An Ecological Modeler's Primer on R

N. Thompson Hobbs

January 16, 2017

Natural Resource Ecology Laboratory and Graduate Degree Program in Ecology, Colorado State University, Fort Collins CO, 80523

Contents

1 Rationale and approach **6**
1.1 Why programming? . . . 6
1.2 Why R? . . . 6
1.3 Teaching approach . . . 7
1.4 A learning attitude . . . 8
1.4.1 An R notebook . . . 8
1.4.2 A folder containing useful code . . . 9
1.4.3 A log of your work . . . 9
1.4.4 Version control . . . 9
1.5 About this primer . . . 9

2 Installing R **10**

3 The R environment **10**
3.1 The console . . . 11
3.2 Setting up your working directory . . . 11
3.3 Writing programs . . . 12
3.4 Commenting your code . . . 13
3.5 The console remembers . . . 14
3.6 Saving your work . . . 14
3.7 Getting help . . . 15
3.7.1 Introduction to R . . . 15
3.7.2 Getting help from the console . . . 15
3.7.3 Searching list serves . . . 16
3.7.4 Third party books . . . 16
3.7.5 Googling an R topic . . . 16

4 Basic operations in R **17**
4.1 Assignment . . . 17
4.2 Operators . . . 17
4.3 Numbers . . . 18
4.4 Order of operations . . . 18
4.5 When does a variable exist? . . . 19

5 Data in R **20**
5.1 Scalars . . . 20
5.2 Strings . . . 20

5.3 Vectors . . . 20
5.3.1 Vector basics . . . 20
5.3.2 Mathematical operations on vectors . . . 21
5.3.3 Logical operations on vectors . . . 21
5.3.4 Vector functions . . . 22
5.3.5 Declaring vectors . . . 23
5.4 Matrices and arrays . . . 23
5.5 Creating matrices . . . 24
5.6 Sub-setting matrices . . . 25
5.6.1 A common error when creating matrices . . . 27
5.7 Arrays . . . 27
5.7.1 Functions for arrays and matrices . . . 28
5.7.2 Matrix algebra . . . 29

6 Programming in R **29**
6.1 Iteration . . . 29
6.1.1 The basics of for loops . . . 29
6.1.2 For loops controlled by sequence vectors . . . 32
6.1.3 Nested loops . . . 33
6.2 Functions . . . 34
6.2.1 Why use them? . . . 34
6.2.2 Function basics, example 1 . . . 34
6.2.3 Function basics, example 2 . . . 35
6.2.4 Scope of Variables . . . 40
6.3 Debugging . . . 41
6.4 Returning more than one thing . . . 42
6.5 Using functions for structured programming . . . 43

7 More about data **45**
7.1 Lists . . . 45
7.2 Getting Data into R . . . 48
7.2.1 Formatting Excel Files for R . . . 48
7.2.2 An important caution about importing files from Excel . . . 49
7.3 Data Frames . . . 49
7.3.1 Accessing components of data frames . . . 50
7.3.2 Accessing data frames using `attach( )` . . . 51
7.3.3 Why not always use `attach( )`? . . . 52
7.3.4 Accessing data using `with( )` . . . 52

7.4 Manipulating elements of data frames . . . 53
7.4.1 Joining . . . 53
7.4.2 Merging . . . 54
7.4.3 Ordering data frames . . . 54
7.4.4 Editing data frames . . . 55
7.4.5 Dealing with missing values . . . 55
7.4.6 Converting arrays and matrices to data frames . . . 56

8 More about plotting **57**
8.1 Controlling Output to the Plot Window . . . 57
8.2 Options for plotting . . . 58
8.2.1 Specifying the type of plot . . . 58
8.2.2 Adding labels . . . 58
8.2.3 Scaling the axes . . . 59
8.2.4 Mathematics, Greek symbols, subscripts and superscripts . . . 59
8.2.5 Adding headings . . . 60
8.2.6 Changing symbols and line types . . . 60
8.2.7 Multiple plots on a page . . . 60
8.2.8 Specifying options in one statement . . . 61
8.2.9 Overlaying points and lines on a basic plot . . . 61
8.2.10 Adding legends . . . 61
8.2.11 Exporting graphics to documents . . . 62
8.3 Some other plotting functions . . . 62
8.3.1 Plotting columns of matrices as series . . . 63
8.3.2 Plotting mathematical expressions . . . 63
8.4 Making publication quality output . . . 64

9 Special topics **64**
9.1 Making tables . . . 64
9.1.1 Output to spreadsheets . . . 64

10 Answers to exercises **64**
10.1 The diameter of the earth . . . 64
10.2 Vectors . . . 65
10.3 Logical operations on vectors . . . 65
10.4 Creating a matrix . . . 65
10.5 Matrices . . . 65
10.6 Matrices, continued . . . 66

10.7 3 dimensional arrays 66
10.8 A Gompertz model of plant growth 66
10.9 Nested for loops 67
10.10 Function for normalizing a vector 67
10.11 Functions for moment matching 68
10.12 Lists 69
10.13 Lists and functions 69
10.14 Manipulating data frames with $ and [ ] 70
10.15 Using attach and detach 70
10.16 Replacing NA's 70
10.17 Exploring the Gompertz 71
10.18 The final problem 71

List of Algorithms

1 Illustration of `for` loop 29
2 `for` loop including storage of state variable in a vector. 30
3 Example code for a function using an `if()` statement to control execution . 36
4 Example code for creating a table for importing into a spreadsheet. 64

List of Figures

1 The R environment under RStudio. 11
2 Illustration of command for subsetting a matrix. 72
3 Illustration of `x-y plot using legend()` and `plot()` options. 73

List of Tables

1 Basic mathematical operations in R 18
2 Logical operations in R 22
3 Frequently used functions that operate on vectors in R. In the examples below, `z` is a vector. 22
4 Extracting elements of data frames. 50
5 Extracting elements of data frames. 51
6 Some codes for symbols and line types used as options in R plotting statements. 60

1 Rationale and approach

1.1 Why programming?

This course is about building models and using them to gain insight from data. Why do you need to learn to write computer programs to do so? After all, there is software that allows you to point and click your way to reasonably sophisticated models of ecological systems. Using these tools, you never need to struggle with the inevitably arcane syntax of modern programming languages. So why use a programming language to build models and analyze them, which seems like an awfully lot of trouble, rather than using a point and click software? The reason is simple: there is no more adaptable, flexible skill in modeling, or in science for that matter, than being able to write your own code. Moreover, unless you know how to write computer code, you will have a great deal of difficulty fusing models with data, which is why you are here.

The other reason to learn a programming language for your data analysis and modeling is arguably more important. Sophisticated analysis in ecology requires lots of steps—data must be reformatted and manipulated, analyses require several operations, and you must be able to reproduce *exactly* what you did to obtain a given result. This is not possible with point and click approaches to analysis—although convenient and easy to learn, these provide no written history of the steps you used. Spreadsheets are notorious in this regard, following the logic of your analysis can be difficult because of multiple dependencies among cells. In contrast, computer code provides a sequential, line by line record of what you did. Every step is explicitly represented in the programs you write. There is absolutely no ambiguity about how a result was achieved. This is critical for careful science. So, one of the most important skills of modern ecologists is computer programming. You will learn how in this course.

1.2 Why R?

For almost a decade, I taught this course using Visual Basic for Applications running under Excel. There were many advantages to this approach, not the least of which is that Excel is something that students are comfortable with, but I finally decided that modern students need to know R even though it is, well, a bit uncomfortable. Why this choice? The reason you should learn R is simple: it is the premier language for scientific computing of all sorts, but it is especially well suited to the purposes of this course, model-data integration. R was originally written as a language for statisticians, but it has become much, much more than that—it is a language for scientists. It has all you need to write models of all kinds; estimate their parameters, and compare one model against another—the meat of this course.

There are other reasons to learn R; it is free and open source. Because it is free, you can take it wherever you go with you at no expense, which, of course, is not true of commercial software. Because it is open source, some of the best minds in the world are at work every day writing all kinds of code that will be useful to you in the future. This means that R is constantly getting better, more useful, and more powerful. It is, perhaps, a contradiction of the usual free market dogma that software produced for free by academics surpasses software

that is produced at a cost for academics. This is the way of all sorts of open source software, but nowhere is it more clear than with R. The other reason is that R is "one stop shopping." It is great for almost everything you need to do in research–simulation, data manipulation, statistics, and graphics. Most of what you require with a programming language can be done in R as well or better than it can be done elsewhere. In my experience, the one area relevant to modeling where other software surpasses R is in symbolic computation, which is not emphasized in this course. In this case Maple, Matlab, Derive, and Mathematica are superior.

There are some other downsides. R executes much more slowly than some other computer languages (notably C, C++, and FORTRAN) but no slower than other "interpreted languages" (e.g., Matlab, Maple, VBA)[1]. If you write big models with hundreds of parameters, a thousand lines of code, and that must be run over thousands of pixels for hundreds of years, well, you would be nuts to use R. In that case, you will need to migrate to a faster language. But the concepts that you learn here will serve you well as you learn other computer languages. In my experience, once you have learned one programming language, the next one is easy.

The other downside is that R can be difficult to learn as a first language, at least if you are working on your own. It is difficult for three reasons. First, languages that are flexible and powerful always require more struggle to master than those that are rigid and weak. R is the former—it has ways of doing things that are compact and fast, but that will appear bewildering to the novice. Second, the documentation is a bit diffuse[2] and, for the beginner, is hard to use. That said, there is nothing that I have wanted to do with R that I have not been able to figure out with a little effort and patience. Finally, there is so much depth to R that it is really hard to know where to start. That is where I come in. If I am a good teacher, and, immodestly, I think I am, then I can make the learning processes easier for you by identifying and explaining key bits, by focusing your learning on what you really need to know to move forward. I promise that you will struggle a bit and will be frustrated at times. Your struggles will be worth it. The R you learn in this course will not be comprehensive, but it will give you a sturdy foundation for self teaching, a foundation upon which you can build over your entire career, or at least until some better language comes along, which it probably will.

1.3 Teaching approach

Self-teaching is one of the single most important skills of people who work in technical fields like science and engineering. These fields change rapidly, and the pace of change means that your ability to compete depends in a fundamental way on your ability to learn new things

[1]Interpreted languages cannot produce files with an .exe extension, while compiled languages cannot. Compiled programs are faster because their instructions can be directly understood by the computer. Interpreted languages require, well, an interpreter for the computer to understand. Think of a conversation that is carried out in your native tongue vs. one that requires an interpreter. The difference is magnified in computing—complied code is 10 to 100 times faster than interpreted code.

[2]This is changing as a small cottage industry produces well-written, well-organized texts on R and scads of great tutorials on the web.

by yourself. Computer languages offer a great opportunity to practice self-teaching, because programming offers the ideal way to "learn by doing."

You can do a huge amount of work with a relatively small amount of knowledge of most contemporary programming languages. This is because they are versatile—you can handle an incredibly broad range of tasks using a single skill. However, this versatility also creates some challenges for learning because you only need to accomplish a relatively tiny subset of all of the tasks that contemporary languages were built to handle. As a result, there is a formidable amount of material that is largely irrelevant to your needs as a modeler. This can make learning difficult, because you don't know where to start.

Here, I offer some training in R to provide a focus for your own self-teaching. As with other parts of this course, the objective is to offer a foundation for your own learning by doing. My objective here is to sort out the things that you need to know about R (to build models and integrate them with data) from the rather immense amount of stuff that you don't need to know (all the huge number other things that R does). R is a deep, read bottomless, topic, but you can learn enough to do real work with a fairly brief initial investment of time and effort if your investment is properly focused. That said, remember the job of learning is never done. Every time you do a programming task, you will probably learn something new about R. I do.

1.4 A learning attitude

Learning R proceeds most rapidly if you work with a bit of curiosity and willingness to experiment. You aren't't going to break anything; there is always the undo key, and your progress will be more rapid and fun if you approach every programming problem with learning as one of your objectives. The R console, which you will learn about shortly, facilitates this sort of exploratory learning. Believe me, this course will be a genuine chore if every time you sit down to program, the only reason you do so is to get an assignment done. Instead, make every assignment a chance to explore a fantastically powerful tool.

This exploration will be most useful to you if you develop three habits: 1) Write a notebook on R, 2) Make a directory containing bits of particularly useful code and, most importantly, 3) Always, always, always, keep a log of any serious work and, as you mature as a programmer and take on more complicated projects, use version control. Let me describe each of these in turn.

1.4.1 An R notebook

A notebook on R contains a set of entries on what you have learned about the language that you think is not covered well in the documentation. There is not need to rewrite what is already documented but if you figure out a cryptic error message, write down what you did, what the message was, and how you fixed it. If you discover how something works, write down your discovery in your R notebook. If the documentation is not clear on some point and you figure it out, write it down. If you discover an informative website or pdf file on a topic, include that in your notes as well. Believe me, you are going to forget many of

these things and your R notebook will prove, over and over and over again, to be a vitally important reminder. Read it from time to time and search it often.

1.4.2 A folder containing useful code

My computer contains a folder called R where I keep my R notebook, useful files on R from the web, and so on. There is also a subfolder named Useful Code where I keep a set of R scripts for things that I do often, but not often enough to remember in detail. It is usually easier to remember how to do something that you have done before than to consult the documentation, so I put bits of code that I want to remember and use again in this folder. By far the bulk of my code is organized under folders for specific research projects and it can be hard to remember where you put a script with a particularly nice trick. So those go into the Useful Code folder.

1.4.3 A log of your work

This is a critical part of any serious analysis or modeling project. It is analogous to the laboratory notebook you learned to keep for bench work in biology and chemistry. The idea is that you keep a file with a master list of the names an locations of all files relevant to the project you are working on and daily, dated entries saying what you did on the project and what you accomplished. I cannot overestimate the importance of work logs to careful analysis of models and data.

1.4.4 Version control

It turns out that a more modern way to document your work, a supplement to your notebook, is version control software, a tool that keeps track of every change you make in your code. This is a bit advanced for what you are doing now, but I strongly urge you to learn some type of version control software. It is essential for serious work. The best known is probably Git. The command line version is free and can be obtained at `https://git-scm.com/`. Excellent tutorials are found there. Another good introduction is `https://www.git-tower.com/learn/`.

1.5 About this primer

First, a bit of education on the word primer. The definition of primer as I use it here is a book to introduce a topic. Primers were originally school books that taught reading using simple stories. The proper pronunciation of primer in this context has a short i, the vowel sound found in "big". Primer as it is used here is not the same as the word you would use to describe a first coat of paint or a blasting cap, which is pronounced with a long i, the vowel sound in "high".

In the best tradition of primers, this one will be organized with lessons interspersed with exercises. It is absolutely critical that you work through each exercise in sequence, mastering it before proceeding. The exercises become increasingly challenging and if you move too fast,

you will eventually fail to understand what is going on. All of the exercises have answers provided at the end of this document. It is ok to look at these after you have struggled on your own, but you must really understand the answer before you proceed.

You will be tempted to cut and paste examples from the document into R programs. This is not a good idea for a very simple reason. You will make errors in entering R commands and learning how to figure these out is a critical, if painful, part of learning the language.

I will try to be consistent throughout in the use of fonts:

Times roman will be used as the main text.

Sans-serif will be use to describe menu choices in RStudio.

`Typewriter will be used to refer to specific R commands.`

2 Installing R

Ok, it's free. How do you get the software? Downloading R is easy. Go to http://www.r-project.org/. Follow your nose starting with clicking on CRAN under Download. You will be asked to choose a site from which to get files. Picking any of the U.S. sites is fine. You will then need to choose among operating systems. Follow your nose from there.

R loads with a base program and a few "libraries." Libraries aka pacagages contain code for special tasks that everyone who uses R may not need. Rather than load all the software, you are given the option to simply include parts that are appropriate to your work. It would be good to load the reshape and popbio packages for this tutorial. Go to the Tools dropdown and follow your nose to load these now. We will get others as needed.

You will also need the RStudio Desktop editor. Download from `https://www.rstudio.com/products/rstudio/download3/` [3]

3 The R environment

We will use R Studio as an editor for all programming in this course.[4] I will not go into great detail about the features of RStudio, many of which we will not use. Excellent tutorials can be found on the web if you need an in depth treatment (see for example, `https://support.rstudio.com/hc/en-us/sections/200107586-Using-RStudio` . I will provide a brief tutorial on using R Markdown, which is the main motivation for using the RStudio editor.

[3] You are free to use whatever editor you like, but if you want my help using R Markdown, I urge you to use RStudio.

