show us our variable name and what it is equal to, but it didn't this time because we told it not to by adding a semicolon after the instruction. The percent sign tells MATLAB 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 MATLAB. 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 hit F5 while the script file is selected. Hitting F5 tells MATLAB to execute the script file. If you are using Octave, you need to type in the name of the script file at the command prompt and hit enter. For both software packages, make sure your software is pointing to the folder that contains your script. The standard place to save the file is in the work directory of the MATLAB folder. If you want to save it someplace else, like on your desktop, make sure that MATLAB is looking there. At the top of the main MATLAB window, there is a section labeled "Current Directory" where you can find and select different folders. Make sure the folder selected is the folder that contains your code. At the command line in Octave, you can use the `dir` command to list the directories available and the `cd` command to change into those directories. For example, `dir` list all the directories such as Desktop, Music, Programs, and so on. The command `cd Desktop` allows you to access files from your Desktop.

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. You can also double click on the name `n1` in the workspace window. 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 F5 again. Typing `n2` into the command prompt and hitting `ENTER` also gives us the answer. Remember, if we remove the semicolon at the end of the line, MATLAB will automatically return the value of n2 to us.

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`, and hit F5 to run the script again but with the new values.

We can also type in the name of the file (in our case popgrowth) on the command prompt and run the script file without ever opening it. When using, saving, and running script files, make sure MATLAB 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

In MATLAB, vectors are collections of 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 MATLAB as a collection of numbers separated by commas (or blank spaces) and surrounded by square brackets. They can be given names just like variables storing only one number. After running the previously created script file from above, go back to the command prompt and type

```
>> popsize = [n0, n1, n2, n3, n4, n5] % 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.

```
>> size(popsize)
 ans
      1, 6
```

The output indicates the number of rows and the number of columns (in that order always).

Elements of vectors can be displayed or changed, erased or added. For example, to display the second element of the vector `popsize`, we use the parenthesis notation to indicate the row and column we want displayed.

```
>> popsize(1,2)
ans
     40
```

Since the vector `popsize` only has one row, the row number in the instruction can be omitted. The same result is obtained with the instruction `popsize(2)`. To see all the values in a vector at once, type `popsize(:)`. MATLAB will print off all the population size from n0 to n5. Vectors do not have to be constant in size throughout your entire program. MATLAB easily adds and deletes values from vectors. Given the nature of our examples and questions, we will focus more on adding values to vectors.

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

```
popsize(1,7)=popsize(1,6)*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`. MATLAB 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 multiplying it by r. Referring to the vector on the right side of the equals sign tells MATLAB 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. 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 MATLAB. Open up a new MATLAB script file and type in the following commands, saving the file as vectorgrowth.

```
r=2; % growth rate
popsize = 20; %defining initial conditions
for time = 2:1:4 % defining the start and end of the loop
  popsize(time)=popsize(time-1)*r; % the instruction to be repeated
end % 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(1,1) = 20;`

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

```
for time = 2:1:4
```

First we must tell MATLAB 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. We want to progress one time step at a time (the 1). And we want to only go through time step four. This is all coded in the `2:1:4` which can be read as start at two and finish at four increasing the variable `time` by one each time. This will run the code 3 times `(time = 2, time = 3, time = 4)`. 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

```
end % closing the loop
```

This `end` tells MATLAB that the `for` loop is finished. When MATLAB gets to this line, it causes the computer to go back up to the `for time = 2:1:4` and add one to `time`. If `time` is 4 or less, the cycle repeats, but if it is greater than four, MATLAB stops the `for` loop and continues with the code after the `end % 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. MATLAB 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 `end`. It goes back up the the line for `time = 2:1:4` 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 `end` 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 elements 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 `end` 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. MATLAB is not forgiving, but tries to help you correct the errors as fast as possible. It is common to forget to type `end` to complete a `for` loop. Remove the `end` statement in your file and attempt to run your program. MATLAB quickly tells you (in bright red):

