EEID 2009 Workshop

The Basics of R for Ecologists: by Stu Field*

May 17th, 2009

Contents

*Specials thanks to the organizers of the 7^{th} annual Ecology & Evolution of Infectious Disease Workshop for insightful comments, suggestions, & examples in development of this tutorial, however all careless grammar, mistakes, and typos are my own.

1 Introduction to R

Reading a tutorial about the numerous applications and capabilities of R is a great start, but until you actually do it for yourself and are actually forced to solve problems for yourself in R, you will not become proficient. That's what the following exercises are designed to do; give you practical programming experience in R. I assume you've at least been introduced to R by skimming some of *An Introduction to* R *for Ecological Modeling* by Steve Ellner & Ben Bolker. Before you start, I recommend you get used to running R from script files, rather than the command window, it'll make your life much simpler. For those of you using a Mac, R already comes with a fairly nice script editor and for those using PCs, a supplementary script editor for Windows (e.g. Tinn-R), will simplify things significantly.

At the top of your script file you should start with `rm(list=ls())`, which removes all objects from R's memory. You'll use it often. To remove a single object use `rm(objectname)`. The objective of this tutorial is to create your own library of scripts for analyses which grows with your knowledge of R. Always be sure to annotate your scripts *diligently* with the '`#`' symbol, otherwise you may come back to the script months later and not remember what you did (extremely frustrating!). I tend to have paired files with the same name 'studata.csv' and 'studata.R', where the *.R file contains all the necessary commands, analyses, and plots for that particular data set.

Installing Packages

Packages containing additional functions for R to use can be downloaded from the internet.[1] There are essentially two steps to using packages, 1) you need to download & install the package onto your machine (once), and 2) you need to load the package (each R session). Step (2) is simple: type `require(packagename)`. Step (1) is done as follows (for a PC):

1. Make sure you are connected to the internet.
2. Click on 'Install package(s)' from the drop-down 'Packages' menu.
3. Select you CRAN mirror (I usually choose USA(MI) for no particular reason).
4. Select the package you want from the menu (in alphabetical order).
5. Hit OK & wait for it to install (should auto-install).
6. Package is now installed and ready for you to call for it (Step 2 above).

[1] The process if fairly simple, although some issues with admin rights may arise if you don't have rights to modify the R directory.

Not all packages are independent (i.e. stand alone) and therefore some depend on other packages. These packages sometimes already 'know' this and get the required secondary packages automatically, others times you will get an error message and have to install them yourself. Lastly, if you go to the Packages menu and click on 'Load package', you will see a list of the packages that have been successfully installed on the machine. For the Mac, the process is very similar except you click on 'Packages & Data' from the program menu and then 'Package Installer' for package installation; and to see which packages are currently installed on the machine, go to 'Package Manager'.

The Help Menu

One of the best features of `R` is the extensive Help menu for general information about syntax and arguments for a given function (`?function`). It is a very handy tool that I use almost daily. Let's take a look at the Help menu using the function `apply()` as an example. Type:

```
> ?apply
```

This should bring up a Help window with details about the `apply` function. First, you can see that this function *applies* a given function (`FUN`) to the row or column (`MARGIN`) of object (`X`). The object (`X`) will have to be a data frame, matrix, or array (any 2-dimensional object). The '`...`' refer to any additional arguments of (`FUN`) you may want. The '`Usage`' section tells you how to formulate your syntax and construct your arguments within the function and the '`Arguments`' section gives more information about the actual pieces of information `apply` will need to to its job. Here is an example:

```
> Mx = matrix(1:16, 4, 4)    # create a matrix called Mx
> apply(Mx, 1, mean)
> apply(Mx, c(1,2), sqrt)
```

This tells `R` to calculate the mean of the *rows* (, 1,) of `Mx`. In addition, the Help menu says in the `MARGIN` description that using `c(1,2)` tells `apply` to perform the function by row *and* by column (i.e. entry-wise). The third line above uses the `sqrt()` function to take the entry-wise square-root of `Mx`. **Note:** how you construct the command partially depends upon the function you want use (i.e. it wouldn't make sense to use the `mean` function with an entry-wise construction because you'd be taking the mean of one value and end up with the same matrix). Lastly, there is an '`Examples`' section for you to see `apply()` in action.

I have provided the full `R` code associated with all figures and exercises in this tutorial. The intent is that you first try to complete the exercises on your own, then only use this code when/if you get stuck. If you're *really* stuck, I suggest you leave it and we will take care of it during the workshop. Good luck & happy Rrrrring!

2 Importing Data

First things first. You need to get your data, which was perhaps collected in the field and now exists as an excel worksheet or something similar, loaded into R memory.[2] It's a relatively simple but essential process, yet you would be surprised how many unexpected issues can arise. The simplest, but by no means the only, way is to save your excel file as a comma delimited file (`filename.csv`). **Hint**: make sure your home R directory is set to the location where your datafile is located. Also, make sure your headings are clear but simple (i.e. no spaces, funny characters, etc.). The function `read.csv()` converts the .csv file into a data frame (a data frame is essentially a 2-dimensional array that can contains any combination of vectors (columns of data) that are of integer, numeric, or non-numeric (e.g. factors) class.

One last note on data file organization: it's important to know the difference between *wide* format and *long* format. Table 1 shows the same data organized in two ways. On the left, it's in long format: there are a few long columns since there's one row for each observation. On the right, it's in wide format: more, shorter columns, with multiple observations on each row. For some analyses it's important to have your data in one format or another. For whatever reason, most people seem to want to put their data in wide format, but many analyses require the data in long format. R provides a function, `reshape()`, that converts between the formats (somehow I can never remember how to use it!). Use whatever tool you like to get your data into long format.

Exercises

1. Read the .csv file `Tree data.csv`[3] into memory. Add the argument `(..., row.names = 1)` and note what it does. It is usually easier to assign the new data frame a name such as 'mydata' so that you can refer to it later if necessary (e.g. `tree.diameter = mydata$dbh`).

2. Call the data frame and take a look at what was loaded into memory. Explore the `ls()` function as well as these below to see what they do.

   - `names(mydata)`
   - `attributes(mydata)`
   - `sapply(mydata, class)`
   - `attach(mydata)`

[2]For this section and the next I suggest you look at Ch. 2 of Ben Bolker's book *Ecological Models & Data in R.*

[3]There is a package which can read *.xls files into memory (gdata).

Table 1: Example of how to organize your data for statistical analyses.

Long format			**Wide format**			
No.	Species	Variable	No.	*sp.* A	*sp.* B	*sp.* C
1	*sp.* A	10.88	1	10.88	13.38	4.31
2	*sp.* A	11.59	2	11.59	14.97	6.13
3	*sp.* A	10.91	3	10.91	12.01	5.66
4	*sp.* A	10.78	4	10.78	15.14	4.99
5	*sp.* A	10.00	5	10.00	12.60	5.77
6	*sp.* B	13.38				
7	*sp.* B	14.97				
8	*sp.* B	12.01				
9	*sp.* B	15.14				
10	*sp.* B	12.60				
11	*sp.* C	4.31				
12	*sp.* C	6.13				
13	*sp.* C	5.66				
14	*sp.* C	4.99				
15	*sp.* C	5.77				

Notice what `ls()` gives you before and after you attach your data frame. Are there any new objects? I prefer to assign each variable individually for this reason, so I know there is nothing going on in the background I can't see.[4]

3. The first thing you'll need to do is make sure your data set is complete, that means no pesky `NAs` (the code for missing data in R). First, determine if some of those `NAs` should actually be zeros. You can identify whether you have a complete data set with the command `complete.cases(mydata)` or if you have a *huge* data set you may want to simplify your life with `which(!complete.cases(mydata))` and hope R returns `integer(0)`. If not, and you're sure you want to remove cases with with missing data, you can remove them with `mydata <- na.omit(mydata)`.[5]

4. You wish to determine how many of each species of tree is present within the dataset. This is particularly effective if you are organizing data by a factor (such as species, which you just identified as a factor with `sapply()`[6] function above), or some discrete variable rather than a continuous one. Try

[4]`attach()` can have unintended consequences particularly because you might think that your variable is an object (why not? you can call it like one). The problem is that if you manipulate such an 'object' the changes aren't propagated back to the original data frame, thus 'x' and 'mydata$x' no longer are the same! In addition, you might attach two data frames where headers are duplicated. Which is which? Take home message; use `attach()` with care!