[4] There are two reasons. I will often need to sit down at your machine and work with your code to give you the help you need. If you are working in `vi` (most of you won't know what `vi` *is)* then, well, I am a bit rusty. You get the idea. Suffice it to say that I have also found that it is easier for the TA and me to help you in labs if we can work in a single, consistent environment. The second reason is that R Studio supports R markdown, which you will need to produce lab reports.

3.1 The console

Now that you have successfully installed R and Rstudio, start your first session by clicking on the RStudio shortcut. Three windows will open, the *console* is the one on the left (Figure 1). The console is distinguished by the prompt >. It accepts input directly (at the >) and takes input indirectly from scripts (more about scripts in a moment). All of the output resulting from R commands shows up in this window. To illustrate direct input and output at the console prompt, type in few calculations, for example 4 + 13^ 3, sqrt(9), 56*4. Your results should look something like

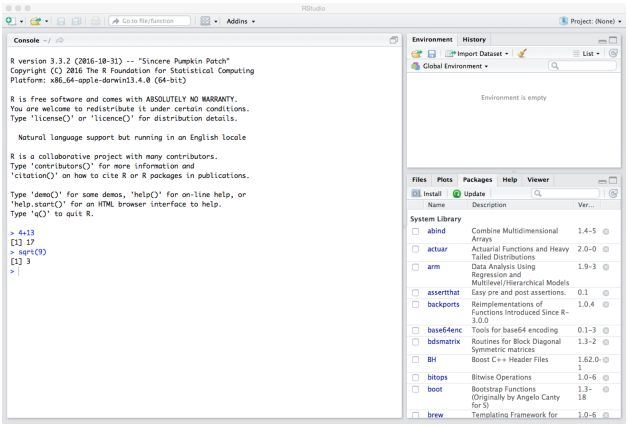


Figure 1: The R environment under RStudio.

```
>
> 4+13^3
[1] 2201
> sqrt(9)
[1] 3
> 54*6
[1] 324
```

with output from each command given next to a [1]. This illustrates R's features as a pocket calculator, which is something we will rarely exploit in this course. However, before I illustrate a feature we will use heavily, one more point about entering commands from the console. It has a memory of what has been entered. You cannot "go up" the stack of commands with the mouse and use them, but you can bring them back to the prompt. For example, with the cursor adjacent to a blank line adjacent to >, press the up arrow key 3 times, then the down arrow key 3 times. You will see statements that you have entered before appearing at the prompt. When a statement is beside the prompt, it can be edited using the left and right arrow keys and entered with, well, enter. If you want to execute 2 statements on the same line, which can be handy for recalling and editing, simply separate them by a semicolon.

3.2 Setting up your working directory

As with all projects, it is helpful to organize your work into folders. There is a class GitHub repository `ESS_575_2017`. You *must not* put any of your own work in these folders because your work will be lost whenever I push changes to the repository. Instead you can create a subfolder of `ESS_575_2017` called something catchy like `MyWork` with subfolders for labs and lectures. Alternatively, you might want to copy the subfolders `Admin, Labs, and Lectures` to some other folder on your disk and repeat that copy each time the repository is updated,

again presuming that none of your own work is stored in these folders. [5] Now put some file, any file including a blank one, in the folder where you want to store your work. Once you have done this, type in an exceedingly useful command at the prompt:

```
> file.choose()
```

R will respond with finder in Mac OS and Windows Explorer in Windows. Navigate to your working folder and click on the file that is in it. R responds something like

```
> file.choose()
[1] "/Users/Tom/Documents/MyESS575/Code/Lab1/Untitled.R"
```

Now select everything in the path except the file name and paste it, with quotes into the following command at the >

```
> setwd("/Users/Tom/Documents/MyESS575/Code/Lab1/")
```

Quotes are mandatory. Now the "working directory" is ("/Users/Tom/Documents/MyESS575/Code/Lab1/". When you save or output files, they will go here without specifying the path name.
To check that this worked, use

```
>getwd()
```

There is an important caveat here. You will ***not set a working directory*** if you are working in RMarkdown. Instead, the working directory will be set to wherever you save the R Markdown document. But we are getting a bit ahead of ourselves. More about this soon.

3.3 Writing programs

Most of your work in R will be done with programs, or strictly speaking, scripts. There are two types, plain R scripts and R Markdown documents. You will learn about R Markdown documents soon enough, but for the remainder of this Primer we will be using scripts. [6]. To illustrate, go to File and click on New file then R script. A window opens that allows you to type in programs, collections of statements that can be submitted entirely or in bits and saved for later use. For example, enter the following lines of in your script window:

```
r = .2
N0 = 100
N1 = N0*exp(r)
N1
```

[5] Notice I have left out spaces in folder names, also a good practice for filenames. This is because UNIX doesn't like spaces and UNIX tools are incredibly powerful.

[6] Strictly speaking because R is an interpreted language and the instructions given to its interpreter are called scripts. The instructions given to a compiler (for complied languages like C) are, strictly speaking, called programs. But I rarely make this distinction.

Now do two things. First, put your cursor at the top of the file and press cntrl-enter if you are using Windows R (cntrl-enter or cmd-enter if you use a Mac[7]) at each line, observing the console each time. Your output should look something like:

```
> r = .2
> N0 = 100
> N1 = N0*exp(r)
> N1 [1] 122.1403
```

Note that when you entered `N1` at the console, R returned its value. If it had not been assigned a value then R would return:, `Error: object "N1" not found`. To illustrate, enter `N0` (which has been assigned the value 100) then `N2` (which has not been assigned any value. More about this message shortly.

Now select the first two lines with the mouse and again press cntrl-enter and observe what happens. Next simply press ctrl+shift+enter. You have now learned 3 ways to input from a script to R: one line at a time, a few lines selected, or all of the lines in the widow selected. All three of these methods will be used heavily in this course.

3.4 Commenting your code

This is important. R allows you to make little notes to yourself with code to help you remember what you did when you come back to a program after six months. Or, these comments allow someone else to understand what you are doing. Making a habit of commenting your code is one of the best things you can learn in this course. To add a comment, simply put a # in front of what ever your write. So, for example, to add appropriate comments to the code you just wrote:

```
# intrinsic rate of increase
r = .2
# initial population size
N0 = 100
# Calculate new population size after one unit of time
N1 = N0*exp(r)
# Output new population size to console
N1
```

Ok, ok, ok—these are baby comments. But you get the idea. The comment symbol can also be useful to prevent a line of code from executing We will talk about this as the opportunity arises in lab. No programmer likes to take the time to write comments and I am not terribly good at it. However, if you fail to do so, "the future your will be really mad at the past you."

A better way to document programs is to create R Markdown documents, which you will learn about shortly. However, you will still used comments in these documents.

[7]To make life simple, I will use the cntrl-enter syntax but those using Mac's can use the more familar cmd wherever cntrl appears.

3.5 The console remembers

As long as your R session is running, R remembers everything that you entered into the console. To find out what R is remembering, enter ls() at the console. Remembering can be a good thing in that it allows you to work interactively entering commands at the console or to write short scripts that depend on one another without including all of the commands in one single, large script.

However, remembering can also cause problems. If you write a script that depends on the state of R's memory when you write it, you may not be able to reproduce the script if the state of the memory changes. For example, there may be variables that must be declared for your script to run and these declarations are in memory but not in your script.

So, remembering can actually be a problem when you do your work in scripts, which you almost always will. Stand alone aka "stateless" scripts, that is, scripts that have no dependence on the state of R's memory, are the best way to document your work. When your work is done at the end of the day, you have a clear written record of what you did, with comments of course, in an R script file. To assure that my script can run without any assumptions about what R remembers, I put the following statement at the top of all of my programs:

```
rm(list = ls())
```

This makes R forget everything before the script runs. I want my scripts to run from scratch, without depending on things I did in the console and forgot to write down. For now, just remember this statement exists and what it does. Once you begin to write scripts, you will probably want to use it.

3.6 Saving your work

RStudio has a nice auto save feature that helps in case of a system crash, but you need to be in the habit of saving your script files often while you are working. This can be done by navigating to the window where you are working on a script and pressing cntrl-s or going the File menu and pressing Save or Save as. You can name them anything you like, but you shouldn't change the R extension. This will allow R to see them when you click on Open in the file menu.

When you exit, R will ask you if you want to "Save the workspace image?" If you want R to remember everything that you did, remembering the variables that exist and their current values, then say "Yes", in which case R will save a file named .Rdata to the current working directory. (This is one reason that setting your working directory is important.) The next time you start R from that directory, everything that was in R's memory when you saved .Rdata, will be restored to memory. What this means will become more clear as you become familiar with the R console, creating variables, etc. If you want to start you next session from scratch, with nothing in memory, say "No." I often say "No", because I want the programs I write to stand alone.

However, as you work through this tutorial, it would be wise to say yes to allow your console work to be preserved from session to session. I recommend that you explicitly name your

workspace file something like `Rtutorial.Rdata`. You can save the workspace using the Save workspace as option of the Session dropdown. the If you do this, then you will need to explicitly open the workspace each time you start R using the Load Workspace command of the Session dropdown.

```
#This script depends on loading the workspace
#(give path name).
```

Alternatively, (better) you can simply put a line of code in your script, substituting the proper path and file name for `"full_path_to_file.Rdata"` to load the workspace into R's memory:

```
load("full_path_to_file.Rdata")
```

3.7 Getting help

R has been criticized for having poor documentation, which I think is undeserved for the most part. The documentation does take a little getting used to. R is an enormous and complex language and it is tough to organize a description of what it does and how to do it. That said, I have been able to figure out virtually anything I need to do, particularly with the help of web searches. Here are some ways to get help. I am putting this material up front in this document, but you probably won't be able to use it very much until you start getting into the material below. I suggest you read through this now to know what is here, but wait to use the material once you start doing things in R in the next section.

3.7.1 Introduction to R

If you click on Help on the RStudio tool bar, then Rhelp, then An introduction to R you obtain a very nice overview of the R programming language. There is some overlap between that manual and this primer, but the motivated among you will read the Introduction as well as what I have written. At the minimum, it will give you another view of important concepts, and learning from alternative viewpoints is the best way to master material. It is a very useful, informative document. There are lots of other resources under the Help pulldown and I urge you to explore them.

3.7.2 Getting help from the console

You can enter ? before any R statement or function name[8] on the console and access the documentation. You can also use `help(" ")` where the topic you want to learn about is enclosed in quotes. One of the most useful ways to get help is to enter `example( )` with an R statement within the quotes. For example, try: `example(plot)` You will need to hit

[8]Some statements, for example, `for()`, `while()`, `function()`, and `if()`, will not work with the `?` approach—they think you are actually writing code and respond with a + continuation rather then the Help documentation. If that happens use the `help(" ")` version. More about these statements soon.

enter to see the plots. The code generating each is written to the console. I use `example( )` and `?R statement` a lot. Remember that you can execute the example code from the console or the program window—this often helps to see what does.

Another very useful help function used from the console is `apropos()` The `apropos()` function brings up a list of all of the functions that are related to some general topic. You can then access each one using `?function_name`. Try this:

```
>apropos("file")
>apropos("plot")
```

If nothing else, this should give you a sense of the tremendous variety of functions available in R.

3.7.3 Searching list serves

Click on `searc.r-project.org` on the Help menu. This is a very useful site. Also see `http://tolstoy.newcastle.edu.au/R/`

and

`https://stat.ethz.ch/mailman/listinfo/r-help`

3.7.4 Third party books

As R gets more and more popular, many great third party texts are emerging. Two references that I like are

Crawley, M. J. 2007. The R Book. John Wiley and Sons, West Sussex, U.K.

Adler, J. 2010. R in a Nutshell. O'Reilly, Sebastopol, CA, USA.

I confess that although I once used texts like these a lot for reference material, I now find most of what I need with web searches.

A more in-depth compliment to this primer is found at

http://www.atmos.albany.edu/facstaff/timm/ATM315spring14/R/The%20Art%20of%20R%20Programming.pdf

People I respect rave about this free text. The table of contents looks excellent. I haven't used it.

3.7.5 Googling an R topic

There is a ton of material on the Web about R. Often you can find exactly what you need with a simple Google search—actually, quite often this is the best way to find the manual page that you need. I just did a Google search on "Debug and R" to help me write the section on debugging. It took me right to the page in the manual I needed. Moreover, the rapid rise of the popularity of R is motivating all sorts of tutorials and Power Points etc. on R. My favorites are

`http://www.statmethods.net/index.html`(particularly good for beginners)

and

`http://wiki.r-project.org/rwiki/doku.php`

and,

`http://addictedtor.free.fr/graphiques/`

There is also a R specific Google search tool:

`http://rseek.org/`

Before you proceed, take a look at all of these sites and explore them a bit.

So, to learn R, I urge you to work from multiple sources.

4 Basic operations in R

4.1 Assignment

We will write lots of equations in R. Equations consist of variables and numbers on each side on an assignment operator, which is subtly different from the = sign that you are familiar with. The commands for the population growth example, above, illustrate assignment of values to variables. For example, `r = .2` means "assign the value `.2` to the variable `r`. Note that R also allows `<-` for assignment. The statement `r <- .2` accomplishes exactly[9] the same thing as `r = .2`. The `<-` syntax that is required by JAGS, which we will use heavily in the course, and you will see it often in R code written by others, so it is good to know what it means. I use `=` rather than `<-` because I have found some intractable bugs when I used `<-` in function arguments (more about those later) which seem to require `=` to work properly. However, as long as you can remember that function arguments must take a `=` and not a `<-`, you might want to use the `<-` syntax to reduce the number of syntax errors you receive in JAGS.

Variables are called, well, variables, because the value they contain depends on what you assign to them. The value of 2 cannot change in R, but the value of named variables can. The names of variables can consist of letters (a-z and A-Z), letters combined with numbers, and letters and numbers combined with special characters. So, for example, all of the following are legitimate names for variables: `a, a1, alpha, carbon, x.coordinate, habitat_area`. Some special characters are not allowed (for example, any of the characters designating operations, below). R is case sensitive, so the variable `a` is different from the variable `A` and `survival` is not the same as `Survival`.

4.2 Operators

Arithmetic is accomplished by the usual operators `+, -, *, / and ^` (for raising to a power)1. In addition, there are the common mathematical functions `log( ), exp( ), sin( ), cos( ), tan( ), sqrt( )`. There are also many useful functions that operate on vectors and matrices, but we will learn about those later. The list of mathematical functions

[9]For the code wonks among you, there actually are some subtle differences in the operation of `=` and `<-`. If you want the detailed treatment, see
`http://csgillespie.wordpress.com/2010/11/16/assignment-operators-in-r-vs/.`

Table 1: Basic mathematical operations in R

Operation	Expression	R statement
addition	$a = b + c$	a = b + c
subtraction	$a = b – c$	a = b - c
division	$a = \frac{b}{c}$	a=b/c
power	$a = b^c$	a=b^c
square root	$a = \sqrt{b}$	a=sqrt(b)
root	$a = \sqrt[c]{b}$	a=b^(1/c)
exponential	$a = e^b$	a=exp(b)
natural log	$a = \ln b$	a=log(b)
log base 10	$a = \log_{10} b$	a = log10(b)

available in R is long—we will learn many more than the simple ones listed here—but these give you an idea of the basic mathematical operations.

4.3 Numbers

Equations can contain numbers as well as variables. So for example, the statement `N1 = 100*exp(r)` contains all three elements. However, equations can also consist of variables and numbers without arithmetic operations e.g., `N0 = 100`. (What does this statement do?)

4.4 Order of operations

The order in which calculations are carried out can influence their outcome. Consider `a / b + c`. In R, this statement would be executed as divide `a` by `b` then add `c`. Of course, this gives a very different results than dividing `a` by the sum `b + c`. In R, exponentiation is done first, followed by multiplication or division followed by addition or subtraction. When in doubt, use parentheses, e.g., `(a / b) + c` if you want to divide first (which is what R does without parentheses) or `a/(b+c)` if you want to do the addition first (which requires parentheses). When in doubt, use parentheses. To illustrate the importance of parentheses, enter the following at the console

```
1000*1.06^20/12
```

and

```
100*1.06*(20/12)
```

Exercise: The diameter of the earth. Write an equation in the console that assigns the diameter of the earth at the equator (in km) to the variable `earth.diameter`. The data you need include the circumference of the earth (24,901.55 miles at the equator) and the relationship between miles and kilometers (1 km = .621 miles). I am assuming you know the relationship between circumference and radius or diameter of a circle. You will probably need to do a little algebra first. As a useful hint, enter

```
> pi
```

After you have entered your equation, enter
`> earth.diameter`

If you have done this right, R will respond:
`[1] 12763.94`

Now do the same thing by writing a program (script) that calculates the diameter of the earth. Start by assigning numeric values to variables (using symbols of your choice). Write your equation in terms of those variables (i.e., your final equation should not contain any numbers) and execute it.