```
??? Error: File: C:\MATLAB6p5\work\vectorgrowth.m Line: 5 Column: 27
```

```
"end" expected, "End of Input" found.
```

MATLAB tries to help you as much as possible by telling you where the error occurs (the first line of the error statement) and what the error is (the second line of the error statement). The statement `"end" expected, "End of Input" found.` occurs when MATLAB requires a word or symbol (in this case the word `end`), but gets something completely different (in this case the end of the program). In this regard, Octave isn't as helpful, but does direct you to the line where the problem is. With any errors, the line the error returns should be used as a starting point to find the error. The problem with your code may not be on that exact line, but somewhere around that line.

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 `end` statement back into your program!). In the MATLAB command window you will see:

```
??? Subscript indices must either be real positive integers or logicals.

Error in ==> C:\MATLAB6p5\work\vectorgrowth.m
On line 4  ==>   popsize(time)=popsize(time-1)*r % the instruction to be
repeated
```

The first line of the above code tells you the command that is causing problems for MATLAB. 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 in the error statement tells you which file has the problem, which is our `vectorgrowth` file. The third line in the error statement actually tells you which line in the file contains the error. This helps you quickly track down most errors. When you get an error, take a breath and read what MATLAB is telling you. Many people get frustrated by the error and annoyed by the redness. Don't. Reread what MATLAB writes to your screen and correct the error.

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, that `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 = 1:3 % defining the start and end of the loop
  popsize(time+1)=popsize(time)*r % the instruction to be repeated
end % 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 a new window with the graph, which we will call the figure window.

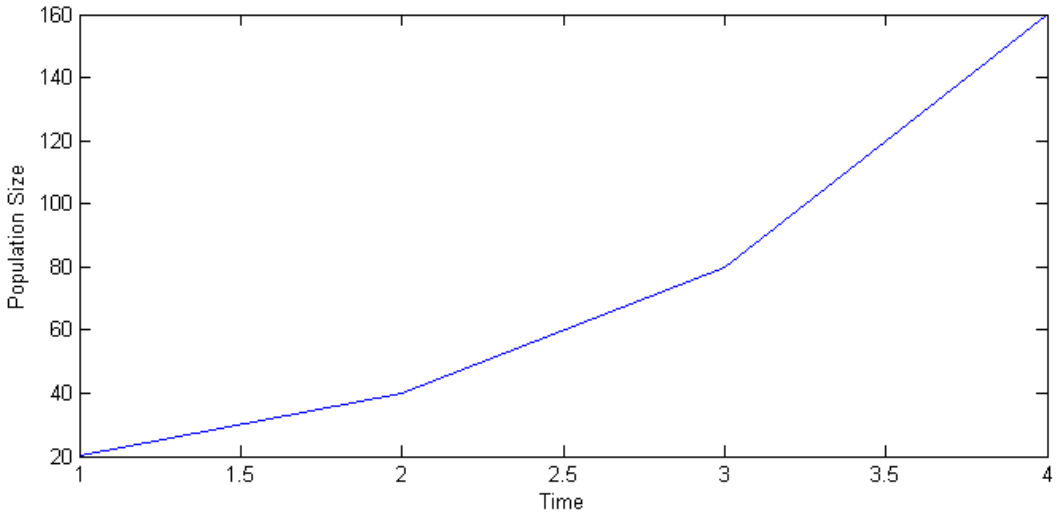

Figure 2. Population size plotted against its position in the `popsize` vector.

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 MATLAB 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 = [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 = [0:1:1000];
```

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

```
>>xvalues = [0:1:3];
```