[5]Again, Ch. 2 of Bolker's book gives a detailed description regarding how to deal with `NAs`.

[6]The 'gdata' package has a simplified version called `ll()`, try `ll(mydata, dim=TRUE)`.

assigning a column of `mydata` a name[7] (i.e. `species = mydata$spp`) and using the `table()` function, `table(species)`. Try doing this with other variables within the data frame.

5. Lastly, especially for data frames with factors (which you will probably want to do analyses by), the function `levels()` can be useful. Explore this function.

3 Exploring Your Data

Check to see that your `mydata` is still in memory using the `ls()` function. If not, re-read it into memory. Then type `summary(mydata)` to get a basic overview of the descriptive statistics of your data.

You'll have to be able to manipulate your data set at some point. This is primarily done using the `sort()` and `order()` functions. Make sure you know the difference between them! Typically, `sort()` will organize according to your specifications *independently* of the other columns, which you may not want to do (be thankful R doesn't rewrite your original .csv file!), and is therefore in my opinion more appropriate for arranging stand alone vectors (see Section 4) as opposed to data frames or matrices (see Section 6). Lets try a simple example:

```
> mydata[order(mydata$dbh), 1:ncol(mydata)]
```

This tells R to 'sort' the data frame by increasing values of 'dbh' and to order **all** columns in this fashion. The part after the comma tells R which columns to return (I used the `ncol` function to extract the number of columns of the data frame). Alternatively you could simply do for all columns:

```
> mydata[order(mydata$dbh),]
```

The default for `order()` is increasing; add `rev(order())` to arrange the data frame according to *decreasing* values of the desired column.

[7]Note: it's best to avoid using the following: c, q, t, C, D, F, I, & T. These are already reserved for other meanings in R (T & F = TRUE & FALSE). Its also a good idea to avoid l and O (lower-case 'L' & upper-case 'O'), because they are easily confused with 1 (one) & 0 (zero).

Table 2: List of the main logical operators used by R.

`x < y`	less than
`x > y`	greater than
`x <= y`	less than or equal to
`x >= y`	greater than or equal to
`x == y`	equal to
`x != y`	not equal to
`&`	AND
`&&`	AND with IF
`|`	OR
`||`	OR with IF

Exercises

1. Now your turn; use the `order()` function to re-sort `mydata` according to *increasing* bark thickness, then decreasing sapwood area (be sure that the neighboring columns are also affected!). If you want the changes to be stored as new objects in memory, name it first (e.g. `mydata2 = mydata[order(...)`. Also try sorting by the column 'spp', which by now you know to be a factor, what happens?

2. You can also sort by multiple variables/factors. Sort `mydata` first by 'species' alphabetically and then by decreasing 'area without bark'. **Hint**: no need now for `rev()`.

3. Now create a new data frame called 'tree1' which contains only tree #, SapDepth, & SapArea and is sorted by decreasing SapDepth. I suggest you play with this a bit as you will be doing much of this with your own data and the data sets we will be using during this workshop. There is a function called `subset()` that can simplify this process, but since this function does not do any sorting, you'll have to do this in two steps. If you have time you should explore this way too. How would you make a data frame containing only those trees measured in Winter?

4. You can also exclude certain cases from the data frame that do not satisfy a given set of arguments. For example:

   ```
   > mydata[mydata$BarkThick != 0.0, ]
   ```
   [8]

   Whereby you're asking R to return all rows (notice the comma) where the column 'BarkThick' is **not** equal to zero. These logical arguments can also

[8]For this command and all that follow throughout this tutorial, if you attached your data using `attach(mydata)` you do **not** need to refer to objects from the data frame using this construction. Nevertheless, I would steer clear of `attach()`.

be used in combination with others using the '&' symbol. For example, limit the data frame to only those trees which have a sapwood area $\leq$ 100 cm^2 *and* have a sapwood depth ≥ 2 (there should be 7). If you think you'll be using this 'new' data set, simply give it a new name and work with that new data frame. Other operators for comparisons & selecting data are in Table 2.

5. No preliminary exploration is complete without taking a look at some graphs (psst: advisors *love* graphs), so lets take a look at the usual suspects in data exploration.

   - The Histogram (the overall distribution of your data)
   - The Boxplot (discontinuous data; relationships)
   - The Scatterplot (continuous data; relationships)
   - The xyplot (scatterplots separated by e.g. sex, season, etc.)
   - The Barplot (data displayed by category as bars)

   Here is the R code for Fig. 1 (ensure that `mydata` is still in memory!):

```
> hist(mydata$SapDepth, xlab= "Sapwood Depth", main="Histogram of
+     Sapwood Depth", col= "gray50")
> boxplot(SapDepth ~ spp, data= mydata, ylab= "SapDepth", col=
+     "darkslateblue")
> plot(mydata$dbh, mydata$Heartwood, pch= 17, col= "darkred", ylab=
+      "Area (cm^2)", xlab= "DBH (cm)", main= "DBH vs. 2 tree
+      characteristics")
> points(mydata$dbh, mydata$SapArea, pch= 19, col= "darkgreen")
> legend("topleft", legend=c("Heardwood", "Sapwood"), pch= c(17,19),
+      col =c("darkred", "darkgreen"), bg= "gray95")
```

   Fig. 1 shows examples of various types of exploratory plots. Use the code above to replicate it[9]; you'll need to preface this code with `par(mfrow=c(1,3))` which tells R to create a plotting window array of 1 row $\times$ 3 col and place whatever you plot next into that graphics array. Look at `mydata` and explore some relationships for yourself graphically (e.g. do species differ in sapwood depth?). Upon visual inspection of Fig. 1b it looks like they do. To confirm this, you would then have to show if this difference is statistically significant (see Section 11). From the scatterplot it also appears there is a relationship between dbh and the two chosen tree characteristics. A correlation or linear regression analysis would bear this out statistically (see again Section 11).

   Lastly, the `xyplot()` function produces a number of scatterplots separated into panels by a given factor. This can be extremely useful for exploring

[9]You may have to remove the '+' and '>' symbols if you decide to cut & paste this code.

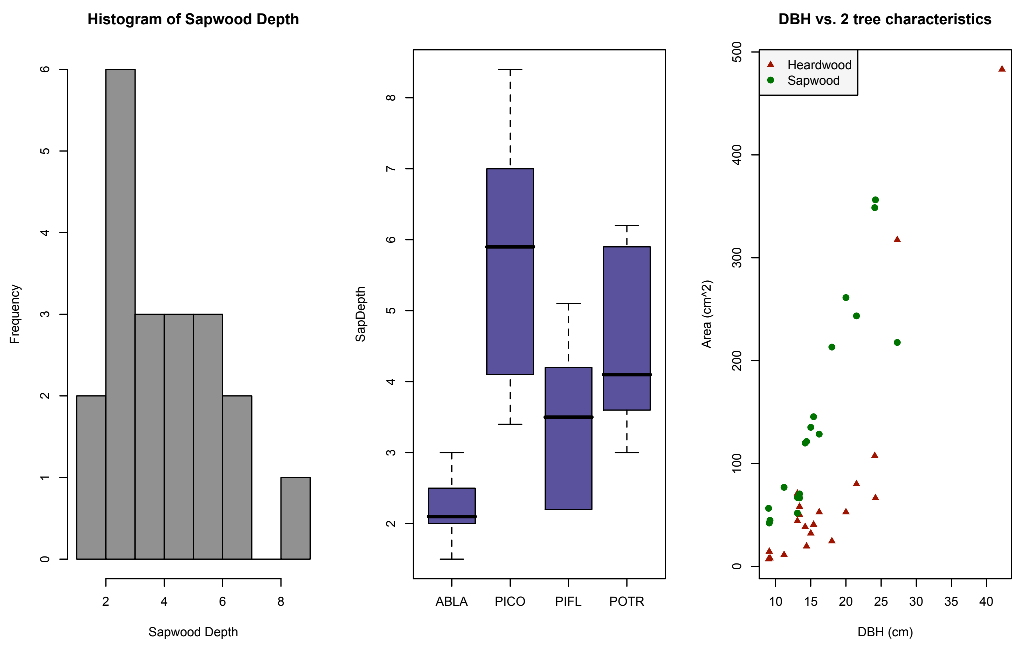

Figure 1: Quick & dirty visual exploration of data.