4.5 When does a variable exist?

Try the following. Enter:

```
> axe
```

R will respond with the message:

```
Error: object "axe" not found
```

because you have not told R about the object `axe`. It doesn't't know a exists, therefore it responds, sensibly, that it can't find it. Now try the following. Enter:

```
> axe = 2
```

R responds with

```
>
```

Now enter

```
> axe
```

to which R responds:

```
> 2
```

This is a very important thing to understand about R: objects don't exist until you assign a value to them or until explicitly declare them (declaration is described in the next section.) This turns out to be useful. It prevents errors that can result from misspellings etc. For example, recall your value for the diameter of the earth by entering

```
> earth.diameter
```

Now try the same thing, but make a small mistake, something like

```
> earth.diametter
```

How does R respond?

5 Data in R

Before you build your first model, you need to know about how R stores data. In this section you will learn about four simple structures for data: scalars, vectors, matrices, and arrays. Later, we will cover more advanced structures: lists and data frames.

There is potential for confusion here about what we mean by "data", particular for those who have some modeling background. Usually, we think of data as observations from experiments, samples, surveys, etc. R includes such observations as data, but has a somewhat broader definition. Data include values that are assigned to variables represented by its data types (i.e, scalars, vectors, and arrays). So, simulation results stored in an array, are, in R's view, data. They are not observations. I will probably jump back and forth between these views in the course—there is simply no way to be totally consistent without inventing a new word. I presume you are sufficiently clever to juggle these uses of the term.

5.1 Scalars

The variables we have been working with above are scalars. Scalars are variables that contain a single numeric value.

5.2 Strings

Strings are like scalars, except they contain character values. So, to assign a string to a variable, you will use syntax like `a = "Fred"`.

5.3 Vectors

5.3.1 Vector basics

Unlike scalars, vectors are objects that contain more than one value. They are analogous to a row of data in Excel. So, for example consider the following in Excel:

	A	B	C	D
1	234	17	42.5	64

To create and display an analogous vector (named `v1`) in R enter :

```
> v1 = c(234, 17, 42.5, 64)
> v1
```

R responds:

```
[1] 234.0 17.0 42.5 64.0
```

The `c( )` is a *concatenate* function .

In Excel you access the value of a cell using an indexing system based on letters (for columns) and numbers for rows. So, 234 is the value contained in cell A1. The indexing system in R is subtly different, but much more flexible as you will see shortly. Because vectors are like "rows" in R , to get the value of an element, you simply given the position of the element brackets [ ] next to the vector name, e.g.:

```
> v1[3]
```

will output the third element of the array `v1`.

It is also possible for vectors to include characters. For example:

```
study.areas = c("Maybell", "Poudre", "Gunnison")
```

Exercise: Vectors 1. What is the value of `v1[2]`? Create a vector called ages that contains the ages of everyone in your workgroup. Create a vector called names that contains the names of everyone in your workgroup. Output the value of the fourth element of `v1` to the console. Multiply the second element of `v1` by 2. Enter `v1[1:3]`in the console. What does the : operator do?

5.3.2 Mathematical operations on vectors

Vectors can be used in arithmetic expressions. Operations are performed element by element. Try a variety of arithmetic and mathematical operations on the vector v1. For example,

```
v2 = v1*2
v2
```

5.3.3 Logical operations on vectors

Here, I introduce logical operations in the context of vectors, but they have wide application in R–for example in use with `if()` statements, which will be covered later. Logical expressions differ from arithmetic expressions. Where arithmetic expressions evaluate to some number (the expression 2* 2 evaluates to 4), logical expressions evaluate to `TRUE` or `FALSE`. So if the value of `a` is 2 then `a < 4` evaluates to `TRUE` while `a > 4` evaluates to `FALSE`. R does logical as well as arithmetic evaluation. The symbols for frequently used logical operations are shown in Table 2.

Exercise: logical operations on vectors. Enter the following at the console `v1; v1 < 200; v1[v1 < 200]` Discuss with you lab mates what is going on here. Using logical statements like the one above, form a new vector from the from elements of `v1` containing elements that are a) greater than twenty b) less than 200 and greater than 20 c) not equal to 17 d) equal to 17 or equal to 42.5

Table 2: Logical operations in R

Symbol	Meaning
`<`	less than
`>`	greater than
`<=`	less than or equal to
`>=`	greater than or equal to
`==`	equal to
`! =`	not equal to
`!`	logical not
`&`	logical and
`|`	logical or

Table 3: Frequently used functions that operate on vectors in R. In the examples below, `z` is a vector.

Operation	Result
`length(z)`	number of elements in z
`max(z)`	maximum value of z
`min(z)`	minimum value of z
`sum(z)`	sum of values in z
`mean(z)`	arithmetic average of values in z
`median(z)`	median of z
`range(z)`	vector including minimum and maximum of z
`var(z)`	variance of z
`sd(z)`	standard deviation of z
`cor(z,y)`	correlation between vectors z and y
`sort(z)`	vector sorted
`rank(z)`	vector containing the ranks of elements of z

5.3.4 Vector functions

R has a rich set of functions that operate on vectors (Table 3).

Exercise: vector functions. Find the mean, max, minimum, range, and variance of v1. Find the average of the maximum and the minimum of v1. Compose another vector `v2` containing 4 numeric values of your choice. Now, execute the commands `min(v1,v2)` and `pmin(v1,v2)`. Discuss with your lab mates what the two commands are doing. The difference between `min()` and `pmin()` is really important because failing to understand it can cause mistakes that are very hard to diagnose. There is an analogous version of `max()` and `pmax()`.

5.3.5 Declaring vectors

In the examples above, R figured out how long a vector is and what kind of data it contains by the assignment you gave it, that is by what was on the right hand side of the assignment operator. It is also possible to create a vector of zeros, which can be very useful if you want to create a data structure that will be assigned values latter. Here are some ways to do that.

Most often, we will use something like

```
v = numeric(10)
```

where 10 is simply the number of elements in the vector, its length. (Remember, I am using v as a variable name here, it could be anything you want to name the vector.) This statement says "create a vector that can hold numbers.

It can be useful to find out the length of a vector, which is accomplished with the statement:

```
length(v)
```

It is also possible to create a vector of integers[10] using:

```
v = integer(10)
```

Exercise: Put the following code in a script and run it line by line. Explain what is going on to your lab mates.

```
d = numeric(5)
d
(d + 1) /3
length(d)
```

What do you do if you know you want to declare a vector, but you don't know how long it should be? You can let R know that it should recognize v as a vector and allow you to assign values to any element in the vector. For example,

```
v=NULL
v[8]=254
v[2]=10
```

The symbol NA is used to indicate missing data, more about this soon. Explain what is going on here. Now enter `v[12]=13.4` and explain the result.

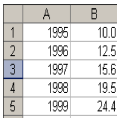
5.4 Matrices and arrays

Vectors are analogous to rows in a spreadsheet. Matrices are analogous to sheets; three dimensional arrays are analogous to workbooks. First we will work on matrices and then turn to arrays of higher dimension. The matrix will be a true workhorse in this course. Matrices are special types of two dimensional arrays; more about arrays later. They differ from arrays because certain mathematical operations, for example, transforms, are valid only for matrices.

[10] You can also create vectors of single precision real numbers using v = real(10) or vectors of double precision real numbers using v=double(10) but these distinctions will not be needed in this course.

5.5 Creating matrices

Consider the following bit of data contained in Excel:



	A	B
1	1995	10.0
2	1996	12.5
3	1997	15.6
4	1998	19.5
5	1999	24.4

In R, as in Excel, specifying the value in a "cell" now requires both rows and columns. In R, this is done by the order of the indices. The first index gives the row number, the second index gives the column. To remember this order, think RC[11] .

To create matrix analogous to the rows and columns in the sheet, begin by using syntax like this:

```
A = matrix(0,nrow=5, ncol=2)
```

This syntax says "Create an matrix named A with 5 rows and 2 columns containing the value 0 for all elements." (It doesn't't have to be named A, you could call it Bookshelves if you want to.)

Exercise: Creating a matrix Create and output a matrix named B that contains seven columns and 5 rows with all elements containing the value 7.

Working with rows and columns of a matrix R has exceptionally powerful ways to work with matrices using commas to designate entire rows and columns, effectively making them vectors. For example, the syntax:

```
A[,2]
```

specifies all of the elements in the second column of the matrix A, while

```
A[4,]
```

specifies all of the elements of the fourth row of the matrix A. Remember, rows and columns treated this way create vectors. You will use this capability a lot in this course, so it is truly critical that you understand it.

Exercise: matrices. Enter the data in the Excel example above into an matrix called M. Write and execute a script that

1. Sets up the matrix M and fills it with zeros using the `matrix()` statement.

[11] When I took my first programming course (FORTRAN in 1975 in the Engineering College at Vanderbilt) the instructor gave us a way to remember the indexing of rows and columns. "If you are religious, think RC, Roman Catholic. If you are not, then just think RC cola." Those of you from the South who are old enough will get the second reference. This might be *none* of you.

2. Enters the appropriate values in the matrix row by row Hint: to enter the first row use: `A[1,] = c(1995,10.0)`
3. Outputs the entire matrix. 4
4. Outputs each row
5. Outputs each column.
6. Finds the largest value in the matrix
7. Finds the average of the second column

5.6 Sub-setting matrices

Very often we want to find particular rows and columns of a matrix, for example, we want to find the rows that contain specified values of a column. R has extremely powerful syntax to accomplish these operations, but be warned, it requires some effort to understand. For example, lets say we want to obtain a subset of the array M containing the data for years earlier than 1998. The syntax for doing this is as follows:

```
M[M[,1] < 1998,]
```

to which R responds:

```
[,1] [,2]
[1,] 1995 10.0
[2,] 1996 12.2
[3,] 1997 15.6
```

Well, now that is not exactly intuitive. Let's take this statement apart so that you can understand how it works. First, enter

```
M[,1] < 1998
```

to which R responds

```
[1] TRUE TRUE TRUE FALSE FALSE
```

What is going on here? Well, you have a logical operator ($<$) and any time you use a logical operator, then R will return values of TRUE or FALSE. So, `M[,1]` specifies a vector consisting of all of the rows of M (because there is a "," in the row position of the index) in column 1. R returns a vector of TRUE and FALSE by going element by element through that vector and determining if the element is < 1998. If you don't understand this after thinking about this and discussing it with your lab mates, then call on one of the TA's or me. It is very important that you understand this.

Now let's add the outer "wrapper" that does the rest of the work we need done. I am intentionally adding some spaces to make this more clear:

```
M[ M[,1] < 1998 ,]
```

What is happening here? We have put the set of TRUE TRUE TRUE FALSE FALSE produced by

```
M[,1] < 1998
```

within `M[ ,]`. R now looks this over finds all of the rows of M that have a value of TRUE produced by our logical comparison, that is that have a value less than 1998 in column 1. OK we have found the rows, what does R return? R returns all of the columns of M because of the comma immediately before the trailing bracket is not followed by any specific column or columns. (Remember, M[r,] means row r and all columns.) If we simply wanted the values of column 2 returned for the rows where years $<$ 1998 then our syntax would be:

```
M[ M[,1] < 1998 ,2]
```

to which R would respond

```
[1] 10.0 12.2 15.6
```

So to wrap this up, consider the following diagram to gain some general understanding of subsetting matrices:
Now, the error you will make, I promise. If you enter

```
M[ M[,1] < 1998]
```

R will respond:

```
[1] 1995.0 1996.0 1997.0 10.0 12.5 15.6
```

Hmmm, what is going on here?
Some examples might reinforce the use of subsetting statements:

```
# Returns rows of M that contain 1995 or 1999 in column 1
M[M[,1] == 1995 | M[,1] == 1999,]
#Returns values in column 2 of M that are in a row containing 1995 or 1999 in colum
M[M[,1] == 1995 | M[,1] == 1999,2]
#Rows of M that contain values in column 2 that are #greater than the average value
M[M[,2] > mean(M[,2]),]
```

Exercise: matrices, continued.

1. Find the all of the values of column 2 of M that are less than 19.
2. Find and output the row that contains the minimum value contained in column 2.

3. Now, to illustrate a particularly devious error, enter the following at the console: `M, M[1], M[10], mean(M[1]), mean(M[,1])`. Discuss what is going on here with your lab mates. What problems that can arise if you leave out the comma when you intend to index a row or column? When you enter a single subscript (with no comma) for a two dimensional matrix, how is R treating the matrix? Hint: enter the commands `length(M)`, `length(M[1,])`, `length(M[,1]`at the console. This is particularly important to understand because it is a source of potential errors that are very hard to detect. This illustrates that flexible languages like R give you tremendous power—but with that power comes the responsibility to understand what you are doing. Don't go forward until you really understand this part of the exercise.

5.6.1 A common error when creating matrices

A common error is something like:

```
> Z =matrix(0, nrows = 5, ncol = 7)
```

in which case you will get the sort of cryptic error message:

```
Error in matrix(0, nrows = 5, ncol = 7) : unused argument(s) (nrows = 5)
```

This tells you, in R's way, that you put an s at the end of nrow, an s that shouldn't be there.

5.7 Arrays

In this course, we will use lots matrices, which are essentially 2 x 2 arrays. I say essentially because matrices have some special mathematical properties that are not shared with 2 x 2 arrays, but we needn't go into these in detail. So although matrices, lists, and data frames (more about these soon) are the data structures that we need in this course,, most of the time there will be occasions when we need "a stack of matrices", which, roughly speaking is a 3 dimensional array. A three dimensional array is analogous to a workbook—it is like a stack of sheets, each containing rows and columns. A four dimensional array is like a set of workbooks containing a stack of sheets containing rows and columns, and so on. So, a 2 x 5 x 3 dimensional array is stack of 2 dimensional arrays—it consists of 3, 2 x 5 sub-arrays. The syntax is a pretty simple. To set-up a 3 dimensional array name A3 consisting of 3 sub-arrays, each of which contains 2 rows and 5 columns each use:

```
> A3 = array (0, dim = c(2,5,3))
```

Again, the indexing works, rows, columns, arrays (RCA)—the first number in the dim statement gives the number of rows in the "subarray" (i.e, 2), the second number gives the number (i.e., 5) of columns in the sub array and the third number (i.e., 3) gives the number of sub arrays.

Exercise: Three dimensional arrays A couple of years ago, I worked on a problem that involved capture histories of mule deer that are potentially infected with Chronic Wasting Disease. These histories took the following form. The animal can be in 1 of 4 unique states: alive and susceptible, dead and susceptible, alive and infected and dead and infected. Our research team observed the animals state over a 5 years. Data for years are contained in columns and there is a 4 x 5 subarray for each of 100 animals. For a given year, the animal's state is indexed by rows in the column for that year (row 1=susceptible alive, row 2 = infected alive, row 3 = susceptible dead, row 4 = infected dead); there is a 1 in the row if the animal is in that state and a 0 otherwise. Please do the following.

1. Create a 3 dimensional array containing all 0's to hold these data and output the sub arrays for the first 5 animals.
2. Enter initial conditions for this data structure, assuming all animals are susceptible in year 1. 3
3. Presume that the 5th animal is susceptible and alive in years 1 and 2 becomes infected in year 3 and dies in year five. Create the appropriate capture history for animal # 5 by filling in its sub array with 1's and 0's.

5.7.1 Functions for arrays and matrices

It is important to understand that you can use bracket notation, (i.e., `M[1,]`) to create a vector from a matrix. In this case, all of the function that apply to a vector (Table3) will apply to the vector that you extract from a matrix. Moreover, these functions will also apply to the matrix itself, but with results that may not be immediately intuitive. For example, consider matrix M above. What do you get if you execute the command `mean(M)`? `length(M)`? `min(M)`?

There are times when you want means or sums of each for the rows or each of the columns of a matrix. Of course, you could do this one row or one column at a time using the bracket notation as shown above. However, there are functions that make this easier, `rowSums()`, `rowMeans()`, `nrow()`,`colSums()`, `colMeans()`, and `ncol()`. Consider the matrix Q:

$$Q = \begin{pmatrix} 8 & 81 & 12 \\ 6 & 2 & 34 \\ 13 & 12 & 1 \end{pmatrix} \tag{1}$$

Figure out how each of these matrix functions work by entering Q into R and using each function, e.g., `rowSums(Q)`.