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([0:1:3], popsize);
```

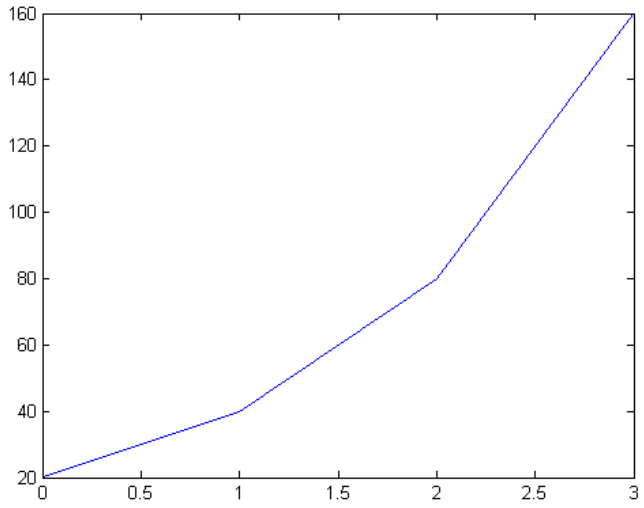

Figure 3. Plot of population size against time step.

This corrects the plotting problem and will produce a graph where the initial values is plotted at zero on the x axis. The plot command can be included at the bottom of our MATLAB 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([0:1: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. For a column vector, `length` will return the number of rows (there is only one column). For a row vector, length will return the number of columns. We save the number of columns in `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 everytime 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 `[0:1:numiterations]`.

MATLAB opens a new window for graphics and this window can be used to add title, axes labels, change color, etc. 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.

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 = 2:1:10 % defining the start and end of the loop
  %this section modifies the growth rate depending on the year
  if (mod(time,5) == 0)    %this determines whether it is a great year
          r = 4;
  else % or just an ok year
          r = 2;
  end
  popsize(time)=popsize(time-1)*r % the population growth equation
end % 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 `mod(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 `mod(time,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 ride 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. MATLAB will often tell you that you have made a mistake:

```
??? Error: File: C:\MATLAB6p5\work\vectorgrowth.m Line: 5 Column: 19
Assignment statements do not produce results. (Use == to test for equality.)
```

Octave warns you to use `parenthesis around assignment used as truth value` near the error.

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 of the above sample gives us

```
if (mod(time,5) ~= 0)
    r = 2;
else
    r = 4;
end
```

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;
end
```

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;
end
```

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;
    end    % this ends the (time >= 10) if else statement
end    % this ends the (time <= 6) if else statement
```

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 `end` 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 the `end` statement will help you keep everything in order. Despite getting the job done, our example is more confusing than it needs to be. MATLAB 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;
elseif (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;
end % 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, it 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;
end
```

In this case, our two `if` statements were combined into one using the ***or*** connector (|). 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;
end
```

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 taking the growth equation described in section 2, Variables and Mathematical Operations, and solve it analytically. We want to answer this question using MATLAB 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 <= 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
end % closing the loop
```

The first three lines defines 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 `end` statement 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` (a 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. While the MATLAB window is selected, hold down the Ctrl button and hit the Break key (usually coupled with Pause button, found near the Scroll Lock button in the upper right hand section of the keyboard). This will cause MATLAB to immediately quit whatever code it is running. In Octave, you can hold down the Ctrl button and hit the letter C to stop code, although this command may vary by operating system. 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 `size` command, `plot` command, and `mod` 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 = 2:1:4 % defining the start and end of the loop
  popsize(time)=popsize(time-1)*r; % the instruction to be repeated
end % 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([0:1: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 on the first line of the file. In this case, we are going to create a function named `discretegrowth` and therefore must also name our script file this. As stated before, we will modify three variables (parameters for our growth equation) within our function. We need to tell MATLAB 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.

```
function d = discretegrowth(r,initial,howlong)
```

This line tells MATLAB to create a function called `discretegrowth` that will use three variables that will be defined by the user (`r`, `initial`, `howlong`). After the function is complete, it will 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.

```
function d = discretegrowth(r,initial,howlong)
%Notice that the line below is commented out.
%r=2; % growth rate
popsize = initial; %defining initial conditions

for time = 2:1:howlong % defining the start and end of the loop
```

```
  popsize(time)=popsize(time-1)*r; % the instruction to be repeated
end % closing the loop

numiterations = size(popsize,2);    % the number of values in popsize
numiterations = numiterations - 1;    % because we start with an initial
                                                    % condition at
popsize(1),
                                      % we run one less year than popsize
plot([0:1:numiterations], popsize)    % this will plot everything nicely
d = popsize;                          % function returns popsize vector
```

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 that line out, the function would have used a growth rate of 2 every time you used the function. Save this file as discretegrowth.m.

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 MATLAB, 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);
```

With this, two graphs should appear. 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. Close out the graphs and 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. Also, the only way to get a variable back from a function is by having the function return a value. We have `d = discretegrowth(r, initial, howlong)` in the first line of our function file. 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)

$$\frac{dN}{dt} = r\,N$$

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.