conditional relationships visually. Lets take a look at sapwood depth vs tree diameter by species (see Fig. 2). What you are telling R in the script below is: produce a scatterplot of `SapDepth` vs. `dbh`, separated by the grouping factor 'species', and use the data frame `mydata` as the source for this plot. The rest of the arguments simply define the visual presentation of the xyplot. By default R place the panels from bottom left → to top right, which has always seemed odd to me, so the'as.table' argument reorders them. Try it out for yourself with different dependent and independent variables. What happens when you type 'spp * Infected' instead of just 'spp' as your grouping factor? You will need the 'lattice' package for xyplots.

```
> require(lattice)
> xyplot(SapDepth ~ dbh | spp, data= mydata, xlab= "DBH
+     (cm)", ylab= "Sapwood Depth", rows= 2, pch= 19, col= "navy",
+     as.table= TRUE)
```

The function `coplot()` produces similar graphics output to `xyplot()`, however it may be more appropriate for multivariate data, especially where the 'organizing' variable (spp in Fig. 2) is continuous (e.g. temperature, altitude, etc.). Experiment with `coplot()` on the above instead of `xyplot()` and compare.

6. Make a Barplot of the mean *Area Without Bark* first by **season** & then by **species**. The conclusions from these results are meaningless but you'll soon

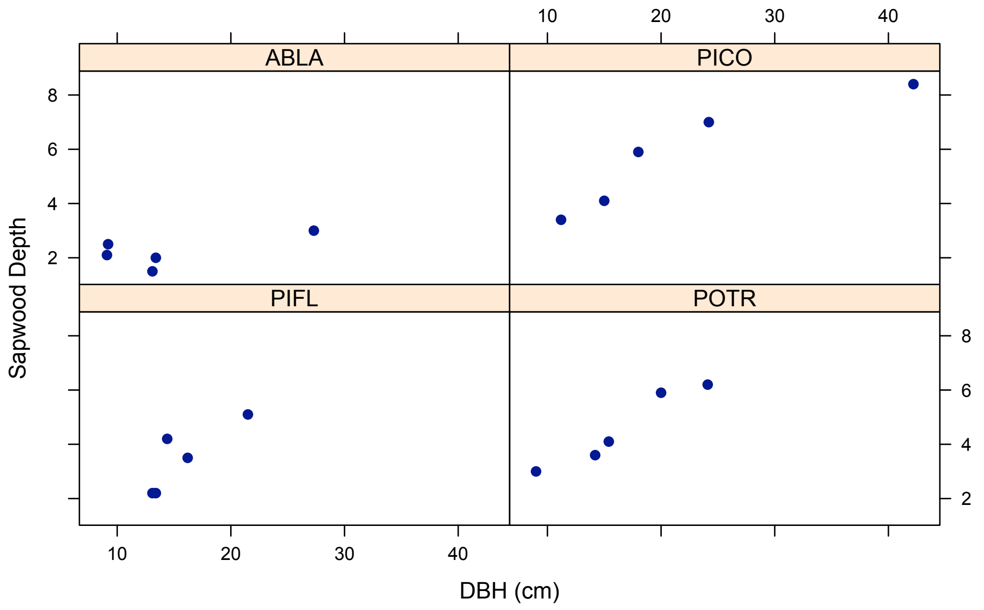

Figure 2: More quick & dirty visual exploration of data.

have a fairly complete script for making nice R plots. I will start you off with the season, can you follow what is happening at each step? `tapply()`[10] is a extremely useful function (the 't' stands for table) that applies a function to the values in a vector to produce a table based on one or more grouping variables (e.g. factors) & showing the result of the function in each. This sounds complicated but it really isn't. Try typing `tapply(mydata$dbh, mydata$season, sd)`. What did R return? All I'm doing below is telling R to take that table and make it a vector instead. Now complete the Barplot exercise for season, and then repeat for species or another dependent variable.

```
> Nobark.season = as.vector(tapply(mydata$NobarkArea,
+     mydata$season, mean))
> x.axis = levels(mydata$season)
> barplot(Nobark.season, names= x.axis, ylim= c(0,max(Nobark.season)))
```

Notice that when defining the vector containing means, they are combined by `tapply()` from left to right *alphabetically*. This is good because in the next step I define the x-axis with the `levels()` function, which also arranges its levels in alphabetically. Otherwise it would decouple the mean value with the season to which it corresponds! Pay attention to these kinds of operations, always double check against known values (do they look right?) and always scrutinize the graph (does it meet your expectations?).

[10]Similar to `apply()` from above, but can be applied to 1D objects.

4 The Vector

A vector is basically just a list of numbers (i.e. a $1 \times n$ matrix) and is one of the most often manipulated objects in R. They are used for everything from calculating means, plotting graphs, storing outputs of functions (aka subroutines), or simply storing parameters for later use within a larger program. It is essential to learn to master their manipulation. Some of the most basic and useful functions for manipulating vectors are in Table 3. **Remember:** operations & functions are generally applied to vectors in R in an *element-by-element* basis.

Table 3: A few of useful commands for creating and modifying vectors in R.

`1:x`	Vector from 1 to x by increments of 1.
`seq(from,to,by=x)`	Vector values in increments of x (if $x < 0$ = decreasing)
`seq(from,to,length=x)`	Vector of evenly spaced values of x length
`c(u,v,...)`	Combine numbers and/or vectors into a single vector (from left $\rightarrow$ right)
`rep(a, b)`	Creates vector of repeating a elements by amounts of b
`abs(x)`	Absolute value of x
`cos(x),sin(),tan()`	Cosine, sine, tangent of angle x in radians
`exp(x)`	Exponential function, e^x
`log(x)`	Natural logarithm of x (log_e)
`log10(x)`	Common logarithm of x (log_{10})
`sqrt(x)`	Square root of x
`head(x,n)`	Returns the first n entries of vector x
`tail(x,n)`	Returns the last n entries of vector x
`round(x,n)`	Round entries of x to n decimal places

Exercises

1. Create a vector with the numbers: `[2, 5, 8, 2, 1, 9, 4, 7, 5, 6]`.
2. Create a vector with the numbers: `[2, 2, 2, 5, 5, 9, 9, 8, 8, 8, 8]`. Try to use the `rep()` command.
3. Create a vector that ranges from 0 to 6.5 by increments of 0.5. How many entries are in this vector? (I don't want to see anyone pointing at the screen and counting!).
4. Understand why `rep(1:5, 1:5)` and `rep(1:6, c(1, 2, 3, 3, 2, 1))` produce what they produce.
5. Create a vector called 'all.vecs' that includes all the numbers you just created in one long vector (you may choose the order).

6. Create a vector of 15 random numbers from 10 $\rightarrow$ 75 and name it 'mass'.[11]

7. Assume this is the yearly increase in mass for some treatment of interest, create a new vector called 'mass.d' representing the daily weight gain. Create a new vector called 'mass.r' with these values rounded to two decimal places. What is the average daily weight gain for the treatment?

8. As I said in Section 3, I think `sort()` is more appropriate for stand-alone vectors, so try using it on 'mass' (both increasing & decreasing).

9. Create a new vector of length 7 with the minimum value, maximum value, mean, sum, median, and range of `'mass.r'`.

10. Use this new vector in multiplying the mean daily weight gain by the total weight gain, then divided by its median. I don't know why you would want to do this, but you get the point; you can use vectors to make calculations and then use those entries to make other calculations.