I urge you to look into the `apply( )` function, which had many uses for summarizing matrices. The syntax is `apply(X,MARGIN, FUN)`where `X` is the name of a matrix or an array, `MARGIN` specifies the subscripts of the matrix or array that you wish to apply the function `FUN`, which is any R function, including functions that you write. So, for example

`apply(M,2,prod)`returns the product of the elements of the columns of the matrix `M`, `apply(J, 1, sort)`sorts the rows of the matrix `J`. The `reshape` package offers a much more general set of tools for summarizing data in matrices (and data frames) and is worth study.

5.7.2 Matrix algebra

R has a rich set of functions (for example, eigenvalues and eigenvectors) as well as operators that apply to matrices (for example, multiplying a matrix with a vector). These are beyond the scope of this tutorial, but you should know they exist. We will learn some of these later in the course. If you are curious now, take a look at the Introduction to R manual or the `popbio` package.

6 Programming in R

6.1 Iteration

6.1.1 The basics of for loops

One of the things we will do a lot in this class is to iterate an equation over time. As an example, consider the equation,

$$N_{t+1} = \lambda N_t \tag{2}$$

which is the basic, discrete time model of exponential population growth. On the left hand side, we have the value of a state variable, N, at a future time and on the right hand side, we have a function that calculates the future value of a state variable in terms of its current value and the parameter λ. This is known as an iteration equation.

R, and all other computer languages, have several methods for executing operations iteratively, saving you the tedium of doing the same thing many times. The method we will learn is called a for loop. It will be a genuine workhorse in this class, so you should pay close attention here. For loops are best explained by example. Consider the following:

Algorithm 1 Illustration of `for` loop

```
lambda = 1.2
N = 10
for (t in 1:10){

    N = lambda * N

} #end of N loop
N
t
```

Write this code as a script and execute it. What is going on? We start by assigning an initial value to the scalar `N`. This tells R that `N` exists and has the value of 10. The code then tells R to do the following things:

1. Starting with the value of t = 1,
2. Do the operations within the { }, in this case update the value of N by making it equal to the old value of N multiplied by the (constant) value of `lambda`.

3. Each time you do whatever is within the `{ }`, increase the value of `t` by 1
4. When the value of `t = 10`, stop and go to the next statement.
5. Output the values of `N` and `t` to the console.

Well, that is interesting, but not terribly useful. The loop allows us to find out the value of `N` after 10 intervals of time. But what we really want for most models is the value of `N` at each time. It does illustrate how the right hand side of an expression for a variable replaces the value on the left hand side, but you probably knew that anyway. For this to be really useful, we need vectors and arrays. But before we go on those topics, notice two programming conventions. The code within the `{}` is indented to make it easy to see it is part of a loop. Also, there is a comment at the end to the loop to identify it.

To make this construct more useful, study the code in Algorithm 2.

Algorithm 2 `for` loop including storage of state variable in a vector.

```
for
#value of growth rate
lambda = 1.2
#Vector for holding state variable
N = numeric(10)
#Initial value of N
N[1] = 10
#Loop to calculate N over time. Use t to index vector.
for (t in 2:10){

    # store the values of N in a vector
    N[t] = lambda * N[t-1]

}
plot(N)
```

Ok what is going on here? This is pretty important, so let's look at it in detail:

1. `lambda = 1.2` lambda is a parameter in our model. This model has only one parameter, other models will have many, but the general idea is exactly the same–the value of a parameter does not change with time . We assign the variable lambda the value 1.2.
2. `N = numeric(10)` Now we create a vector to hold the values of N in sequence.
3. `N[1] = 10` If you think about equation 2, above, we can't calculate the left hand side without knowing an initial value for the right hand side. Given that starting value, each subsequent value can be calculated because the new value is based on the old one. So, we must give an initial value for N, that is, the first element in the N vector.

4. `for (t in 2:10){N[t] = lambda * N[t-1] }` This is where the real work is done. The for loop starts with a value of t = 2 . Why not start at 1? We start at 2 because we have specified the value of `N` at `t = 1`,so the first time through the loop, we calculate the value of `N[2] = 1.2 * N[1]`. The second time through the loop, we calculate `N[3] = 1.2 * N[2]`, the third time through the loop, `N[4] = 1.2 * N[3]` and so on until `t = 10`.

5. `plot(N)`We now use a plot statement to display the results of the model. Usually, the plot statement has the syntax `plot(x,y)`where x is a vector of values for the x axis and y is a vector of values for the y axis. Again, these vectors can be named anything; x and y are simply placeholders in this example. If plot has only one argument, as in our example above, then the plot statement assumes this is a time-series creates a sequence of x values of the appropriate length starting at 1 to match with the y values we gave it. So, in this case the plot statement determines the length of N and makes an x-axis for you with values 1 – 10.

Exercise: A Gompertz model of plant growth. Plant growth can be modeled as the accumulation of biomass over time. The Gompertz equation is often used to represent growth of individual plants and plant communities. For an individual plant we have:

$$B_{t+1} = B_t + \mu_0 B_t \left[1 - \frac{k}{\mu_0} \ln\left(\frac{B_t}{B_0}\right)\right] \Delta t \tag{3}$$

where B_t is the biomass of the plant at time t, B_0 is the initial plant mass, μ is the mass-specific maximum rate of plant growth (which occurs when plants are very small), and is the exponential rate of decline in plant growth rate that occurs with time as plants age. You may assume that $\Delta t = 1$.

1. Use equation 2 to develop a simulation model of plant growth. Assume that $\mu = 1, k = .3$, and $B_0 = 10$. Run a simulation for 30 time steps and plot your results. Hints: As you no doubt guessed, a useful template for this problem is given by the code for the discrete time exponential model in algorithm 2. The only real difference is that the update equation here (equation 3) is a bit more complicated that the one in the example (equation 2) So, be especially careful to properly translate equation 3 into R code. This is where you are likely to make mistakes. When you get it running, save your code; you will need it for part 3 of the exercise.

2. Now add two new variables to your model, R_t, the absolute growth rate (g/time) and r_t, the relative growth rate (g /g/time)[12]:

$$R_t = \mu_0 B_t \left[1 - \frac{k}{\mu_0} \ln\left(\frac{B_t}{B_0}\right)\right] \tag{4}$$

$$r_t = \frac{R_t}{B_t} \tag{5}$$

[12]You should understand that $R_t \cong \frac{dB}{dt}$.

Write code to update these variables and plot each of them along with your plot of biomass over time. So, you will now have 3 vectors to hold state variables (`B, R, and r`) and you will use 3 plot statements to output them. To see your plots, you will need a special line of code for putting several plots in a group. Put this statement ahead of your plot statement: `par(mfrow=c(2,2))`. This sets up a window to hold 2 rows and 2 columns of plots.

3. In vectors `R` and `r` you will not have an initial value other than 0 at time 1 which gives you an odd plot. Figure out a way to plot the values 2 through 10, omitting the value at t = 1.

4. This next exercise is challenging. It requires understanding of matrices and looping and putting them together. If you can get this right, you are making terrific progress. Using the code you wrote for 1, eliminate the vector for B_t and instead create a matrix to store the value of time in column one and the value of B_t in column two. Plot the array using the syntax `plot(x,y)` where `x` is column 1 and `y` is column 2 of your array.

Ok, these three exercises cover a lot of important material. They contain the essential elements of dynamic modeling which we will emphasize in the later part of the course. These are truly workhorse concepts. Do not go forward with this tutorial until you get totally familiar with the answers to these exercises, which are given in Appendix A. I am betting that the area you may get confused in the difference between the index of an array and the value contained in the position of the array determined by the index in exercise 4, above. Ask for help from me or one of our brilliant TA's or both until you completely understand what is going on here. We will work with you until you get it.

6.1.2 For loops controlled by sequence vectors

The behavior of `for` loops is determined by a sequence. Above, I made things simple by using 1:10 to describe a sequence of 10 integers starting at 1 and ending at 10. However, `for` loops can be controlled by vectors that define a sequence. This requires that we understand the `seq()` function. Enter the following code in the console or from a script window:

```
s1 = seq(from = 0, to = .10, by = .02) s1
#A shorthand version of the same statement is
s1 = seq(0,.10,.02)
```

This illustrates how you can form sequences of numbers over any range (with endpoints defined with `from` and `to`) with any increment (defined with `by`). The second version is shorthand for the first—if you can remember the order, then you can dispense with the `from =, to =, and by =` . If you use these, it is easy to forget the =, which will produce an error. Sequences can be useful for many things, but their primary use in this course will be to control for loops. So, the statement:

```
for (j in s1){
}
```

will iterate over the values contained in `s1`, giving values in sequence to `j`

6.1.3 Nested loops

Sometimes we want to iterate over two indices rather than one. This can be accomplished using nested for loops. For example, write this script in the program window and execute it:

```
a = matrix(0,nrow=5, ncol=5)
a
for (i in 1:5){

    for(j in 1:5){
    a[j,i] = j*i
    } #end of j for

} #end of i for
a
```

Nested loops work this way. The outer loop goes slowly, the inner loop, quickly. So in the above example, in the first trip through the loop `i = 1` then `j = 1,2,3,4,5`. Then `i = 2` and `j = 1,2,3,4,5` and so on. In this example the results of a calculation are stored in the array a where rows are indexed by `j` and columns are indexed by `i`.

Exercise: a model experiment using nested for loops We will often want to look at how changing a parameter value alters the trajectory of a model's predictions over time. This means that we want to iterate over different parameter values for a model, and for each value, store the vector of values of the state variable over time. This exercise will be your first experience doing that. Building on the discrete time model of exponential population growth developed above, run an experiment where you vary the values of lambda from 1 to 1.6 in steps of .1. Display the results together as a family of curves on a single graph. You are going to need some hints to do this. Here they are. This will seem hard at first, but after you have done this once, it will be much easier the next time.

1. Make a vector (using the `seq()` function) of values of lambda from 1 to 1.6 in steps of .1.

2. Make a matrix with 10 rows and a number of columns equal to the length of your lambda vector. Initialize all of columns in row 1 of your array to hold the value 10. (Remember how you can do this in one, simple step using a comma to index your matrix). Call your matrix `N`.

3. Create nested loops. The outer one controls the value of lambda and the column where you want the results of the simulation for each value of t. The inner loops controls t. Within both loops is the calculation of the population size based on the value of lambda indexed by the outer loop and time indexed by the inner loop. Store the result in your 2 dimensional array. Each column should hold the values of N calculated at the different time steps.

4. At the end of your simulation, plot the results using `matplot(N, type = 'b')` where `N` is the name you gave your matrix and `type = 'b'` says you want results plotted with lines and symbols. More about this command later.

6.2 Functions

6.2.1 Why use them?

In mathematics a function maps one value of a number to another value using mathematical operations. Functions in R do the same thing—code specifies how to map one value to another. You have been introduced to several functions in R, particularly functions that work on vectors and matrices. However, although R is very rich in the functions it offers, there will inevitably be things that you want to do that are not included in the set of R functions. So, you could simply write code within your programs to do these things. However, there are three strong reasons to write functions.

First, there are things that you will do over and over in many different programs and it is handy to write the code once and use it over and over. Second, code that is written in functions allows you to use the R debugger, which we will learn more about in a moment. Finally, the most important reason is this. If you break your code into pieces, where each piece does a specific job, then your code will be easier to write, understand, debug, and maintain. It allows to break a problem into parts, concentrate on each one, then put them all together. It is much harder to do something with the values of variables that you don't intend to do–which is to say the functions "protect" the data. This is the main reason that I write functions.

6.2.2 Function basics, example 1

Think about a mathematical function, say $y = f(x)$. The symbol x is the argument to the function; y is the result, and f specifies the operations that map x to y. Here is the way that R implements this in code:

```
function_name = function(arguments){
some code doing things that results in a value for variable based on the arguments
return(variable)
}
```

There are many nuances to functions and I think the best way to learn them is to see some examples. Let's start with a very simple one that calculates square roots (I know, I know this already exists, but it is a good example. Note that I am giving another name because of this.)

```
squarert = function (x){

    y = x^(1/2)
    return(y)
    }

z = squarert(9)
z
```

So what is going on here? We assign a calculation or set of calculations to a name (remember, any name). The function name takes an argument, that is the variable within the ( ). Based on the value of the argument, we execute the calculation and "return" the result. This is a simple example, but it has all of the key bits: naming a function by assigning it, taking an argument, doing a calculation based on the argument (or arguments) and returning the result of the calculation. Now, instead of doing the calculation, we can simply use the function name with an argument, e.g.

```
z = squarert(9)
```

Note that we can accomplish the same thing using much more compact code

```
squarert = function (x) x^(1/2)
```

meaning if you have a single line of calculations, you can omit the `{ }` and the `return( )` statement. I like the more detailed one because I think it is easier to read and understand and because you must do it this way for more complicated functions. However, you should be aware of the shorthand version because you will see it and because you might like it better.

Often we have a list of numbers and we want to normalize it such that each value is rescaled to take on a value between 0 and 1 and the sum of the elements = 1. Given a vector v containing n elements, we obtain the normalized vector, η, using:

$$\eta = \frac{v_i}{\sum_{i=1}^{n} v_i} \tag{6}$$

Exercise: normalizing a vector. Write a function that takes a vector of numbers and returns a vector containing its normalized elements. Hint: take advantage of vector arithmetic and the `sum( )` function. Do not use a for loop.

6.2.3 Function basics, example 2

Let's look at another example, more sophisticated this time. You have no doubt learned about Akaike's Information Criterion if you have done any reading in the primary ecological literature. We actually won't use it much in this course, but it nonetheless offers a useful exercise for writing function. In the following function, we calculate AIC from likelihoods and counts of model parameters. R has existing functions that do this, so we are reinventing the wheel purely as a learning exercise. The formula for AIC for large samples is

$$AIC = -2\ln\left(L_{mle}\right) + 2K \tag{7}$$

where L_{mle} is the model likelihood evaluated at the maximum likelihood estimates of the model parameter and K is the number of parameters estimated from the data. When we have small samples, we use AICc , that is, AIC corrected for small samples:

$$AIC = -2\ln\left(L_{mle}\right) + 2K\left(\frac{n}{n-K-1}\right) \tag{8}$$

Algorithm 3 Example code for a function using an `if()` statement to control execution

```
#AIC example
    get.AIC = function(likelihood, K, n, small, log){
        if (log == TRUE){
            if (small == FALSE) AIC = -2*(likelihood) + 2*K
            if (small == TRUE) AIC = -2*(likelihood) + 2*K * (n /(n-K-1))
            }
    if (log == FALSE){
    if (small == FALSE) AIC = -2*log(likelihood) + 2*K
    if (small == TRUE) AIC = -2*log(likelihood) + 2*K * n /(n-K-1)
    }
    return(AIC) } #end of function
    a = get.AIC(likelihood = c(-30.61), n = 20 , K = c(7) ,small=TRUE, log=TRUE)
    a
```

So, we are going to write a function that calculates AIC or AICc given input on model likelihoods, the number of model parameters, the sample size, and a "switch" that tells AIC to do large or small sample AIC. Consider 3.

Type this up as a script and run it.

Wow, there is a lot to cover in these few lines. It is important that you think about what is going on, emphasizing how you might generalize this specific example to other functions. I want to use this code to illustrate five concepts: 1) named arguments and default values, 2) invoking functions, 3) logical variables, 4) use of `if statements`, and 5) scalar vs. vector arguments.

First look at the way the function is made:

```
get.AIC = function(likelihood, K, n, small, log){
    stuff
} #end get.AIC function
```

We are creating the function by assigning some instructions to the function name AIC. Within the ( ) are *arguments*, the values that the function needs to do its calculations. The function body, the guts of what it does (i.e. stuff), is contained between the {}. Understand that the left hand side of the = could be any R name (z, x2, alpha, calculate.aic, my.AIC.function, bookshelf, etc). Similarly, the arguments could be given any names. I chose the names for arguments to make the function easy to understand, but they could have been any names.

Two ways to invoke functions First, how do you use this function? There are two ways to invoke it

```
a = get.AIC(-30.61, 20 , 7 , TRUE, TRUE)
```

or

```
a = get.AIC(likelihood = -30.61, K = 7, n = 20 ,small=TRUE, log=TRUE)
```

both achieving exactly the same result. In both cases, we assign to the variable a the AIC value calculated from log likelihood -30.61, n = 20, K = 7 and our decision that the sample size was small. In one case, the arguments we give must be in exactly the same order as the order of the arguments when we created the function. This is pretty error prone—if you forget the order, you could easily get erroneous results. The second form is far safer—you don't need to remember the order, only the names of the arguments. And if you misspell an argument name, R will give you an error message. Note in the examples above, I deliberately switched the order of `n` and `K` to demonstrate that you get the same result in the second version (with the order switched) as in the first (that must have the same order as the function assignment.