MATLAB provides tools for solving and simulating 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 an ODE in MATLAB can be done for simple 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 as MATLAB functions, making our lives easier. To simulate an ODE we need two files; one will have the form of a script, and the other will contain a function. First, we need to translate our equation into MATLAB code as a function. Create a new MATLAB M-file named expgrowth.

As this is a function, we need to define the function on the first line of the file. Remember, the file name has to be the same name as the function name.

```
function dy = expgrowth (t,y,r)
```

We are not only defining the function, but also defining the three parameters that our function needs: `t` refers to time, `y` to the previous solution, and `r` is the growth rate of the population. Other letters can be used, but we will stick to this notation, as it is found in the MATLAB documentation. To understand how to convert our differential equation into MATLAB code, it maybe be easier to see a 'flat' notation of the equation:

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

The variable `r` has already been given to the function, as has the value of `N`, although we defined it as `y` from the function definition. As you notice, we have two variables on the left side of the equals sign, `dN` and `dt`, which will cause problems. In this case, we are going to represent the rate of change (`dN/dt`) as the single variable `dy`. With respect to `y` and `dy`, MATLAB is expecting a vector, but we only want it to use the first value in the vector, therefore we will use `y(1)` and `dy(1)`. This is a requirement of the ODE solver itself and not something we should be worried about. Making all our substitutions, our 'flat' equation becomes:

```
dy(1) = r * y(1);
```

Add this to your MATLAB file expgrowth. Including comments, your file should look like this:

```
function dy = expgrowth (t,y,r)
%This function defines the differential equation to be simulated.
% the file that actually simulates the ODE is simulexp.m
dy(1) = r * y(1);
```

Create a new M-file, this one named `simulexp`. The script `simulexp` that runs the simulation can be divided into three parts: parameters for the ODE, parameters for the simulation itself, and simulating the equation. In the file `simulexp`, write a comment to identify the file and define `r`, the growth rate.

```
%this file simulates the ODE found in file expgrowth.m
% parameters for the ODE
r = 2.0; %This defines the growth rate
```

The next section defines parameters for the simulation itself. Two parameters are needed: the initial condition (`ic`) and the time span for the simulation (`timespan`). The initial condition is a number representing the initial population size, initialized here with 10 individuals. The time span is defined as a vector with two entries: the start and endpoints of the simulation. In our case, the time span will start at 0 and go to 10.

```
% parameters for the simulation
ic = 10; % initial population size
timespan = [0 10]; %vector of start and end time points.
```

Now that all parameters are defined, the code that actually simulates the ODE can be written. This example will employ the function `ode45`, a general simulator of ODEs. As expected, the function `ode45` requires the name of the file defining the ODE (`expgrowth`), the initial condition (`ic`), and the time span (`timespan`) as parameters. The growth rate `r` is an additional parameter required for our specific differential equation.

```
% this script simulates the ODE
[t,y] = ode45(@expgrowth,timespan,ic,[],r);
% next we plot our results and label the axes
plot(t,y)
xlabel ('Time')
ylabel('Population density')
```

When calling the function `expgrowth` within the function `ode45`, one must prepend an @ before the function name (called a handle). The empty square brackets indicate an empty set of options for `ode45`, thus accepting the default configuration. If you are interested in the many options available in the ode45 simulator, type `help odeset` at the command prompt in MATLAB. Help for odeset is not available in Octave. After you have defined the options or accepted the default values for the function `ode45`, then the additional parameters needed by `expgrowth` can be specified. In our case, we have the single variable `r`, the growth rate. Without the empty square brackets, the growth rate will be interpreted as an option for ode45 and an error will occur as "Input argument 'r' is undefined." Forgetting the @ before `expgrowth` also produces a similar error.