11. Now lets use the imported tree data to do something. The process is similar to above. Assume individual leaf area is a function of numerous other tree characteristics and can be calculated as follows:

$$Leaf\ area = \left(0.1 \cdot \delta^{(1-\beta)}\right)\ +\ \sqrt{\frac{\tau+1}{\sigma}} \tag{1}$$

    where $\delta, \beta, \tau,$ and σ are tree diameter, bark thickness, total no bark area, & sapwood area respectively. This would be similar to writing an equation in Excel and then copying it down the column. **Note**: you could use either `sqrt()` or ^ $(\frac{1}{2})$, but for cube roots or higher, you must use the 'fractionated' syntax.

[11]See Table 4 for more information about random number generation.

5 Generating Random Numbers

R provides great flexibility & options for generating random sequences of numbers from various distributions. Table 4 shows the major functions used to produce numbers according to these distributions.

Table 4: Random number generation in R from various distributions.

`runif(n,min,max)`	Random n numbers from uniform distribution
`rnorm(n,mean,sd)`	Random n numbers from a Normal distribution with given mean & sd
`rpois(n,lambda)`	Random n numbers of Poisson distribution
`rbinom(n,size,p)`	Random n numbers from Binomial distribution; size = # trials; p = probability of 'successful' trial
`rnbinom(n,size,mu)`	Same as above except *negative* Binomial distribution (mu = alternative parameterization via the mean)
`rgamma(n,shape,scale)`	Random n numbers from Gamma distribution

Of course you can also determine the probability density, distribution, & quantile functions for each of these distributions using either 'd', 'p', or 'q' instead of 'r'. For example, `dbinom`, `pbinom`, `qbinom` will produce the probability density, distribution, and quantile functions respectively from a Binomial distribution for a given set of parameters.

Exercises

1. It is important to get some experience 'playing' with these functions, especially what they look like and what they're used for. Look at Fig. 3 which shows what each of these functions do. Try to reproduce it using the `curve()` function.[12] Below is the code for the first plot to get you started:

```
> par(mfrow=c(2,2))
> curve(dnorm, -4, 4, xlab= "z", ylab= "Probability Density",
+     main= "Density", col= "darkgreen", lwd=2)
```

2. What does the following R code (below) produce and why? Can you predict what the plot will display before you run it? It is often convenient when generating random numbers to set the *seed* using `set.seed()` so that you can ensure that the *same* random numbers are generated each time. Doesn't that make it *non*-random? Yes, but it's good for 'debugging' and code verification (and making sure students get the same result you do!). The actual seed can be any arbitrary integer (666 produced a nice histogram). Can you produce

[12] This function is first introduced in Section 8. Look ahead for clues.

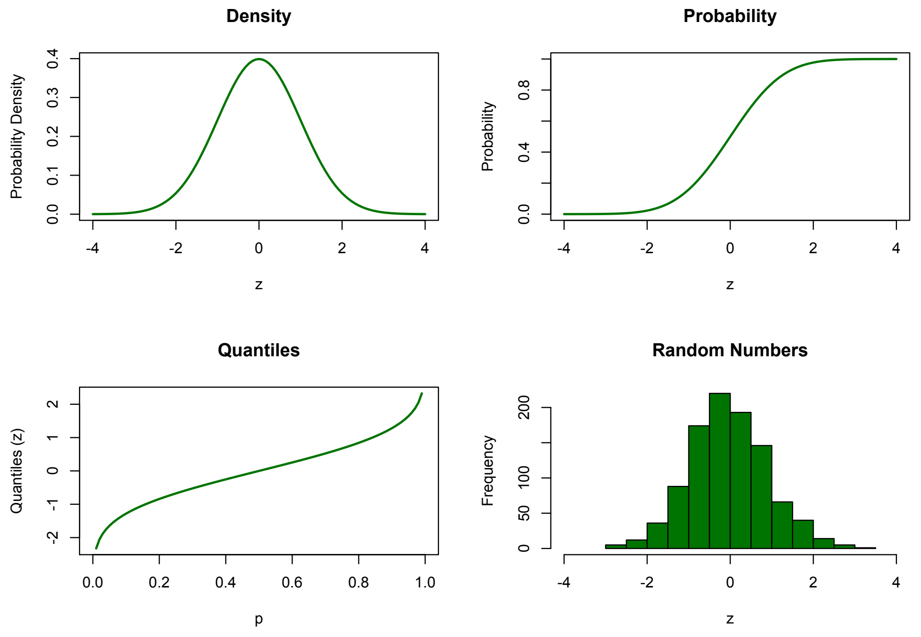

Figure 3: Reproduced shamelessly from *The R Book* (p. 219).

something similar, but with sapwood depth as in Fig. 1a. Try it for a Binomial distribution that you define yourself.

```
> set.seed(666)
> nD = rnorm(1000, 400, 25); L= length(nD)
> ks.test(nD, pnorm)      # what does this line do?
> hist(nD,col= "gray88", main="", xlab="nD", ylab= "Frequency")
> lines(seq(0,L,1), 10*L*(dnorm(seq(0,L,1), mean(nD), sd(nD))),
+    col="navy", lwd=2)
```

6 The Matrix

Besides evoking images of bullets flying in 3D surround slo-mo, the matrix (or array) can be put to good use in R. Aside from their obvious use in population projections and classical matrix algebra (be careful! R does its operations in such a way that mathematicians would pull their hair out), if nothing else they are very useful to store output of a given program/function, which is then to be used to some other purpose (e.g. plotting).

Table 5: A few important functions for working with matrices in R.

`matrix(v,m,n)`	$m \times n$ matrix using the values in v
`nrow(),ncol()`	Number of rows and columns of a data frame or matrix
`t(A)`	Transpose (exchange rows & columns) of matrix **A**
`dim(A)`	Dimensions of matrix **A**. (dim(A)[1]=rows, dim(A)[2]=cols)
`edit(A)`	Call up 'spreadsheet' interface to edit the values in **A**
`diag(v,n)`	A diagonal $n \times n$ matrix with v on diagonal, 0 elsewhere (by default v = 1, so `diag(n)` gives an $n \times n$ identity matrix; if v = vector it is placed on the diagonal & n = length(v))
`cbind(a,b,c,...)`	Combine **compatible** objects by attaching them by columns (e.g. a data frame with only numeric data)
`rbind(a,b,c,...)`	Same as `cbind` but attaching them by rows (can also bind multiple matrices in this manner if dimensions match)
`as.matrix(x)`	Convert object of another type to a matrix (if possible)
`outer(v,w)`	'Outer product' of vectors v, w: the matrix whose $(i,j)^{th}$ element is `v[i]*w[j]`
`A[,n]`	Returns a vector of column n
`A[m,]`	Returns a vector of row m
`A + B`	Matrix addition (or - for subtraction); entrywise
`sum(A)`	Sum of all elements of matrix **A**
`A %*% B`	Matrix multiplication ($A \times B$); (or %/% for division)
`A * B`	Entrywise multiplication ($A \times B$); the Hadamard product
`det(A)`	The determinant of matrix **A**
`eigen(A)`	The eigenvalues & right eigenvectors of matrix **A**

Exercises

1. Use the construction `matrix(v, row, col)` to create the matrix in Equation (2) where v is a data vector created using the `seq()` function from Table 3. You will probably need to modify the function with the argument `byrow=TRUE`.

$$\mathbf{Mat} = \begin{pmatrix} 16 & 12 & 8 \\ 4 & 0 & -4 \\ -8 & -12 & -16 \end{pmatrix} \tag{2}$$

2. Create the matrix **A** below in Equation (3). You can do this either by using the `c()` command and typing the entries individually or first creating a vector with the entries you want and then putting that vector as the first argument in the `matrix()` function. Try it both ways.

$$\mathbf{A} = \begin{pmatrix} 2 & 3 & 7 & 8.8 & 11 \\ 3 & 4.3 & 8 & 9 & 12 \\ 8 & 16 & 0.1 & 5 & 9 \\ 5 & 0.4 & 9 & 1.7 & 3 \\ 0.7 & 6 & 5.9 & 4 & 7 \end{pmatrix} \tag{3}$$

3. Use `dim()` to determine the dimensions of **A**. The output is a two column vector (rows, columns), which is the standard for R (rows first, then columns), however this example is perhaps not ideal as nrows = ncols (5×5). Alternatively, you may use `nrow()` or `ncol` instead of `dim()[1]` and `dim()[2]` respectively. These commands are especially useful when writing functions (see Section 9).