Logical arguments: This sounds like something they train you to make in law school, but logical arguments are different in functions than in law. In our example the variable small is a logical argument, also called a "switch" or a "flag." It can take on the logical values TRUE or FALSE (you must use uppercase– true or false won't work). There are two logical arguments in the function, log and small. These control how we do the calculations, as you will find out in a minute. The log switch is used to tell the function whether you want AIC calculated from likelihoods or log likelihoods (you may have either one depending on how you estimated the model parameters). If you have likelihoods, then you should set the value of the switch to be FALSE. If you have log likelihoods, then you set the value of the switch to be TRUE.

The second switch is the logical argument small. If our sample size is small (more about that later in the course) we make the value of small = TRUE. If our sample size is large, we make the value of small = FALSE. These values control which form of the calculation we make using if statements. Using if statements to control calculations in a function The logical arguments control the calculations in the following statements, taken as a fragment from the full code above:

```
if (log == TRUE){

    if (small == FALSE) AIC = -2*(likelihood) + 2*K
    if (small == TRUE) AIC = -2*(likelihood) + 2*K * (n /(n-K-1))
    }

if (log == FALSE){

    if (small == FALSE) AIC = -2*log(likelihood) + 2*K
    if (small == TRUE) AIC = -2*log(likelihood) + 2*K * n /(n-K-1)
    }
```

R executes this line by line, starting with the the first `if` statement. If it is true, it executes the next line. If it is false, it jumps the line past the closing } for the first `if`. So, the first thing this function does is to decide whether to do the calculation based on log likelihoods or likelihoods. The next level of if statements determine whether the function is done using the large or small sample formula. If small = TRUE, the small sample formula is used, if small = FALSE, then the large sample formula is used.

This seems straightforward enough, but there is an opportunity for error here, and I almost promise that you will make it. Note that the logical if statement uses = = not =. If you use =, R will generate an error message, something like

```
Error: unexpected '=' in "if(
```

Vector vs scalar arguments: This next bit is really cool. Those of you who program in lower level languages (C or VBA for example) are going to love this. R doesn't care whether the argument you give it for the likelihood and K are scalars or vectors. If they are scalars, then it returns a scalar. If they are vectors, then it does the calculations using vector arithmetic and returns a vector! (Pardon me for getting excited about this.) So, if you want to compare 10 models, all with different likelihoods and different numbers of parameters, all you need to do is to create 2 vectors (or a 2 column matrix) and give vector arguments (or the columns of the matrix using the [,] trick) to the function. The likelihoods and the K for a given model must be in the same position in the two vectors (i.e. must have the same index) , but as long as you get them lined up properly, the function will return a vector of the 10 AIC or AICc values. So, try giving the get.AIC function two vectors with 3 likelihoods (their value can be any decimal, positive or negative) and 3 values for K. Observe the result.

Exercise Moment matching. I imagine you have some familiarity with probability distributions, at least with the normal distribution. Probability distributions are represented mathematically by functions[13] $f()$, that return the probability density of an observation, y_i, given (strictly speaking, conditional on) a set of *parameters,* $\boldsymbol{\theta}$, i.e.,

$$[y_i| \mid \boldsymbol{\theta}], \tag{9}$$

which reads "the probability distribution of y_i conditional on the vector of parameters $\boldsymbol{\theta}$. So, for the normal distribution the parameters are the mean and the standard deviation ($\boldsymbol{\theta} = (\mu, \sigma)'$) [14]and the density function is

$$[y_i| \mid \boldsymbol{\theta}]=\frac{1}{\sigma\sqrt{2\pi}}e^{\frac{-(y_i-\mu)^2}{2\sigma^2}} \tag{10}$$

The normal distribution is unique among probability density functions in that the moments of the distribution, i.e., the mean and the variance, are very easily derived

[13]To simplify life, I will focus on continuous data in this example, but when data are discrete we use discrete density functions. These are differ in a somewhat subtle way in the values they return, but we will go into this difference in class. You will become deeply familiar with both types of functions in this course; the example here offers a brief start to what you will learn in depth as the course proceeds. More soon.

[14]Note that $(\mu, \sigma)'$ is mathematical notation for a vector containing elements μ and σ.

from the parameters, the mean and the standard deviation. This is not true for all of the other distributions that we will use. For example, the beta distribution is used to describe data that can take on values between 0 and 1. For our purposes here, the formula for the beta distribution is not important, what is important is that the relationship between the parameters (α, β) and the moments (μ, σ^2) is

$$\mu = \frac{\alpha}{\alpha + \beta} \tag{11}$$

$$\sigma^2 = \frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}. \tag{12}$$

Solving two equations in two unknowns, we can find the parameters in terms of the moments:

$$\alpha = \frac{(\mu^2 - \mu^3 - \mu\sigma^2)}{\sigma^2} \tag{13}$$

$$\beta = \frac{\mu - 2\mu^2 + \mu^3 - \sigma^2 + \mu\sigma^2}{\sigma^2}. \tag{14}$$

This technique, where we solve for shape parameters knowing the moments or solve for moments knowing shape parameters, is called *moment matching.* We will use this heavily in the course.

1. Write a function that does moment matching for the beta distribution. You function should have 2 arguments, one for mean and one for the standard deviation and it should return a vector of shape parameters.

2. Modify your function so that it has a logical argument to control whether it calculates moments from shape parameters or shape parameters from moments. Set up if statements in the function to use equations 11 and 12 or 13 and 14 as controlled by the logical argument. Hint-be very careful about the order of operations in you code.

3. Use the following code to plot the beta distribution with mean = .7 and standard deviation = .3. Think about what is going on here (more about plotting soon).

```
x=seq(0,1,.01)
#beta density function.
#Get a[1] and a[2] from moment matching function
y=dbeta(x,a[1],a[2])
plot(x,y,typ="l", xlab="y", ylab="P(y)")
```

4. How could you test your functions? Hint—the R function `rbeta(x,shape1,shape2)` generates a vector of `x` random numbers from a beta distribution with shape parameters, shape1 and shape2. How could you use these to test if your function is behaving as it should?

6.2.4 Scope of Variables

This is a sort of challenging topic, so I advise you to tackle it when you are fresh and have sipped an espresso. You really must give this section your best concentration—it isn't't intuitive. However, if you master this topic for R, it will serve you well with virtually all programming languages. It is a critical concept in all of them. Ask questions of me and the magnificent lab assistants until you are solid with this topic.

To understand the concept of scope, you need to understand how an R program works—it executes one line at a time, starting at the top and going to the bottom. If you define a function, then R puts it in memory, just like any other object you define. When you invoke the function, then R goes to the code for that function in memory; in essence, when you use a function, R "jumps" to the code that defines it. So at any one time during the execution of your code, there is a "place" where R "is"—think of this place as a pointer[15] that moves through the code line by line. The location of this pointer determines whether a variable is "in scope" or "out of scope".

R knows about all of the variables that are assigned outside of a function, even when the pointer is within a function. However, variables that are assigned within a function are not available outside of the function. Execute the following code, trivial as it is:

```
rm(a,b)
a=7
fx=function(x){
y = a

    print(y)
    b = 17
    print(b)
    }

fx(9) # invoke function fx; values of a and b are printed
print(b) # Once you are out of the function, what happens to b?
```

Note that the value of `a` is known within the function `fx()` because it is assigned outside of any function. It is "in scope" anywhere. However the value of `b` is remembered by R only when the function `fx` is executed. There are two reasons why modern programming languages are structured this way. The first is that you can have functions that are created to serve many different scripts, regardless of the variables that are in the scripts–the internal workings of the function and the values of its variables are kept separate from the script that calls the function. This means, for example, that you could reuse the code you wrote for moment matching over and over without worrying about whether it conflicts with the particular script where you use it. Second is the idea of "data integrity." If you break your program into a series of functions, it makes it much more difficult to inadvertently change the value of a variable, which means that you are less likely to create logical errors (i.e.

[15] But please don't confuse this illustrative use of the term pointer with pointers in C, which are entirely different things.

"bugs"[16]) in your program). However, you inevitably will make these errors, which leads to the next topic, debugging.

6.3 Debugging

The toughest errors to detect in computer programs are errors in your logic- you think you have told the computer to do one thing, but in fact you have given it instructions that cause it to do something else. Discovering these errors is called debugging. This usually involves executing your code line by line, looking at results as you go, until you see your error in logic or coding. It is possible to do lots of debugging from the console—you simply execute your code line by line in the program window using cntrl-r and observe the results in the console. You can use the console to check the value of all of the variables that are in scope.

However, this can be difficult to do within for loops or within functions. In these cases, R has built in debugging tools. The debugging tool that I use almost exclusively is the `browser()` function. You place the statement `browser()` anywhere in your code and it causes program execution to stop at that place, bringing up a prompt `browse >` in the console. You may now move through the subsequent code step by step by simply entering `n` (for next) at the cursor of the console. You can exit the function or current iteration of the loop by entering `c` at the prompt. You can exit the browser altogether by pressing `Q` at the prompt. As with most things in this tutorial, this is more easily understood by doing than by reading.

Exercise: Using the debugger to understand variable scope: Enter the following code and run it from the program window.

```
#Scope example
#clear all variables
remove(a,b,c1,d)
f1 = function ( )
{ browser( )
c1 = 5
return(c1) }
f2 = function ( ){
browser( )
a = 100
d = 50
return (d)
}
main = function( ){

    browser ( )
    a = 3
    b = 4
```

[16]The origin of the term "bug" used in programming refers to the earliest computers that ran on vacuum tubes. Because these were warm, they attracted flying insects. Occasionally, these "bugs" caused the computer to fail. Debugging was the process of cleaning out the dead bugs.

```
f1 ( )
f2 ( ) }
```

Now execute the function main by entering main ( ) at the console. (It might be useful to see what happens when, by mistake, you leave off the (), something I do often). It will be useful to refer to the code printed here or in your program window. Step through the program by entering n at the prompt. At each step, examine the values of the variables a, b, c1, and d by entering them at the console. In your lab group, discuss their values relative to the scope of the variables.

6.4 Returning more than one thing

R requires does not allow you to return multiple variables, which means that code like"

```
a = 10
b = 13
return(a,b)
```

will cause an error. However, you can put several things in a list, and return the list. So for example, the following returns are legal:

```
a = 10
b = 13
return(list(a=a, b=b))
```

is ok, as is:

```
a=10
b=13
p=list
return(p)
```

To explore how this works, enter the following and execute it.

```
z=function(){
a=10
b=13
return(list(a=a,b=b))
}
m=z()
m
m$a
m$b
```

Lists will be covered in greater detail in section7.1.

6.5 Using functions for structured programming

For many years, *the* language for professional programmers was C++. I imagine it still is. In C++, or C for that matter, nothing can be accomplished outside of a function. This is because C++ and C were built to encourage *structured programming,* a style of work essential for doing reliable work in R when problems are at all complicated. Take a look at the following code that simulates data, runs a non-linear least-square fit on the data, then plots the generating model and the fit model against the data:

```
rm(list=ls())
alpha=38
gamma=1.7
c=5
sigma=2
set.seed(4)
par(mfrow=c(1,1))
L=sort(runif(50,min=12, max = 100))
mu = alpha*(L - c) / (alpha/gamma + (L-c))
plot(L,mu, typ="l")
y=rgamma(length(mu), mu^2/sigma^2, mu/sigma^2)
plot(L,y)
lines(L,mu)
model=nls(y ~ alpha*(L - c) / (alpha/gamma + (L-c)), start=list(alpha=50,gamma=4,c=
s=summary(model)
p=coef(model)
alpha=p[1]
gamma=p[2]
c=p[3]
y.hat=alpha*(L - c) / (alpha/gamma + (L-c))
lines(L,y.hat,col="red")
legend(40,18, c("generating", "nls fit"), lty=c("solid", "solid"), col=c("black", "
```

Get the code for this one from the Git site (`unstructured code example.R`, remind me if it is not there!) Now go to the console at do `ls()` to see what variables have been created. All of these variables are available for modification by any R command–they have global scope (remember global scope?), which is terribly error prone. Moreover, this code is really hard to follow, it is a single stream of commands that are unstructured by any logical framework created by the tasks that need to be accomplished. Now consider the structured alternative that does exactly the same thing.

In contrast, take a look at a structured version of the same code:

```
rm(list=ls())
#A function for the light limitation of trees
#============
g=function(alpha, gamma,c, L){
 mu =alpha*(L - c) / (alpha/gamma + (L-c))
return(mu)
}
#=============
#get_data() simulates data for a Michaelis-Menten model
#It uses the function g() for the model
#=============
get_data=function(alpha, gamma,c, sigma){
 set.seed(4)
 par(mfrow=c(1,1))
 L=sort(runif(50,min=12, max = 100))
 mu = g(alpha=alpha, gamma=gamma, c=c, L=L)
 y=rgamma(length(mu), mu^2/sigma^2, mu/sigma^2)
 gen.y= mu
return(list(y=y, x=L,gen.y=gen.y))
}
#=============
#Function to do non-linear least squared fit
#x are the independent variables
#y are the dependent variables
#Guesses is a named list of initial values for the parameters to be estimated.
#The order of parameters must be the same as the arguments to g( )
#================
fit = function(x,y, guesses){
 model=nls(y ~ g(alpha=alpha,gamma=gamma,c=c, L=x), start=guesses)
 s=summary(model)
 p=coef(model)
 y.hat=g(alpha=p[1],gamma=p[2],c=p[3], L=x)
 return(y.hat)
}
#================
###Function for plotting generating data and nls() fit
#data is the x and y data and gen.y is the generating line
#================
 plot_fit = function(data,fit){
 plot(data$x, data$y, xlab="Light level (lumens)", ylab="Tree growth (cm / yr)")
 lines(data$x, data$gen.y)
 lines(data$x,fit,col="red")
 legend(40,18, c("generating", "nls fit"),   lty=c("solid", "solid"), col=c("black"
```

```
}
#=================
## Maestro function
#==================
run_functions=function(){

    tree.data=get_data(alpha=38, gamma=1.7, c=5, sigma=2)
    nls.fit=fit(x=tree.data$x, y=tree.data$y, guesses=list(alpha=50,gamma=4, c=2))
    plot_fit(data=tree.data, fit=nls.fit)

}
#===============
run_functions()
```

Notice how the basic tasks, getting the data, fitting the model, and plotting the output are done by separate functions. The model is found in its own function (`g()`) which is used several times. Get the code from the web site (`structured code example.`R) and run it. Now execute `ls()` at the console. Notice that the only things that R returns are *functions*–they have global scope, which is good because we need to use them in various places in our code. However, notice that *none* of the data have global scope This means that "data" are protected–they cannot be accessed globally, which means that you can't inadvertently modify something that you didn't really intend to change without doing it editing a function body, which you are not likely to do carelessly.

Structured programming is harder than the simple way where we make everything global, and for small bits of work, the global way is fine. But practicing structured programming well develop habits that will serve you well when you get to tough, big problems. I will actively encourage you to write your code this way, with the confession that some my older code is written in terrible global style. I am forcing myself these days to do better!

One more idea. When I work this way, I often put the function definitions in their own file, in the working directory (remember working directories?) something like "Trees_functions.R". Then, my main program would have only three lines:

```
rm(list=ls() #start with clean workspace
source(''Trees_functions.R'') # Get the functions
run_functions()
```

The thing that is nice about this is that you can have your functions open in one window and your calling code in another. If you modify a function, all you need to do is save the file and rerun the code in the main window.

7 More about data

7.1 Lists

Lists are a special, *very* handy data structure in R. Strictly speaking, a list is an R object consisting of an ordered set of other R objects. These objects are known as the *components*

of a list. This sounds much more formidable than it really is. The simple way to put it is that lists are like vectors consisting of any combination of the R objects we have learned about so far: scalars, strings, vectors, matrices, arrays, logical values, and functions. Vectors must contain numeric values or strings but never their combination; lists can contain R objects in any combination. We will often use lists to return collections of values from functions, as you will discover shortly. Moreover, lists are critical to executing JAGS from R, which we will do *a lot* during the course. So it is important that you understand them.