The output of the function `ode45` will be two vectors: vector `t` will contain all timesteps employed in the simulation, and the `y` vector will contain the corresponding population size. To plot the simulation, give the output vectors `t` and `y` as arguments to the plot function. In previous sections, we have only had one variable on the left hand side of the equals sign. With functions in MATLAB, you can have them output any number of variables and vectors. We need to define enough variables on the left side of the equals sign to catch all the values we are interested in produced by the function on the right hand side. Note the use of square brackets if we define more than one variable. If you are using Octave (tested with version 3.0.5), you need to replace the line

```
[t,y] = ode45(@expgrowth,timespan,ic,[],r);
```

with

```
[t,y] = ode45(@expgrowth,timespan,ic,odeset(),r);
```

You also need to make sure that you have the Octave odepkg installed. If you are not sure, type in `help ode45` on the command line and hit `enter`. If an error occurs telling you `help: ode45 not found`, you need to install the ode package in Octave. See http://octave.sourceforge.net/ for help on installing packages for Octave.

Go to the command prompt, type in `simulexp`, and be amazed at your abilities (Figure 4).

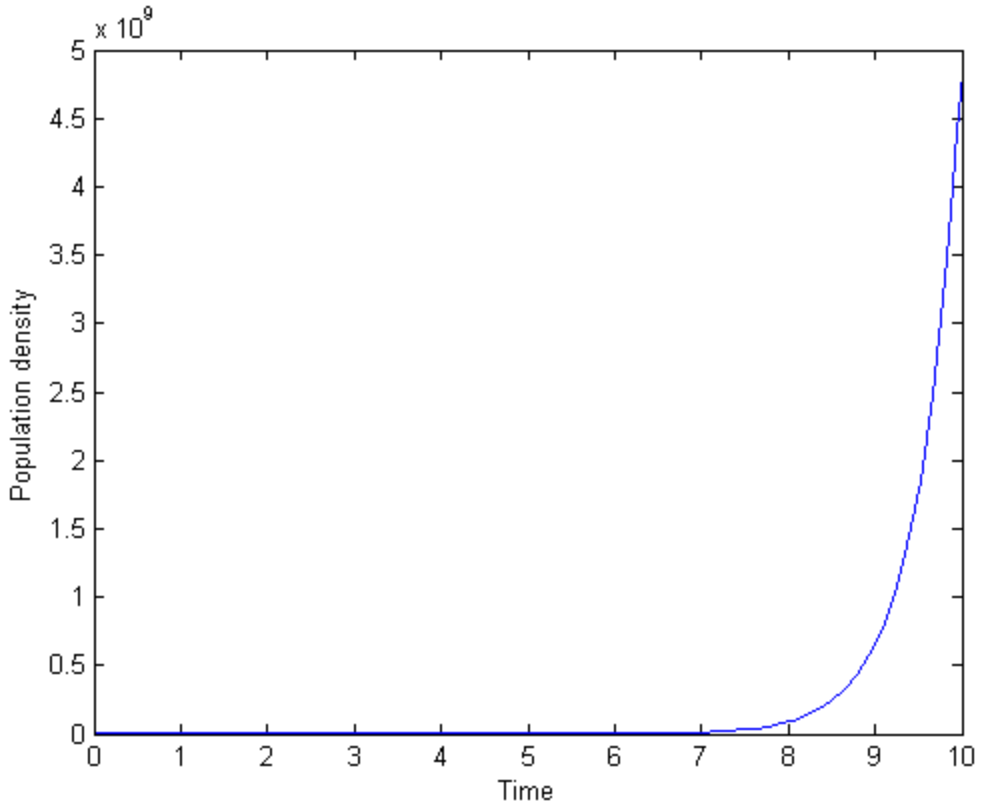

Figure 4. Population density through time of the continuous population growth equation. Solved using an ODE solver.