4. Use the `rowSums()` command to create a vector that contains the totals of each row of **A**. Now do the same for the columns, can you guess what the command is called? It might be useful to give these vectors a name. Now try using the `apply()` function from Section 1 to do the same thing.

5. Now use the commands `cbind()` & `rbind()` to attach these two 'summed' vectors to create a new matrix **B** using the tools above which looks like Equation (4).

$$\mathbf{B} = \begin{pmatrix} 2 & 3 & 7 & 8.8 & 11 & 31.8 \\ 3 & 4.3 & 8 & 9 & 12 & 36.3 \\ 8 & 16 & 0.1 & 5 & 9 & 38.1 \\ 5 & 0.4 & 9 & 1.7 & 3 & 19.1 \\ 0.7 & 6 & 5.9 & 4 & 7 & 23.6 \\ 18.7 & 29.7 & 30.0 & 28.5 & 42 & 148.9 \end{pmatrix} \tag{4}$$

6. Add the vector you created earlier called 'Leaf area' to `mydata` as a column in a similar fashion as above.

7. It's sometimes necessary to export your data frame or matrix. Use the `write.csv()` command to do this (it will go to your home directory). Try exporting matrix

B to a `.csv` file called '`MatrixSum.csv`'.[13] If you get stuck, don't forget about `?write.csv`.

8. Matrix population models use projection matrices to estimate future populations using a set of difference equations which are condensed into a matrix which contains the vital rates of various stages/classes. The simplest of these models takes the form:

$$\vec{x}_{(n+1)} = \mathbf{A} \cdot \vec{x}_{(n)} \tag{5}$$

where $\vec{x}$ is a vector containing the number of individuals in a given age, stage, or class and **A** is the projection matrix. Consider the following projection matrix **M** which contains 3 classes. The diagonal represents the survival rates of the classes, the sub-diagonal (0.1 & 0.3) represents the transition probability (i.e. class 1 → 2 and class 2 → 3) and the top row represents the fecundity of each class (the number of class 1 individuals produced per individuals of class 2 or 3; class 1 does not reproduce).

$$\mathbf{M} = \begin{pmatrix} s_1 & f_2 & f_3 \\ t_1 & s_2 & 0 \\ 0 & t_2 & s_3 \end{pmatrix} = \begin{pmatrix} 0.3 & 1.0 & 3.0 \\ 0.1 & 0.4 & 0.0 \\ 0.0 & 0.3 & 0.8 \end{pmatrix} \tag{6}$$

Assuming $\vec{x}_1 = [1 \;\; 2 \;\; 4]$, calculate $\vec{x}_2$. Be careful, remember how R's matrix multiplication works (you'll need `%*%`). We'll come back to Equation (6) later.

7 The Loop

Loops are essential in any programming language, master them and you're well on your way. There are two basic loops in R, the *for*-loop and the *while*-loop. The *for*-loop runs through the loop x number of iterations, then exits the loop and continues to the rest of the program. The *while*-loop continues looping until some predetermined criteria (*condition*) is met (e.g. as long as x is $\leq$ then 1000, continue on looping). In *while*-loops it is often a good idea to place a 'counter' within the loop so that later you can determine how many iterations were required to exit the loop. In this sense, *for*-loops could be considered a subset of *while*-loops (if the counter were used in the condition statement). In addition, your variable in defining the condition must appear somewhere within the loop and must change as the loop iterates (otherwise you'll enter the loop and never get out!). Below is a simple example of a simple *for*-loop:

[13]If you do export a modified data frame, be sure to clearly rename the 'new' modified data set. It's potentially disastrous to re-enter your data (so be careful). It's usually possible, and preferable, to reshape/manipulate your data *within* the R environment using commands at the beginning of your script.

```
> Popn = c(1, rep(0,9))}  ### set up a vector to store results
> Gen = length(Popn)
> for (n in 2:Gen) {
+     Popn[n]= Popn[n - 1] * 2
+ }
```

Call 'Popn', this should look very familiar, the population seems to exhibit exponential growth. **Note** the values over which you are asking R to iterate (2:Gen). Have you seen this construction before? What do you get when you enter `2:10` in the console? What this means is that it is possible to skip iterations by using something like: `for (i in seq(2,10,2))`.

Here is a similar example to the above using a *while*-loop:

```
> pop.vec = pop.now = 1; count = 1
> while (pop.now <= 25000) {
+     pop.now = pop.now * 2
+     pop.vec = c(pop.vec, pop.now)
+     count = count + 1
+ }
```

So essentially what you're telling R is that it should start with the current population of 1, the storage vector that will contain the population trajectory is also 1 (since it hasn't started yet), and I've added a counter so that following the simulation I will be able to know how many iterations were required to satisfy the condition that the population be $\leq$ 25000. Notice that I've used the `c()` function to 'combine' the previous population sizes with the current population size; R is essentially adding a new number (pop.now) to an ever growing population vector (pop.vec), until the population is more than 25000. Now call 'pop.vec' and 'n' to see the simple projection and how many iterations it took to do so. Now, can you modify this code to replicate the classic expression, "take a penny, double it every day for a month, and you'll be a millionaire"? Is this true? How long does it take to become a millionaire?

Remember, you can also do loops within loops (aka *nested* loops), which is quite common but no more complicated than regular sequential loops, just keep track of what you're doing as they can get messy. Also, people tend to use '*n*' or '*i*' as counters in *for*-loops (I don't know why), but realize that it could be any letter or combination of letters you want.

Exercises

1. Try adding some 'noise' or stochasticity to the first population growth model (the *for*-loop) using some random number manipulation, then roughly plot the results. We'll do more intense plotting in Section 8. For now try:

```
> plot(1:Gen, Popn, ‘b’).
```

Alternatively, you could add noise to the data using the `jitter()` function.[14]

2. Use a loop to create a vector representing a Fibonacci sequence running from $0 \rightarrow 610$. You will need to start with the vector = [0,1]. Then:

   (a) Use this vector to create the matrix below:

$$\textbf{Fibonacci \#1} = \begin{pmatrix} 0 & 3 & 21 & 144 \\ 1 & 5 & 34 & 233 \\ 1 & 8 & 55 & 377 \\ 2 & 13 & 89 & 610 \end{pmatrix}$$

   (b) Use the `sample()` function to create a matrix of random Fibonacci numbers (use `?sample` to determine the proper syntax if necessary). For example:

$$\textbf{Fibonacci \#2} = \begin{pmatrix} 144 & 2 & 89 & 8 \\ 1 & 377 & 610 & 5 \\ 21 & 34 & 233 & 1 \\ 3 & 55 & 0 & 13 \end{pmatrix}$$

3. Write a script using a *nested for*-loop which produces the following 5×5 projection matrix:

$$\textbf{PMat} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 0.1 & 0 & 0 & 0 & 0 \\ 0 & 0.2 & 0 & 0 & 0 \\ 0 & 0 & 0.3 & 0 & 0 \\ 0 & 0 & 0 & 0.4 & 0 \end{pmatrix}$$

4. Use a loop and Equation (6) to project the 3 class population described above for 20 generations (i.e. $\vec{x}_{1\rightarrow 20}$). You will need to use a secondary matrix to store the results of the iterations (and then to plot them in Section 8). You should use Exercise 8 in Section 6 to guide you.

[14]See Section 8 for more on `jitter()`.