To begin understanding lists, write the following code and execute it:

```
#List example
name = "Poudre"
n = 100
a = c(.30, .20, .50)
#create a list
population = list(location = name, size = n, composition = a)
population
```

To which, R responds:

```
$location
[1] "Poudre"
$size
[1] 100
$composition
[1] 0.3 0.2 0.5
```

Note that the same thing could be accomplished using:

```
population = list(location = "Poudre", size = 100, composition = c(.30,.20,.50))
```

This illustrates how you create a list and output it. Notice the relationship between the names of components (location, size, composition) the variables assigned to those names (name, size, a) and the values of the variables (Poudre, 100, c(.30, .20, .50). Understanding these relationships is the essence of using lists. The names in the list allow us to access specific components using the `$` operator. So, for example, enter and execute

```
population$size
```

Thus, by combining the name of the list (population) with the name of the component (size) using $ we can extract the component from the list. It can be assigned to another variable if that is useful, e.g.,

```
psize = population$size
```

Alternatively, we could use double brackets, i.e.

```
population[[1]]
```

would return the first element of the list.
What about components of lists that themselves have multiple, ordered elements, for example, the composition component of the population list has 3 elements, 0.3 0.2 0.5. How do we access the individual elements of a component of a list? Enter the following at the console:

```
population$composition[1]
```

So, it is possible to extract the elements of a component of a list (think of them as "subcomponents") in the same way that we access the elements of a vector, using [ ].

Exercise: Lists 1

1. Extract the second and third element from the composition component of the population list.
2. Extract all elements of the population list that are $>$ .2 using a logical operation. (Review the section on vectors if you are stumped here.
3. Add a matrix containing the elements below to the population list. Name it `transition`.

$$\text{transition}= \begin{pmatrix} 0 & 1.1 & 1.7 \\ .6 & 0 & 0 \\ 0 & .9 & .9 \end{pmatrix} \tag{15}$$

4. Extract the value in the first row, third column of the `transition` component of the population list. The second element of the third row.
5. The double bracket operator [[]] offers an alternative to `the $` operator as a way to obtain the elements of a list. So to obtain the *ith* component of a list you use `listname[[i]]`. Execute the following statements at the console and discuss what is going on with your lab mates.

```
population[1]
population[[1]]
population[1]*2 #Why does this produce an error
population[[1]]*2 #while this does not produce an error?
population[[3]]
population[[3]][1]
```

7.2 Getting Data into R

In all of the examples above we put data into matrices, arrays, vectors, etc. "by hand" using assignment operators. This is fine for small sets of data, but is tedious and error prone if we have lots of data. There are many ways to import data into R, but the most useful one, in my view, uses comma delimited text files. The easiest way to create these files is to save an Excel spreadsheet as a `.csv` file[17]. If you haven't done this, it is relatively easy, simply choose csv as the file type when you are saving a spreadsheet, as shown below.

7.2.1 Formatting Excel Files for R

There are a couple of important considerations in formatting Excel files for exporting to R. First, what to do about missing data? R reads blank cells in Excel as NA, which is the way that R represents values that are missing. A better way, in my opinion, is to enter NA for all cells with missing data in Excel. As an alternative, you can use some other consistent representation (9999 is often used) and convert it to NA once you have imported the dataset into R. For example, presume M is a vector, matrix, or data frame. The statement

```
M[M==9999]=NA
```

converts 9999 to NA throughout M.

The second consideration is column headings. These must be one word. If you need to use 2 words for clarity, connect them with a . or a _. For example, study.site or study_site are fine as column headings, but study site will not work.

Reading .csv files is accomplished with the following syntax:

```
name <- read.csv("pathname")
```

where `pathname` is the full path to the file you want to open and name is the name of the data frame (more about those in a moment) that you want to create. This appears to be simple, but actually you can really get messed up here because Windows pathnames are long and because R requires some special syntax for them. Fortunately, you can find the pathname name that R wants using `file.choose()` This handy command opens a File Select window that allows you to browse your computer for the correct file. When you click on the file, its full pathname, in proper syntax, is written to the console. You can then copy the pathname and surrounding " " and paste it between the ( ) of `read.csv( )`.

[17] After you have saved your `.csv` file, it is a really good idea to look at it with a text editor like Word Pad, Text Edit, or the R editor. Sometimes, for reasons unknown to me, Excel adds a bunch of extra columns, which show up as „„. That is, there are no values between the commas. If you import this file into R (as I describe subsequently) then your data have columns with nothing but NA. Fortunately there is an easy fix. Simply open the `.csv` file in Excel and save it again. The empty columns and the „„ are eliminated, eliminating the columns of NA in R.

7.2.2 An important caution about importing files from Excel

Sometimes you will edit a file in Excel that has already been imported into R. You may have some new data or you may find an error that you want to correct. Be very careful to save the file as `.csv`! If you make the correction on an .xls file without saving it as .csv and them re-run you R program to import a `.csv` file, well, nothing changes in R. I have done it more than once.

One way to avoid this problem is this problem is to delete the .xls file after the .csv file has been made and do your editing directly on the `.csv` file. You can work with it in Excel just like any other file, but when it comes time to import it to R, your changes will assuredly be saved. However, it can be very useful to document original data files with annotations an notes that are permitted in the.xls format, but not in the `.csv`. I use this documentation a lot. So, in this case, your only option is to be very careful about saving the `.csv` version of the file after every edit of the `.xls` version.

7.3 Data Frames

When you import a file into R, it creates a data frame. A data frame is a particular type of list that includes components consisting of vectors (numeric, character, or logical), factors, numeric matrices, or other data frames. Factors are special types of data relevant to statistical analysis, particularly general liner modeling and other analyses that combine category (or class) and numeric data in a single analysis. For all intents and purposes, all character data in a data frame are treated as factors. The purpose of data frames is to organize all data relevant to a particular problem in one place that can be accessed using the list $ and [] notation that we learned above. There are two important differences between data frames and the tabular data structures you have already learned about. First, unlike arrays and matrices, which must contain a single type of data (arrays can contain numbers or characters but not both; matrices must contain numbers), data frames can include numeric, logical, and character data. Second, and this is very important, the columns of a data frame have names[18]. These can be accessed using the statement

```
names(name_of_data_frame)
```

The data frames we will use are analogous to spread sheets where one or more column may contain character information and other columns contain numerical information. The R object `elk.data` that you created above is a data frame that has 3 column names, Year, Population_Size, and SE. You can change (or add) names to data frames using syntax like

```
names(name_of_data_frame)[3] = ‘‘newname’’
```

which assigns `newname` to column three of the data frame called `name_of_data_frame`

[18]It is also true the rows of data frames can have names and that the columns and rows of matrices can have names. Although we will rarely use these features in this course, you should know they exist.

Table 4: Extracting elements of data frames.

Data element(s)	Column name notation	Bracket notation
Vector containing years	`elk.data$Year`	`elk.data[,1]`
Second element of the year vector	`elk.data$Year[2]`	`elk.data[1,2]`
Fifth row		`elk.data[5,]`
Vector containing population size	`elk.data$Population.size`	`elk.data[,2]`
All data for years $>$ 1980	`elk.data[elk.data$Year > 1980,]`	`elk.data[elk.data[,1] > 1980,]`

7.3.1 Accessing components of data frames

Once the data frame has been created you can access it two ways. First, as with any list, you can access its components using the $ and as with any matrix, the [] notation that you learned about above(sections 5.6,7.1). So, think carefully about the following examples using the data you imported above. These examples build on what you have already learned about vectors, arrays, and lists.

Be sure you understand the correspondence between column name notation (middle column above) and array notation (right column above).

Exercise: Manipulating data frames with $ and []

1. Enter each of the above examples into the console of the program window and observe the result. It is particularly important to get your commas right and particularly important that you understand what the commas are doing. Discuss the above examples with your lab mates until you understand the syntax of each one, especially the comma and $ notation.

2. Execute R statements that a) Assign the column containing the variable SE to a vector called std.err. b) Extract the population size during 1990. c) Create a vector name big.years containing all of years for which the population size was greater than 1000 animals. c) Create a data frame named big.years containing all of the data for which the population size was greater than 1000. Hint—if assign rows and $>$ 1 column of a data frame to a new variable, the variable becomes a data frame. d) Create a data frame named big.years containing populations sizes and SE's for population sizes greater than 1000.

3. Use two types of syntax for each statement.

4. Create a data frame for this exercise using the syntax `elk.data.NA = edit(elk.data)`

5. After executing this statement, a spreadsheet-like editor will open. Insert a few 9999's to replace the real data, then close the window, and enter elk.data.NA at the console to see the missing data. Using one statement, replace all of the 9999's in elk.data.NA with NA. This turns out to be a very handy statement. More about missing data later.

Table 5: Extracting elements of data frames.

Data element(s)	Column name notation	Bracket notation
Vector containing years	`Year`	`elk.data[,1]`
Second element of the year vector	`Year[2]`	`elk.data[1,2]`
Fifth row		`elk.data[5,]`
Vector containing population size	`Population.size`	`elk.data[,2]`
All data for years $>$ 1980	`elk.data[Year > 1980,]`	`elk.data[elk.data[,1] > 1980,]`

7.3.2 Accessing data frames using `attach( )`

Ok, up to now we have done things the hard way, because then you will appreciate the easy way. The syntax in column 2 of the table above that relies on the $ operator is admittedly complicated. Is there a way to simplify these statements? You can dispense with the elk.data$— part of the syntax above using the statement:

```
attach(elk.data)
```

After you executed the `attach( )` statement, then the statements in Table 5 have the same effect as the statements in Table 4.

Thus, by using attach(elk.data), R knows you mean elk.data$Year when you give it Year. It knows that you mean elk.data$Year[2] when you give it Year[2]. So, the attach( )statement allows shorthand because R knows the column headings associated with the data frame you have attached. To make it forget what the column headings, you use the statement `detach(elk.data)`. It is important to understand that attach and detach don't have anything to do with whether R sees the data frame—they only affect the named variables in the data frame. To illustrate enter the following code line by line:

```
attach(elk.data)
elk.data
Year
Population_size
SE
detach(elk.data)
elk.data
Year
Population.size
SE
```

Exercise: Using attach and detach After attaching elk.data, find the data for which the standard errors (SE) are $<$ 100 with a statement using the column name SE.

These are important concepts, so be sure that you understand them before proceeding. Talk things over with me and with the lab instructors until you have a firm grasp of how to manipulate data frames.

7.3.3 Why not always use `attach( )`?

Because `attach()` seems so convenient, why not simply use it all the time, forgoing the somewhat cumbersome $ notation. The reason is this: convenience always comes at a price and the price here is potential confusion with what data you are accessing. As long as you working with a single dataset (which we often will do), then using `attach( )` is just fine. There is no potential for confusion. But, when you are working with more than one dataset, particularly datasets that have the same name for different variables, then it is easy to see that you can create some nasty errors if you forget which data set is attached. That is why the most carefully coded programs use $ and [] notation to specify what data are being accessed. One other alternative for doing so, is the `with()` statement, which is described next. It is probably telling that I almost never use attach().

7.3.4 Accessing data using `with( )`

If you want to specify the data frame to use with a specific function you can use a with ( ) statement. For example, if you want to know the mean of `Population_size`, you would use

```
with(elk.data, mean(Population_size))
```

To summarize the data for Population.size you would use

```
with(elk.data, summary(Population_size))
```

Multiple statements can be executed within the context created by with using:

```
with(data frame name,{
statements
})
```

Just as a warning, you are likely to forget to put the `)` after the closing `}`. The statements will not execute unless you do.

Exercise: Using `with()` to couple datasets to functions Detach the elk.data data frame. Plot the population size as a function of year using variable names and a `with ()` statement. Then do the same plot using $ and [] notation. You should have 3 plots generated by 3 lines of code. For Windows users, , put the following statement: par(ask=TRUE) in front of your plotting code. This will allow you to scroll though the three plots one at a time (by hitting enter) rather than having them displayed on top of the other.

7.4 Manipulating elements of data frames

The data you get is almost never in the specific form that you need, so a crucial skill in the analysis of ecological models and data is to be able to manipulate datasets by summarizing, subsetting, transposing, sorting, joining and merging, processes that are often referred collectively to as data reduction. This is a deep topic and I am constantly sharpening my data reduction skills with new tricks. Here, I provide some of the R tools I use most frequently. Remember, however, in R there are almost always more than one way to accomplish the same thing, so what I am showing you here is not the only code that works. Moreover, although I am illustrating several R functions using data frames as examples, they can also be applied to vectors and matrices.

7.4.1 Joining

Often we have columns (or rows) of data that we want to bring together to form a new data set. This task in accomplished using `cbind()` for columns and `rbind()`for rows. The syntax joining columns is: `cbind(object1, object2)` where the two objects can be vectors, matrices, or data frames. The two objects must have the same number of rows. Here is an example to illustrate.

We have the classic time series of data on lynx pelts taken by Canadian fur trappers (I will give you an appropriate Excel file containing these data). The data consist of the number of lynx pelts in a 114 year sequence, but we don't have those data associated with specific years. So, our data frame has an observation number ranging from 1 to 114 and the observed number of lynx pelts and we want to add a column containing the years. Execute the following code and study it.

#Find path for Lynx data

file.choose()

```
#Import lynx data
#Note capital L in Lynx to prevent conflict with the
#internal R data set named lynx. Also note that you need to
#put your path in the following statement.
Lynx = read.csv(Put path to your data here)
#Create sequence of years
Year = seq(from=1821, to=1934, by = 1)
#Check length of year sequence
length(Year)
#get column names from Lynx data frame
names(Lynx)
#check number of rows in Lynx data set
nrow(Lynx)
#Join columns of Lynx data with sequence of years
#Lynx_by_year = cbind(Lynx,Year)
head(Lynx_by_year, n = 10)
```

```
#Plot lynx abundance by year
with(Lynx_by_year, plot(Year, Lynx, type = "l"))
```

There are some things you haven't seen before in this example. Notice that we use the `names( )` statement to find out the names of variables in the Lynx data frame. We check the number of rows in the Lynx data set using nrow(Lynx). This must be the same as the length of the vector we want to join to the Lynx column. We create a new data frame called Lynx_by_year using the `cbind( )` command that glues the columns of one data frame to the column(s) of another data frame, array, or vector. To be sure we got things right, we then take a look at the top 10 rows of the new data frame and the column (i.e., variable) names using `head( )`. Finally, we plot the new data.

Two other points. First, you can bind together more than 2 data frames, as many as you like really, but they must have the same column lengths. Second, the left-right order of the data frame names in the `cbind( )` statement determines the left-right order of the columns in the new data frame. The leftmost argument in the `cbind()` statement defines the leftmost columns in the newly created data frame. `rbind ( )` works similarly, but it joins rows rather than columns. The leftmost argument to `rbind( )` determines the rows at the top of the newly created data frame. Data frames to be joined must have the same number of variables (columns).

7.4.2 Merging

Joining requires that the two objects we bring together have the same number of columns (or rows). There are many times, however, when we want to bring together data that differ in the number of rows, but that share a variable, which is to say that have a column in common. This is best understood by example. I provided you with a couple of condensed versions of files that I recently used modeling the effects of lynx predation on reindeer in Sweden. The first file ("Reindeer_units.csv") contains columns for index, year, unit name, and number of lynx family groups in a unit. The second file ("Reindeer_area.csv) contains index, county abbreviations, and the area of the unit. We want to bring these two data frames together so that we can calculate lynx density in each unit. We do this by merging them on the shared variable, index. The syntax for merge is

```
merge(data1,data2, by=shared.column)
```

Exercise Merging data frames: Read in the two Reindeer datasets and produce a single data set with columns `index, year, unit, lynx.total, county, total.area.`

7.4.3 Ordering data frames

Often we want to arrange the rows of a data frame ordered from top to bottom using some particular column. Using the reindeer data as an example, we might want to create a data frame ordered by year. Presuming that the name for the reindeer data frame is y, the syntax for producing a new data frame ordered by year is:

```
y.ordered = y[order(y$year),]
```

You really must think about this to understand what is going on. Take the expression apart, executing the following statements in sequence:

```
y.ordered
y
y$year
order(y$year)
y[order(y$year),1:2]
y.ordered = y[order(y$year),]
y.ordered
y
```

Discuss this with your lab mates. If you don't understand this thoroughly, call on me or the TA's. One more point–you can accomplish the same thing using the sort() function, but I avoid it because it can result in data frames that are misaligned–that is, where one column is sorted and the others are not. Obviously, this is an undesirable result, creating errors that are very hard to detect.

7.4.4 Editing data frames

Sometimes you want to edit data without going through the trouble of re-entering it from Excel. As you saw above, you can do this using the statement `edit(data frame name)`. For example, enter `edit(elk.data)` from the console. As you can see, this opens up a spreadsheet interface that allows you to manually edit the data. I strong recommend against doing this.