A common error with respect to simulating continuous differential equations as well as when producing larger, multi-file programs is not having all the required files in the same folder. Move the file expgrowth.m out of the current folder it is in. Now run the simulexp program. You will get a undefined function error.

```
??? Error using ==> feval
Undefined command/function 'expgrowth'.
Error in ==> C:\MATLAB6p5\toolbox\MATLAB\funfun\private\odearguments.m
On line 104  ==> f0 = feval(ode,t0,y0,args{:});
Error in ==> C:\MATLAB6p5\toolbox\MATLAB\funfun\ode45.m
On line 155  ==> [neq, tspan, ntspan, next, t0, tfinal, tdir, y0, f0,
args, ...
Error in ==> C:\MATLAB6p5\work\simulexp.m
On line 8  ==> [t,y] = ode45(@expgrowth,timespan,ic,[],r);
```

Another way to phrase the first bold line is that there no file called expgrowth.m in the current directory, as functions and their files have to have the same name. The current directory is displayed in the top of the MATLAB window (immediately above Section 1 in Figure 1). At the bottom of our Workspace window are two tabs that allow us to switch between the Workspace and the Current Directory (Section 2 in Figure 1). By clicking on the Current Directory tab, all the MATLAB files within that directory are displayed in the window. Notice that there is no expgrowth file within the directory, as we told you to take it

out. The second and third bold lines pinpoint the error by showing the file name and line number where the offending instruction can be found. Inspecting the instruction reveals no error such as misspelled file name or missing @ symbol, and therefore one should think the expgrowth file is missing. The rest of the error output is not relevant because the default ode45 arguments are used, and there is no reason to expect that ode45 is buggy.

This same error occurs if the filename is misspelled. To see this, restore the expgrowth.m file back to the MATLAB's current directory. Then change the file name, for example by deleting the letter h. You need to then remove the correctly spelled expgrowth.m from the current directory. Then run the simulexp.m script again. Yet another way of getting the same error is as follows: restore the file name to its correct form (putting back the letter h), edit the simulexp.m file and delete the letter h in line 13, then run again the simulexp.m file. As always, do not be afraid of the red lettering. When MATLAB gives you an error, take the time to read and understand what MATLAB is telling you.

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 program. 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 MATLAB. 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).

To learn more about MATLAB specifically, one only has to view the documentation included with the MATLAB programing language (in the MATLAB window, click on Help->MATLAB Help), assuming you installed it when installing MATLAB initially. Documentation on the Mathworks website (www.mathworks.com) is also useful (click on Support and then select MATLAB from the product list). As a reference manual, we recommend Duane Hanselman and Bruce Littlefield's *Mastering MATLAB*. Any version will work as the basics of MATLAB do not change drastically from one release to the next. Do not read it to learn how to program, but read it to learn about what other functions and options are available to you in the MATLAB programming environment. For a list of errors and solutions for MATLAB, please see the Wikibook *MATLAB Programming* Error Messages section(http://en.wikibooks.org/wiki/MATLAB_Programming/Error_Messages) and read the rest of the book for other nuggets of information. The programming language Octave has Eaton's GNU Octave Manual, which is found at http://www.gnu.org/software/octave/docs.html or by clicking Docs on the Octave main website.

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. *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 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 and differential 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.

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.

Eaton, J.W., Bateman, D. and Hauberg, S. 2002. GNU Octave Manual. Network Theory Limited. ISBN: 0-9541617-2-6.

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

Hanselman, D. and Littlefield, B. 2001. Mastering Matlab 6. Prentice Hall, New Jersey. ISBN: 0130194689 http://www.ece.umaine.edu/mm/

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

MATLAB. The Mathworks Inc. Natick, Massachusetts. http://www.mathworks.com/

MATLAB Programming. Wikibooks. http://en.wikibooks.org/wiki/MATLAB_Programming

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

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

Octave. http://www.octave.org

Octave-Forge. Packages for Octave. http://octave.sourceforge.net/

Roughgarden, J. 1997. Primer of Ecological Theory. Prentice Hall. ISBN: 978-0134420622