8 Plots & Graphs

Remember what I said in Section 3 about advisors and graphs. That aside, plots are an excellent way to summarize various data visually and are essential if you're ever going to publish your data. Luckily, R makes great high-quality plots of all types, colors, and varieties. I also recommend you figure out how to save your plots as a pdf so you can send them to your advisor ☺.

Table 6: List of numeric colors used by R.

No.	Color
1	Black
2	Red
3	Green
4	Blue
5	Cyan
6	Magenta
7	Yellow
8	Gray
9	Black

Exercises

1. Boxplots can be a particularly good way of presenting data & exploring relationships without losing a lot of important information about the distribution of the data (especially for continuous data; I don't like barplots for this reason). Take a look at Fig. 1b and reproduce it for another variable in the data set. Next, say you want to explore the possibility of a relationship between season, infection and some other continuous variable (perhaps trees become infected at a particular time of year as a function of this variable?). Make a doubly grouped boxplot to explore this possibility.

2. Symbols plots are also a great way to add information about specific data points to a basic plot. The `symbols()` function is great for this, and is especially good for geographical data where x & y are grid coordinates. Read the data file 'habsel.csv' into R and make a geographical plot of captures for these data (x & y are coordinates) with blue solid circles & red edges.

3. Now lets produce some fake data of a typical textbook outbreak of personal interest, the scourge that I call '*Uggs with Shorts*'.[15] Lets assume that an individual becomes infected on the date of purchase and individuals are infected for life(!).

[15]seriously, if you do this you're hurting society, please stop. Even the Europeans are laughing at us.

```
> t = 1:30
> set.seed(125)
> infected= jitter(0.4/(1+exp(0.4*(15-t))), amount= 0.035)
> plot(t, infected, ylim=c(0,0.5), main= "Proportion Infected
+     by Uggs with Shorts", xlab= "Months since Lauren first did
+     it on The Hills", ylab= "Proportion Infected")
```

Now plot these **crucial** data, you should get something like Fig. 4a. The `jitter()` function adds random 'noise' to the y-variable (that's why the proportion can go negative – just ignore that).

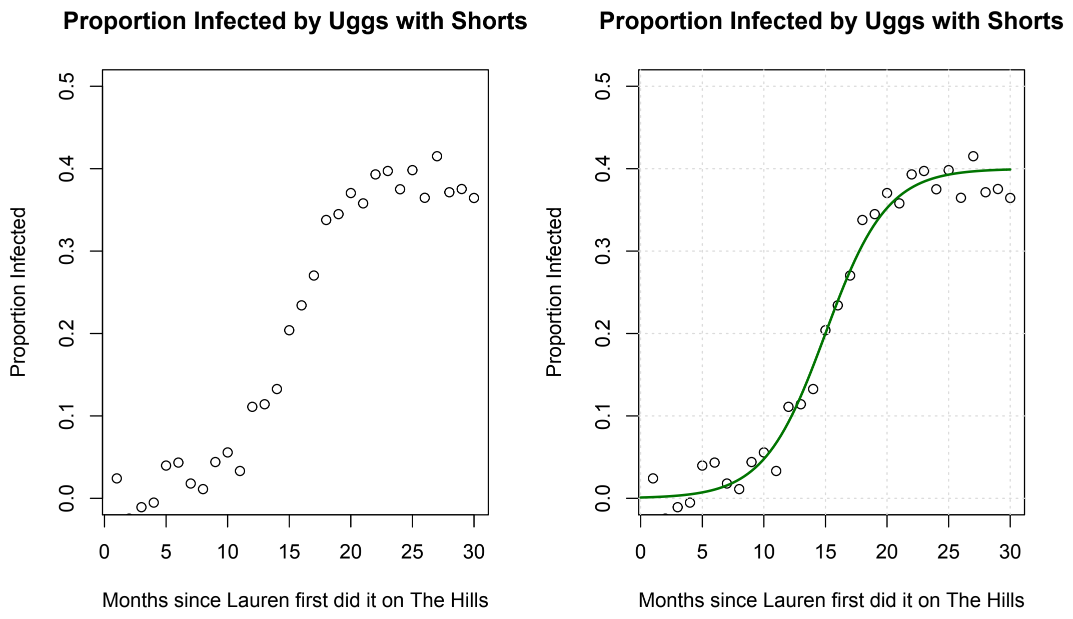

Figure 4: The scourge that could end civilization: *Uggs with shorts.*

4. Plot the population projection you produced in the final exercise of Section 7. You should get something like Fig. 5. Play with the arguments to get it to look like Fig. 5, including the legend.

5. Matplot is a very good command for quickly graphing data multiple lines in the same plotting window. Your data frame must be arranged in a particular format however. Each variable to be plotted must be arranged by column and must have the same independent variable (e.g. time). Read `Pop6C.csv'` into memory and use `matplot()` to produce Fig. 6 (don't forget the grid; is it on top of the legend?).

6. The next essential function for your `R` toolbox is `curve()`, which is a *magical* function that takes a mathematical function/expression and plots over a

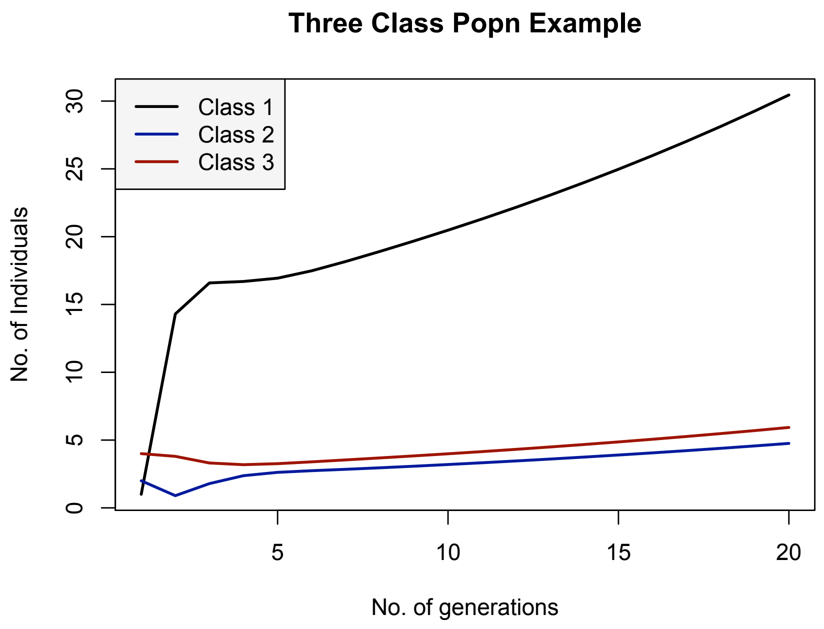

Figure 5: Quick population projection with 3 Classes.

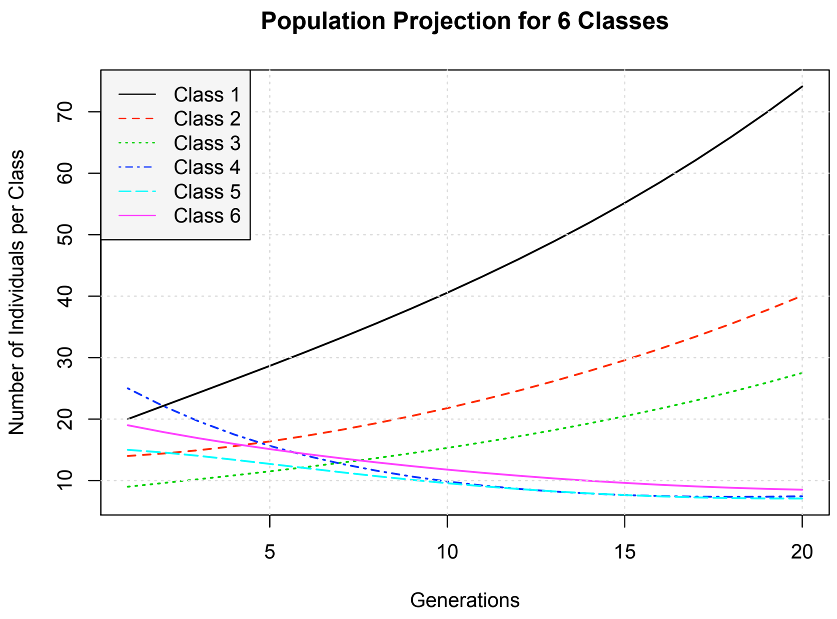

Figure 6: Example of using matplot to quickly graph simultaneous variables with the same independent variable.