Here's why. Your programs should provide absolute documentation of your work starting with the data file you imported. If you make manual changes in a data frame in R, and then import that file again the re-imported file will not reflect your edits. It is really easy to forget about the changes you made. So, I urge you to make you data edits in Excel and re import the edited file using the `read.csv()` approach. This way you can keep all of the steps you executed in one program. I know this sounds like a lot of trouble, but believe me, it will save you headaches in the long run.

7.4.5 Dealing with missing values

As you have learned, the symbol for missing data in R is NA. There are a couple of things you need to know about missing data. First, many functions in R will not accept missing data unless you use the proper logical switch. For example, enter the following at the console:

```
b = c(3,2,5,NA)
mean(b)
mean(b, na.rm = TRUE)
```

What is going on here? The first `mean()` function returns NA because any calculation that includes NA returns NA. In the second mean statement, you set a switch, `na.rm = TRUE`, which causes missing values to be eliminated before the mean is calculated.

Now, what if you wanted to do this yourself, that create a new vector `c` from `b` such that c does not contain any missing values. Well, it would seem that you could use a statement like

```
c =b[b != NA]
c
```

Try it and see what you get. Hmmm. That seem odd. Now try this

```
c = b[b != 2]
c
```

So, it seems that R treats values like 2 and missing values like NA differently. Yes, NA is not a value but is simply a marker for something that is not available. So, when you set up a logical expression with NA like `b != NA` , R treats it as un-decipherable because you are looking for something that R doesn't know anything about. Therefore, all of the values it returns are NA. Now try this:

```
is.na(b)
!is.na(b)
b[is.na(b)]
b[!is.na(b)]
```

Discuss what is happening with your lab mates. This shows that for vectors, you can eliminate missing values using `!is.na( )` and the usual logic for subsetting we have learned above. To eliminate entire rows of data from data frames that contain any missing values, use `na.omit()`. If you give a data frame name to `na.omit( )`, it returns a data frame after eliminating all rows with missing values. However, you should be cautious using this. Remember, it eliminates all rows with missing values, and this may be throwing away data for many relationships in your data.

Exercise: Manipulating data frames with missing data Convert the single value of NA in the elk.data data frame to 9999 using a single line of code without eliminating any rows.

7.4.6 Converting arrays and matrices to data frames

Sometimes it is helpful to convert an array or matrix into a data frame (and vice versa). This allows you to give names to column headings, the use of which makes your code easier to understand. In other words a statement like `plot(A$year, A$N.content)` is far more informative than `plot(A[,1], A[,5])` but both could do the same thing if the data structure A has data for year in column 1 and nitrogen content of litter in column 5.

Enter following example at the console, using the matrix M that you created earlier:

```
#Create a new data frame from the matrix M:
is.data.frame(M)
is.matrix(M)
D = as.data.frame(M)
is.matrix(D)
is.data.frame(D)
names(D)
```

This illustrates a bunch of stuff. First, we can test to see if the R object `M` is a data frame or a matrix using the `is()` statement[19], showing how you can determine if `M` is an object of a given class. [The inquisitive among you will also wish to explore `mode(M)`, `type(M)`, and `class(M)`]. You create a new data frame `D` by coercing the matrix `M` into a data frame using the `as.data.frame( )` statement and list the column names of D using the `names( )` statement. Notice that R assigns default names to the columns of `D, V1` and `V2`. This is considerate of R, but not terribly helpful because `V1` and `V2` are no more informative than `M[,1]` and `M[,2]`. To improve their clarity, you can give new, more informative names to the columns of `D` using:

```
names(D) = c(‘‘year’’, ‘‘mineralization’’)
```

8 More about plotting

Graphics are a very deep topic in R and we will only cover a tiny bit of its graphical capability here. For a great resource for learning what R can do, see

`http://addictedtor.free.fr/graphiques/thumbs.php`

Up to now, we have used simple plot statements producing unadorned 2 dimensional plots of one variable. Here, we cover some ways to improve them.

8.1 Controlling Output to the Plot Window

The default method for writing plots to the graphic window of R, oddly (well, maybe not for Windows), is to simply write one plot on top of another, so that when you make multiple plots in a program, the only one you will see (unless your eyes are quicker than mine) is the last one. In Mac OS R you can sensibly scroll through the plots in the Quartz window by clicking on Quartz and Back or Forward or by putting the cursor in the Quartz window and pressing cmd arrow left or cmd arrow right. In Windows, if you want to scroll through a series of plots, use

```
par(ask=TRUE)
```

Each time you hit Enter, another plot is produced. For Windows, I have not yet figured out how to keep a plot history (the way SAS does and Mac OS does), allowing you to scroll up through plots you have created.

[19] Paraphrasing a former US president, you can find out “. . . what the meaning of is is.” by doing `?is` at the console.

8.2 Options for plotting

Two dimensional plotting will be a workhorse in this course. The basic syntax of the plot function is:

```
plot(x,y,options)
```

where x is a vector of values for the x-axis, y is a vector of the same length for the y-axis[20] and options are optional statements to change the appearance of the plot. So, I admit, the plots we have done so far are pretty ugly—weird labels, funny symbols, etc. In this section we learn ways to make the plots look spiffy. This is a deep topic and I refer you to the R documentation(enter `?plot.default`) for a thorough treatment. You should also understand that the options that work for the plot( ) function also work for many other types of graphics. Here, I will cover the options that will be most useful to you. You can mix and match these as needed by separating them with a comma in the plot statement

8.2.1 Specifying the type of plot

You have already seen how to plot lines or points using the option `type =` within the plot statement, i.e. `plot(x,y)` for points (points are the default) and `plot(x,y, type = ‘‘l’’)` for lines. Here are some other, potentially useful types:

`type="b"` Plot points connected by lines

`type="o"` Plot points overlaid by lines

`type="h"` Plot vertical lines from

See documentation for plot (?plot at the console) for some additional types.

8.2.2 Adding labels

You have no doubt noticed that R will label the x and y axis with the variable names you give it in the `plot(x, y)` statement. In the case of data frames, R is smart enough to put the column labels on the appropriate axes. But what if you want to specify the label? You use an option in the plot statement like:

```
plot(x, y, xlab="text" , ylab = ‘‘text’’)
```

To illustrate, plot the lynx data using:

```
with(Lynx,
plot (Year, Lynx, xlab = "Year", ylab = "Number of Pelts’’)
)
```

[20]The y vector must contain the same number of elements as the x-vector, but some of those elements can be NA. They will simply show up a blank in the plot.

8.2.3 Scaling the axes

Often you may want to limit a plot to a specific range of the data. To do this, you use the options:

```
xlim = c(lower, upper)
ylim = c(lower, upper)
```

where lower and upper are the endpoints of the axes.

8.2.4 Mathematics, Greek symbols, subscripts and superscripts

Sometimes you need labels that include non-conventional fonts, like Greek symbols or superscripts. You also may want to annotate your plots using mathematical expressions. This is shockingly easy to do in R. Here is an example. Lets say you wanted to put the following text on the y axis of your plot "Nitrogen Mineralization Rate (g/m2)." To do this you would use

```
ylab = (expression(paste("Nitrogen mineralization g/", m^2)))
```

There are a few key bits here. First, you use `expression( )` to tell R that there will be some special characters needed in the y axis label. If all you needed were special characters (e.g, m^2) then your work would be done. You would simply write

```
ylab =expression(m^2)
```

However, you want to combine regular characters with special ones, which is where the R statement `paste( )` comes in. `paste( )` makes 1 character statement from the arguments you give it so, for example, enter the following at the console:

```
new = paste(''A'', ''B'', ''C'')
new
```

So in the `ylab =` option, you paste together the text string "Nitrogen mineralization g/" with the special expression m^2 to produce the desired string. The `expression ()` statement can be used to do all sorts of special things, for details see `?plotmath`. Figuring these out can easily occupy an otherwise dull afternoon.

Sometimes, getting the `expression()` statement to do what you want is a bit more trouble than it is worth. So, if I want presentation quality graphics and can't do make it work relatively quickly, I just punt and make a graph without labels in R and then add them in another program like Power Point or Paint. To make a plot without labels specify

xlab = " " and ylab = " " as options to the plot statement.

Table 6: Some codes for symbols and line types used as options in R plotting statements.

Symbol	pch code	Line type	lty numeric code	lty text code
blank	-1	blank	0	
solid circle	19	solid	1	"solid"
bullet (smaller circle	20	dashed	2	"dashed"
square	21	dotted	3	"dotted"
diamond	22	dotdash	4	"dotdash"
triangle point-up	23	longdash	5	"longdash"
triangle point-down	24			
bullet (small circle)	25			

8.2.5 Adding headings

Headings can be added with the option `main = ''text''`. For example,

```
with(Lynx_by_year,
plot(Year, Lynx, xlab = "Year", ylab = "Number of Pelts",
main ="Canada Lynx Trapped")
)
```

Again, note the double parentheses at the end of the statements—one to close the `with()` statement and another to close the `plot()` statement.

8.2.6 Changing symbols and line types

It is possible to change the plotting symbols using the `pch =` option and line types using the `lty =` option (Table 6)

8.2.7 Multiple plots on a page

It is often useful to make collages of plots on a single page. To do that, use the statement

```
par(mfrow=c(row,col))
```

where row is the number of rows and col is the number of columns containing plots. So, for example

```
par(mfrow = c(2,2))
```

will output four plots, two across the page and two down. If you only produce three, one window is left blank initially, but thereafter the plots go in sequence, which may cause you to loose one or more. It is best if the number of rows times number of columns equals the number of plots you will produce. If you want to produce an odd number of plots, simply execute the `par()` statement before you call the plots.

8.2.8 Specifying options in one statement

You can set up a series of options for plotting in a single statement such that these do not have to be re-specified every time you plot. This is done with the `par( )` statement. For example:

```
par(ask = TRUE, pch = 22, lty = ‘‘dashed’’)
```

will cause all subsequent output to pause awaiting an Enter to proceed, will set the default plotting character to a square and the default line type to dashed. The many options available to par can be discovered by entering `?par` at the console. The settings of the current options can be viewed by entering `par()`.

8.2.9 Overlaying points and lines on a basic plot

After a basic plot has been made using a plot() function, you can add lines or points to the plot using[21]

```
#overlay lines
plot(x1-data, y1-data, options)
lines(x2-data, y2-data, options)
#overlay points
plot(x1-data, y1-data, options)
points(x2-data, y2-data, options)
```

Options for the lines and points functions are limited; for example `xlab` and ylab must go with the original plot and are ignored if you put them with `lines`() or `points()`. However, you can specify options for line types and symbols in `lines()` and `points()`.

8.2.10 Adding legends

When you have a series of data on the same plot, it is nice to be able to place a legend on the plot to identify the plotting symbols used. The syntax for a legend statement is

```
legend(x,y, legend=text, pch=v1, lty=v2, col = v3)
```

where `x` and `y` give the coordinates for the upper left corner of the legend scaled in the same units as the two axes, `text` is a character vector giving the text of the legend, ordered in the same way as the columns being plotted, `v1` is a vector of symbol codes, `v2` is a vector of line types, and `v3` is a vector of color codes. One or more of the `v` vectors can be included in the legend.

The following code produces the plot in Figure 3.

[21] There is an important caveat here if you are using R Markdown. You must put the `plot()` statement and the `lines()` and/or `points()` statement within the same chunk. R Markdown forgets that a plot has been called between chunks.

```
h=.04
Rmax=15
S=seq(0,4,.1)
I=S*Rmax/(h*Rmax+S)
data=rnorm(length(I), I, .10*I) #generate some fake data
plot(S,I ,typ="l", ylab="Dry matter intake (g/min)", xlab = "Bite mass (g)", ylim=c
points(S,data, pch=19)
text=c("model prediction", "observations")
sym=c(-1, 19)
lines=c(1,0)
legend(1,6,legend=text, pch=sym, lty=lines, bty="n")
```

8.2.11 Exporting graphics to documents

Depending on what word processor you use[22]it is often possible to simple copy the contents of the graphics window and paste them into a document. From experience I know that this works well with the beautiful .pdf output from the Quartz window in Mac R into Mac programs like Pages and Keynote, and if you use Windows (my sympathy), I think it works for pasting pictures into Word as well.

However, there is a way that should always work, regardless of which word processing or presentation program you use. Consider the following:

```
pdf(file=''filename.pdf'')
#put your plotting code here
dev.off()
```

What this does is to route the graphics output into a pdf file saved in the location specified by "filename.pdf". The trick here, which took me too long to learn, is that *every* time you output a file, you must close it using `dev.off()`. Otherwise, the receiving program (e.g., Adobe Acrobat, etc) will think the file is corrupt. If you get that message, then chances are you forgot to close the file with `dev.off()`.

You can write other types of graphics files, for example `ps(file=''filename'')` exports postscript files, `jpg(file=''filename'')` exports jpegs, and `tiff(file=''filename'')` does tiffs. For more details and options, follow your nose from `?device()`.

8.3 Some other plotting functions

As I said above, R graphics are a very deep topic and I can't hope to do justice to that depth here, but there are a few other plotting functions that can be very useful. I will briefly here.

[22]If you are a scientist and you are using the same software to write your documents as soccer-moms use to make banners and lawyers use to prepare briefs, then perhaps you should explore a different word processing program. This document was prepared with LyX, a front end for LaTex, which, in my view, is the ultimate tool for scientific writing. Appreciation to my son, Nicholas Hobbs, an engineer at Google, who told me about it. It is open source, rock solid, and free. See `http://wiki.lyx.org/LyX` You will learn to use LyX as part of this course.

8.3.1 Plotting columns of matrices as series

Often we will want to plot a series of responses against the same x-axis. This is done using `matplot( )`, which is shorthand for matrix plot. The syntax is

`matplot(x,y,options)`

Where x is a vector of values for the x-axis and y is a matrix. The plot includes one series for each of the columns in the matrix y. It is analogous to overlaying a line or points plot for each of the columns. The default symbol used by matplot() is the number of the column. You can modify this to make a prettier plot using a vector specifying symbol or line codes (Table 6).

8.3.2 Plotting mathematical expressions

Building models usually requires that we visualize equations and examine how they behave as parameter values change. R has a very nice tool for doing that. To illustrate, consider the Michalis Menten function, which is commonly used in ecological modeling to describe phenomena that change asymptotically with changes in an independent variable:

$$y = \frac{\mu x}{h + x} \tag{16}$$

To plot this function from 0 to 100 you would use:

```
mu = 100
h = 30
curve(mu*x/(h+x), from = 0, to = 100)
```

As with all options, this can be condensed to `curve(mu*x/(h+x), 0,100)`as long as the order of the arguments is correct. If you want to put all of the statements on the same line to allow you to recall them more easily in the console, then you can use `mu =100; h = 30; curve(mu*x/(h+x), from = 0, to =100)`

So, the general syntax is

```
curve(equation, from = f, to = t)
```

where `equation` is the expression you want to plot, `f` is the lower limit and `t` is the upper limit. The equation must contain an `x` for the independent variable. The parameters (i.e., `mu` and `h` in the example above) must be given numeric values.

Exercise: Exploring the Gompertz equation: Recall the equation we used earlier to describe the rate of growth of a plant as a function of its biomass:

$$\frac{dB_t}{dt} = \mu B_t \left[1 - \frac{k}{\mu_0} \ln\left(\frac{B_t}{B_0}\right)\right] \tag{17}$$

Explore the behavior of this function using the `curves( )` function by varying its parameters, . To start, use values in the neighborhood of μ = 1, k = .3, and B0 = 10,

Algorithm 4 Example code for creating a table for importing into a spreadsheet.

```
data(CO2)
#make a table with first 10 lines of CO2 dataset
table=CO2[1:10,]
#write table to working directory
write.csv(table)
```

and the range of independent variables 0 to 300. Then try some radical departures in parameter values which will probably require rescaling the x axis to see what is going on. Describe the effect of each parameter on the behavior of the function.

8.4 Making publication quality output

All of the plots you have done so far went to the plot window. How do you send a plot to a file? The way I usually do it is

```
pdf(file=''Figure_1.pdf''), height=6, width=6)
#plotting code goes in here
dev.off()
```

This write a pdf file called Figure_1.pdf to the working directory. You can specify a full path name if you prefer. There are also drivers for postscript files (`postscript(file =....)`) and jpegs (`jpeg(file=....)`). All of these require that the plotting code is ended by the `dev.off()` function. See R help for the specifics of using these very helpful functions.

9 Special topics

9.1 Making tables

9.1.1 Output to spreadsheets

One of the easiest ways to make a table is to construct a data frame in R, write it to a `.csv` file, and import it to a spreadsheet.