given range of x. Most of the arguments are identical to the `plot()` function so you don't have to learn many new ones. Graph the following functions with `curve()` (to start try from=–10, to=10) and finally use the `add= TRUE` argument to create the graph in Fig. 4b.

$$\begin{aligned} f(x) &= \frac{1}{1+x^2} \\ f(x) &= 2x^3 - 8x^2 + 2x + 6 \\ f(x) &= \frac{0.4}{(1 + exp^{(0.4*(15-x))})} \end{aligned}$$

7. Yet another useful plotting function is `abline(a, b)`. This nifty function adds a straight line with intercept = a & slope = b to an existing plot (I learned **m** = slope & **b** = intercept, so now I find it confusing to have an 'a' here & for b = slope; so just be careful). `bmline()` would make more sense, but that's just me ☺. Familiarize yourself with the following:

   - `abline(5, 0.66)`
   - `abline(h = 4)`
   - `abline(v = 2)`

   You must add the line to an existing plot, but all the familiar arguments for plotting lines apply here, you can change its thickness, colour, type, etc., just as you would any other line using `lines()` or `plot()`. We'll be using `abline()` again later in Section 11.

9 Functions

Writing your own functions (aka subroutines) in R can be an extremely useful tool/strategy, especially for repeated calculations. Let's start simple with a generalized function format:

```
> myfun <- function(x, y, z) { ### 'fun' = function, not fun ;)
+     expression 1
+     expression 2
+     expression n
+     Output, return(), or list()
+ }
```

In this generic example, x, y, & z are the arguments for the function and will have to be provided by the user (you). They are essentially the raw material the function will need to perform its 'function' (no pun intended) and can be vectors, scalars, or even matrices. It is also possible to give the function a default value for an argument by writing `function(x, y=5, z)` for example. If no value for 'y' is provided, the function will use 5.

The expressions within the function will describe what `myfun` will do to x, y, & z in order to produce what you want as an output. The last line usually ends with what you want the function to spit out when you hit `<enter>` (this must be described within the expressions). If there is more than one thing you want as output, use `list()`. The function `return()` can be especially useful for returning something from the middle of a function (if necessary). In addition, an output line is not absolutely *necessary*, only if you want R to return something when you hit `<enter>`. Lastly, notice I use '`<-`' instead of '=', you could use either but out of habit I use the arrows for defining functions so they stand out.

Lets try a very simple example. Below is a function that already exists in R, the `mean()` function, which I will emulate so you can double check it with the 'real' function.

```
> xbar <- function(x) {
+     mu <- sum(x)/length(x)
+     mu
+ }
```

So, now to use the function you just created, simply type `xbar(vector)` where `vector` is a predetermined list of numbers already in memory. Type `ls()` and find a vector that's in memory and try it out, then double check it with the `mean(vector)` function provided with R. Also note that in the list of objects in memory, your new function `xbar` is now present. Lastly, it is important to remember that the variable names you choose in a function's arguments and expressions are *not* saved as objects in R's global memory and are used only within the function *locally* to tell R how to manipulate the arguments you've given it. As a result, they can be used elsewhere in your program (i.e. in the above, mu, x, y, & z can already exist as objects, which is good because in a long program you may start running out of obvious abbreviations for your variables). This also means, however, that you *cannot* reassign the value of a variable with a function.

Ok, one last example and then you're on your own. Say for some reason you wanted a function that created a matrix of random numbers. There are numerous ways to do this, here is one:

```
> rMat <- function(m, n, min, max, dec=0) {
+     random <- runif(m*n, min, max)
+     A <- matrix(random, m, n)
+     round(A, dec)
+ }
```

Line one tells R that the new `rMat()` is going to be a function and defines the arguments that will be required for that function to work. The arguments 'm' and 'n' are the dimensions of the matrix (rows, cols), 'min' and 'max' define the range of numbers the matrix should contain, and 'dec' is the number of decimal places the numbers within the matrix should be rounded to (default set to zero decimal places). The second line creates a vector of uniform random numbers of the desired range using the `runif()` function already provided with R. Line three creates a matrix called 'A' from those random numbers and of the desired dimensions. The final line tells R to spit out 'A' with its elements rounded to the desired number of places. Play around with `rMat()` until you feel you're comfortable with the syntax of writing functions, then move on to the exercises.

Exercises

1. Below is a function that produces the $\bar{x} \pm 95\%$ confidence intervals (assumes a normal distribution). It uses Equation (7) to calculate the standard error of the mean, and returns the upper and lower CI95s along with the mean of vector (x) provided as an argument.

$$SE_m = \frac{sd}{\sqrt{N}} \tag{7}$$

```
> CI95 <- function(x){
+       mean <- mean(x)
+       sd <- sd(x)
+       n <- length(x)
+       se <- sd/sqrt(n)
+       CI.vec <- c((mean-(1.96*se)), mean, (mean+(1.96*se)))
+       CI.vec
+ }
```

   Explore this function for a few minutes then use it on a few of the vectors in Section 4 or on any of the variables in `mydata` (e.g. SapArea). Remember, the vector does not have to be named 'x', it can have any name since x above is only stored locally *within* the function itself.

2. Write a function that, given a vector of length=3, containing the numbers of individuals of each genotype [AA, Aa, aa], will produce the allelic frequencies of the population (lets assume no sampling error). These will be your good 'ol p's & q's from Hardy & the Weinberg.

3. Write a function that incorporates the steps you used above in Equation (4) to produce matrix **B**, given an initial matrix **A** (i.e. given a matrix, create a new matrix with an extra column and row which contains the row and column sums of the original matrix **A**).

4. Look at the function below. What does it do? Can you predict its output? First execute the function with its default arguments (`NormFun()`), then explore arguments of your choosing. Finally modify the core of the function in some way in areas of your own interest (binomial?).

```
> NormFun <- function(n=1000, mu=400, sdv=25, seed=666) {
+     set.seed(seed)
+     nD <- rnorm(n, mu, sdv)
+     hist(nD, col="gray88", main="", xlab=""); par(new=T)
+     plot(min(nD):max(nD), dnorm(min(nD):max(nD), mean(nD),
+        sd(nD)), "l", col="navy", lwd=2, axes=F, xlab="",
+        ylab="", bty="n")
+ }
```

Last note on functions: it is common and often useful to use a function as an argument in yet another function (a function *within* a function), analogous to *nested*-loops in Section 7. You will be doing this below in Section 10.

10 Ordinary Differential Equations (ODEs)

Ordinary differential equations (ODEs) govern how one dependent variable (population size or number of infectious individuals) changes relative to changes in some other independent variable (e.g. time). ODEs can be used to mathematically describe mechanistic, biological relationships between dependent and independent variables and to project this relationship in time in order to understand the dynamics of the dependent variables. Based on this mechanistic relationship, once you have the determined equations, the next step is to investigate the dynamics of your model (e.g. changes over time) for a given set of parameter values. This section will show you how to do just than in R. First, you will need to load the 'odesolve' package since we will be relying heavily upon `lsoda()` in this section. This function is the workhorse for ODEs and is peculiar in that it, as mentioned briefly in Section 9, takes a function as one of its arguments.

- `lsoda`'s main arguments are the starting values (`y`), the times at which you want to compute the values of the variables you are interested in (`times`), a derivative function (`func`), and some parameters (`parms`).
- `func` must take as its *first three* arguments the current time (t), the current values of the variables (y), and a vector containing the parameter values. It must also return a list (using `list(item1, item2, item3)`, where the items can be any R objects) whose elements are a vector of ODEs (see code below).