10 Answers to exercises

10.1 The diameter of the earth

```
#Diameter of the earth
ckm = 24901.55 /.6215 ckm
earth.diameter = ckm / pi
earth.diameter
```

10.2 Vectors

```
#Vectors
v1 = c(234, 17, 42.5, 64)
v1[2]
ages = c(21,23,26)
```

10.3 Logical operations on vectors

```
v1.new = v1[v1>20]
v1.new v1.new = v1[v1 > 20 & v1 < 200]
v1.new v1.new = v1[v1 != 17]
v1.new
v1.new = v1[v1 == 17 | v1 == 42.5]
v1.new
```

10.4 Creating a matrix

```
#Creating matrix
B = matrix(7, nrow=5, ncol = 7)
B
```

10.5 Matrices

```
# Answer to matrix exercise
M <- matrix(0,nrow=5, ncol=2)
#Enter data M[1,] <- c(1995, 10.0)
M[2,] <- c(1996, 12.2)
M[3,] <- c(1997, 15.6)
M[4,] <- c(1998, 19.5) M[5,] <- c(1999, 24.4)
#Outputs rows
M[1,]
M[2,]
M[3,]
M[4,]
M[5,]
#Outputs columns
M[,1]
M[,2] #
Find largest value in matrix
max(M)
#Find average of second column mean(M[,2])
```

10.6 Matrices, continued

```
#Find all values in column 2 < 19
M[M[,2] < 19,2]
#Finds and outputs the row containing the minimum value of # column
2 M[M[,2] == min(M[,2]),]
```

10.7 3 dimensional arrays

```
#capture history example
#create the array
x = array(0,dim=c(4,5,100))
#output first 5 tables
x[,,1:5]
#set initial conditions for all animals to susceptible, alive
x[1,1,]=1
x[,,1:5]
#create history for animal 5
x[1,2,5]=1
x[2,3,5]=1
x[2,4,5]=1
x[4,5,5]=1
x[,,5]
```

10.8 A Gompertz model of plant growth

```
#Gompertz model-vector version for exercises 1 and 2.
#parameter values
k =.3
mu = 1
# vector for holding plant biomass B = numeric(30)
R = numeric(30) #added for exercise 2
r = numeric(30) #added for exercise 2
#inital plant mass B[1] = 10
for (t in 2:length(B)){

    B[t] = B[t-1] + mu* B[t-1]*(1-(k/mu * log(B[t-1]/B[1])))
    # the next two statements are added for exercise 2
    R[t] = mu* B[t-1]*(1-(k/mu * log(B[t-1]/B[1])))
    r[t] = R[t]/B[t]

}
par(mfrow=c(2,2))
plot(B) x = seq(2,30)
plot(x,R[2:30])
plot(x,r[2:30])
```

```
#Gompertz model-array version for exercises 3.
#parameter values
k =.3
mu = 1
# matrix for holding time (column 1) and plant biomass (column 2)
B = matrix(0,nrow=30,ncol=2)
#inital plant mass B[1,2] = 10
for (t in 2:length(B[,1])){

    B[t,1] =t
    B[t,2] = B[t-1,2] + mu* B[t-1,2]*(1-(k/mu * log(B[t-1,2]/B[1,2])))
    }

par(mfrow=c(2,2))
plot(B[,1],B[,2])
```

10.9 Nested for loops

```
#Model experiment example
#vector holding value of growth rate
lambda =seq(from = 1, to=1.6, by = .1)
#matrix with 10 rows and columns for each growth rate
N = matrix(0, nrow = 10, ncol =length(lambda))
#Initial value of N
N[1,]=100
#Loop over values of lambda
for (i in 1:length(lambda)){
#Loop to calculate N over time. Use t to index vector.
for (t in 2:10){
# store the values of N in array
N[t,i] = lambda[i] * N[t-1,i]
}
}
matplot(N, type = 'b')
```

10.10 Function for normalizing a vector

```
#Normalizing a vector
z = c(12.3, 45.6, 16.4, 88)
normalize = function (v){
normal.v = v/sum(v)
return(normal.v)
} z.norm = normalize(v=z) z.norm sum(z.norm)
# alternative syntax normaliz = function(v) normal.v = v/sum(v)
z.norm = normalize(v=z)
```

```
z.norm
sum(z.norm)
```

10.11 Functions for moment matching

```
shape_from_stats <- function(mu , sigma ){
 a <-(mu^2-mu^3-mu*sigma^2)/sigma^2
 b <- (mu-2*mu^2+mu^3-sigma^2+mu*sigma^2)/sigma^2
shape_ps <- c(a,b)
return(shape_ps)
}
a=shape_from_stats(mu=.7,sigma=.07)
#check the function
y=rbeta(10000,a[1],a[2])
mean(y)
sd(y)
#plot the beta distributoin
x=seq(0,1,.01)
y=dbeta(x,a[1],a[2]) #beta density function
plot(x,y,typ="l", xlab="y", ylab="P(y)")
#function for moment matching in either direction
mm <- function(x1,x2,flag){
if (flag=="shapes"){
mu=x1
sigma=x2
a <-(mu^2-mu^3-mu*sigma^2)/sigma^2
  b <- (mu-2*mu^2+mu^3-sigma^2+mu*sigma^2)/sigma^2
shape_ps <- c(a,b)
return(shape_ps)
} #end of shape if
if(flag == "moments"){
a=x1
b=x2
mu=a/(a+b)
var=a*b/((a+b)^2*(a+b+1))
shape_ps <- c(mu,var)
return(shape_ps)
} #end of moments if
} #end of function
#get shapes
a=mm(x1=.7,x2=.07,flag="shapes")
#plot the beta distributoin
x=seq(0,1,.01)
y=dbeta(x,a[1],a[2]) #beta density function
plot(x,y,typ="l", xlab="y", ylab="P(y)")
```

```
#get moments
b=mm(x1=a[1],x2=a[2],flag="moments")
mu=b[1]
sd=sqrt(b[2])
```

10.12 Lists

```
#List example
name = "Poudre"
n = 100
a = c(.30, .20, .50)
m = array(0,dim = c(3,3))
m[1,1] = 0
m[1,2] = 1.1
m[1,3] = 1.7
m[2,1] = .6
m[3,2] = .9
m[3,3] = .9
m
#create a list
population = list(location = name, size = n, composition = a, transition = m)
population
population$size
population$composition
population$composition[2:3]
population$composition[population$composition >.2]
population$transition
population$transition[1,3]
```

10.13 Lists and functions

```
AIC.stats = function(likelihood, K, n, small, log){
AICc = get.AIC(likelihood, K, n = 20, small, log )
Delta_i = get.Delta_i(AICc)
L=get.Lmod(Delta_i)
Akwt = get.Akwt(L)
stats = list(AICc = AICc, Delta_i = Delta_i, Like = L, Akwt = Akwt)
return(stats)
}
stats = AIC.stats(LLike, n = 20, K, small = TRUE, log = TRUE)
stats$Like
stats$Like[1]
stats$Akwt
stats$Akwt[3]
```

```
Data into R
#Data into R example
file.choose()
#make the following all one line in R
elk.data <- read.csv( "C:\\Documents and Settings\\tom\\My Documents\\Ecological Mo
elk.data
is.data.frame(elk.data)
```

10.14 Manipulating data frames with $ and [ ]

```
#Data frame exercise
std.err = elk.data$SE
std.err = elk.data[,3]
std.err
elk.data[elk.data$Year == 1990,2]
elk.data[elk.data[,1] == 1990, 2]
big.years=elk.data[elk.data$Population_size > 1000,1]
big.years = elk.data[elk.data[,2]>1000,1]
big.years
big.years=elk.data[elk.data$Population_size > 1000,]
big.years = elk.data[elk.data[,2]>1000,]
big.years
big.years=elk.data[elk.data$Population_size > 1000,2:3]
big.years = elk.data[elk.data[,2]>1000,2:3]
big.years
```

10.15 Using attach and detach

```
#attach and detach
attach(elk.data)
elk.data[SE < 100,]
Using with( ) to couple datasets to functions
#attach and detach
attach(elk.data)
#The , within the [] assure that all columns are included
elk.data[SE < 100,]
```

10.16 Replacing NA's

```
elk.data[is.na(elk.data)] = 9999
Using matplot( )
matplot(N2[,1],N2[,2:7])
#label for x = N2[,1]
#label for y = N2[,2:7]
```

10.17 Exploring the Gompertz

```
k =.3
mu = 1
B0 = 10
curve(mu* x*(1-(k/mu * log(x/B0))), from = 1, to = 300)
```

10.18 The final problem

```
#Final Problem
#Find path for new elk data
file.choose()
new.elk = read.csv("C:\\Documents and Settings\\tom\\My Documents\\Ecological Model
elk.data2 = rbind(elk.data, new.elk)
tail(elk.data2)
head(elk.data2)
#Elk model predictions
#set up vector to hold model predictions
N = numeric(length(elk.data2$Year))
#set initial conditions and parameter values
N[1] = 346
r = .172
K = 940
#conduct simulation
for(t in 2:length(N)){
N[t] = N[t-1] + r*N[t-1]*(1-N[t-1]/K)
}
#make data frame containing observations and predictions
elk.data3 = cbind(elk.data2, N)
#do plots
with(elk.data3, {
plot(Year, Population_size, type = "p", pch = 24, , xlab = "Year", ylab="Population
lines(Year,N)
points(Year, N, pch = 19)
legend(1968,1400,c("Observed","Predicted"),pch =c(24,19))
}
)
```

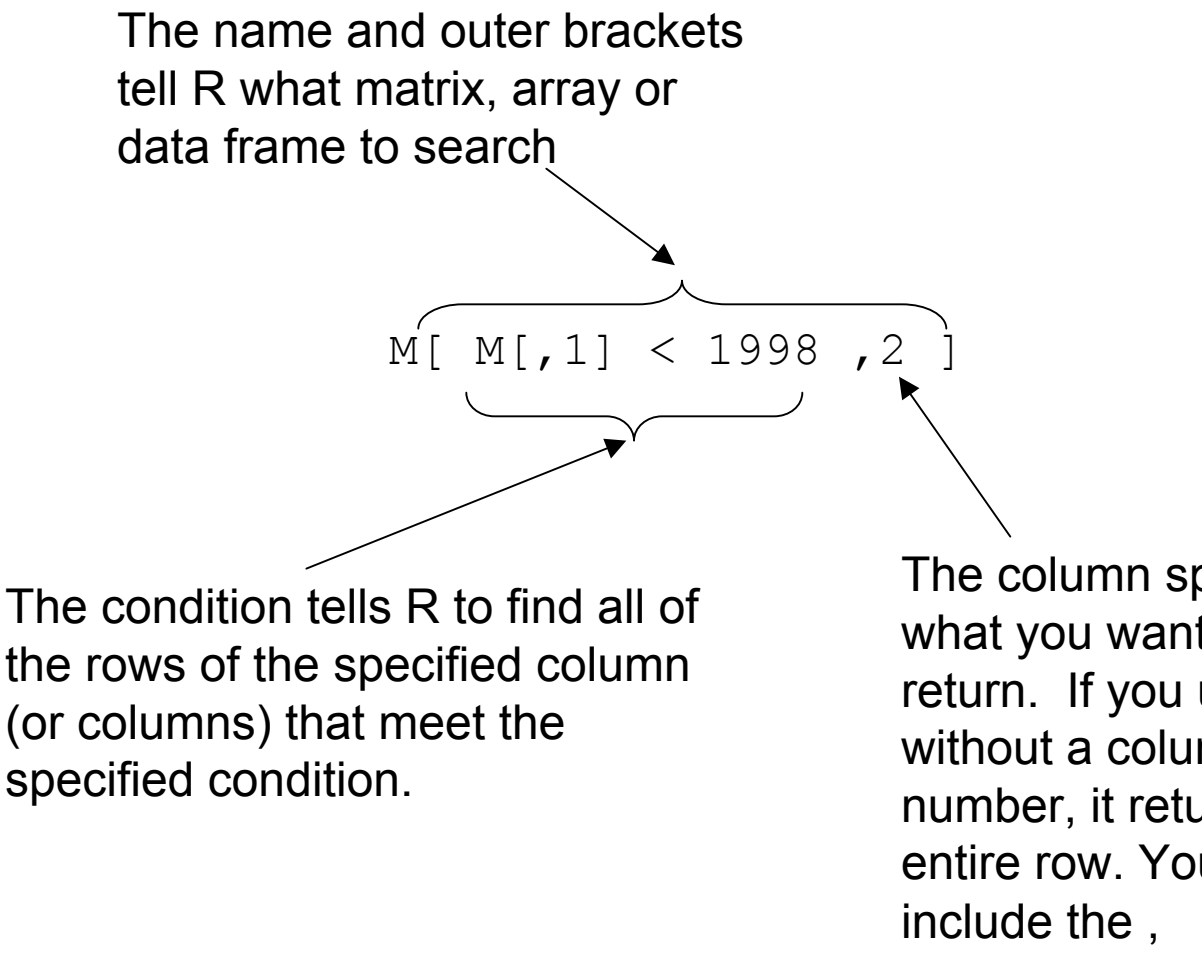


Figure 2: Illustration of command for subsetting a matrix.

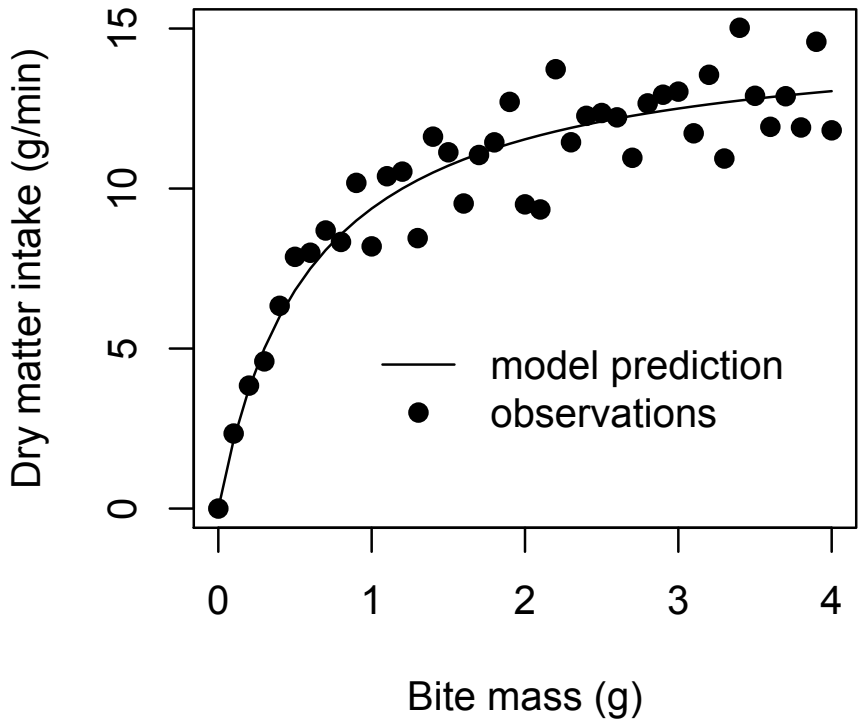


Figure 3: Illustration of `x-y plot using legend()` and `plot()` options.

Index

$ operator, 43
<-, 14

A
apropos(), 13
array(), 24
attach(), 48

B
Basic mathematical operations, 15
browser(), 38

C
c(), 17
cbind(), 50
console, 8
cor(z,y), 19
curve(), 60

D
data frame, 46
debugging, 38
detach(), 48
dev.off(), 59
dim, 24

E
example( ), 12
expression(), 56

F
file.choose(), 9
for(), 26
functions, 31

G
getwd(), 9
Gompertz model of plant growth, 28

H
headings in plots, 57

I
if(), 34
Installing R, 7
is.na(), 53

L
length(), 19
Libraries, 7
list(), 43
load(), 12
logical operations, 18
ls(), 10

M
main =, 57
matplot(), 60
matrix(), 21
max(), 19
mean(), 19
median(), 19
merge(), 51
min(), 19
missing data, 52
moment matching, 36

N
NA, 52
names(), 46
na.omit(), 53
na.rm = TRUE, 52
nested for loops, 30
NULL, 20

P
par(ask=TRUE), 54
par(mfrow=c(,)), 29
par(mfrow=c(row,col)), 57
paste(), 56
pch =, 57
pdf(), 59
plot(), 55
pmin(v1,v2), 19

R
range(), 19
rank(), 19

rbind(), 50
read.csv(), 45
return(), 32
rm(list = ls()), 11

S
scope, 37
sd(), 19
setwd(), 9
sort(), 19, 52
sum(), 19

V
var(), 19
vectors, 17

W
with(), 49

X
xlab=, 55
xlim, 56

Y
ylab =, 55
ylim, 56