Here are two examples to get you started:

Examples

1. Logistic growth[16]

We will start with logistic growth partly because it has only one dependent variable (& one differential equation to solve) and partly because most of you are familiar with this type of growth. The model equation that describes how the population size changes over time ($\frac{dN}{dt}$) for this kind of system at its simplest looks like this:

$$\frac{dN}{dt} = rN\left(1 - \frac{N}{K}\right) \tag{8}$$

where N is the population size (dependent variable), r is the instantaneous growth rate (a parameter), and K is the carrying capacity (also a parameter). To solve this differential equation in R we are going to give `lsoda` a function which describes the model Equation (8), some values for r & K, and some initial conditions (e.g. the initial values of time and population size). `lsoda` likes its arguments to be as follows:

```
> lsoda(initial values, time interval, function, parameters)
```

so lets first create our function describing Equation (8):

```
> ODEfun <- function(t, y, parms){
+     r <- parms[1]
+     K <- parms[2]
+     N <- y[1]
+     dN <- r * N * (1 - N/K)
+     list(dN)
+}
```

Notice our function has the format that `lsoda` likes. The 't' will be used as a time step by `lsoda`, the 'y' is used as the initial value of 'N' ('y' can be a vector if there are more than one dependent variables to follow), and 'parms' is the vector containing the values of the parameters to be used in the differential equation. The fourth line defines how N changes with changes in time ($\frac{dN}{dt}$) and the final line returns a vector containing the rate equations (in this case only one). Next solve the equation using `lsoda()`:

```
> logistic <- lsoda(c(N= 0.1), times= seq(0, 10, by= 0.1),
+     func= ODEfun, parms= c(r= 0.9, K= 5))
> head(logistic)     # take a peek at the first few rows
> plot(logistic[,"time"], logistic[,"N"], ylab="N",
+     xlab= "time", col= "navy")
```

[16]Special thanks to Ben Bolker for insight into `lsoda` and for providing this example.

Notice the start values, time interval, and parameter values were defined directly as arguments within `lsoda`. The starting value of N= 0.1 and r & K have been set to 0.9 & 5 respectively. Lastly, the time interval has been set using the `seq()` function we learned back in Section 4 and goes from $0 \rightarrow 10$ by increments of 0.1 (it's simply a vector of time steps). `lsoda` will produce a matrix of the solutions with 'time' as column 1, which is perfect for plotting as in Fig. 7.

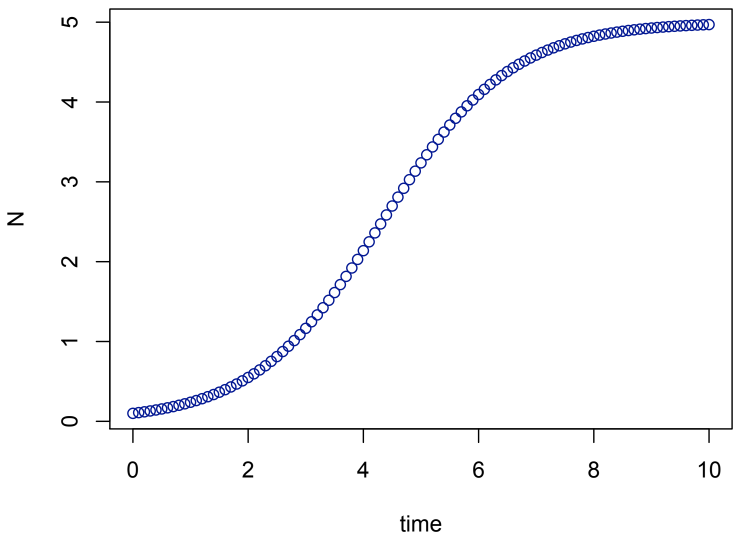

Figure 7: Logistic growth model with $r = 0.9$ and $K = 5$

2. The SI model

Now that you've gone through a simple example with one differential equation, let's follow up with a model system of a set of two, coupled ODEs following two dependent variables of interest. The same general steps apply, we just scale up the process slightly.

One of the most basic ODE disease models many of you are probably familiar with is the SI model[17] which stands for the **S**usceptible & **I**nfected classes within a host population (we're not concerned here about parasite populations ☺).

In Equation (9), β is the transmission rate which governs how quickly S become I, S is the number of susceptible hosts in the population, and (I/N) is the proportion infected (density-dependent infection). One standard assumption is a closed system (i.e. N is constant) and therefore $N = S + I$. We will be describing the rates of change of the susceptible and infected classes and since these two classes sum to a

[17]Loads of assumptions but still quite reasonable for some systems.

constant, only one differential equation needs to be explicitly described in the model (since $I = N - S$, $\frac{dI}{dt}$ is superfluous), yet for illustrative purposes we include both equations as an example of how one might set up a two dimensional (2D) model in R.

What do the differential equations look like for a simple SI model?[18] Pay attention, you'll be doing a full SI**R** (with an additional Recovered/Removed class) in a moment. Here we go:

$$\begin{aligned}\frac{dS}{dt} &= -\beta S\left(\frac{I}{N}\right)\\ \frac{dI}{dt} &= \beta S\left(\frac{I}{N}\right)\end{aligned} \tag{9}$$

Here is the R code I used for the SI model:

```
> tInt = seq(0, 25, by = 1/2)
> pars = c(beta= 0.75)
> Initial = c(S = 4999, I = 1)

> SIfun = function(t, y, parms) {
+     S <- y[1]
+     I <- y[2]
+     dS <- -parms[1] * S * (I/(S+I))
+     dI <- parms[1] * S * (I/(S+I))
+     ODEs <- c(dS, dI)
+     list(ODEs)
+}

> SIout <- lsoda(Initial, times= tInt, func= SIfun, parms= pars)
> head(SIout)
```

Notice that this time I created vectors up front defining the time interval, parameter values, and initial starting conditions and then simply used that vector name as an argument in `lsoda`. This is often a simpler way to a) change values quickly when exploring a model, and b) organize your parameters easily if there are many (here there was only one). I'll leave it to you to plot the results using the matrix `SIout` (since S & I are on similar scales, use `points()` to plot both on the same plot). What do you notice about the susceptible and infected populations? Does the epidemic peak quickly, then peter out, are there oscillations, etc.? Do the dynamics make sense considering the coupled differential equations of the model? Try repeating the process while exploring a few values of β. How do the population dynamics respond?

[18] Just for fun, google SI model and see what you get.

Exercises

1. Now it is your turn to expand on Equation (9) with a complete SIR model. Remember you can assume a closed system and therefore $N = S + I + R =$ constant, so you need only write two ODEs, however I suggest as a learning exercise that you do all three explicitly. Since this is an abstract example and we don't have any 'real' parameters to go on, you may have to play around with them until you get something interesting.

2. Next, consider an different type of two-dimensional model. Imagine a population that grows exponentially. Growth is partially determined by the activation temperature of some enzyme. This activation temperature is most efficient when it matches the temperature of the environment. Individuals who do not match the environment pay a cost in terms of growth. We can write the ODEs for changes in population size x (the ecological part of the model) and for changes in the mean activation temperature in the population, α (the evolution of the trait), as follows:

$$\begin{aligned} \frac{dx}{dt} &= x\big(r - (\alpha - \alpha_{opt})^2\big) \\ \frac{d\alpha}{dt} &= \varepsilon\big(-2(\alpha - \alpha_{opt})\big) \end{aligned} \tag{10}$$

where α_{opt} is the temperature of the environment and r is the growth rate of the population (try to see how the ecological equation captures the reduced growth rate when the mean activation temperature is **not** the same as the temperature of the environment), and ε is the genetic variation of the population for that trait (var(α)). The programming here is analogous to a classical interaction model, so don't let it intimidate you.[19]

   (a) What do the dynamics of population size, x, look like without evolution (i.e. there is no genetic variation)? Try the following values: $r = 0.2$ and $\alpha_{opt} = 10$ with initial conditions $x(0) = 2$ and $\alpha(0) = 9.15$ for a timespan from $0 \rightarrow 100$. Then make a plot of x vs. t.

   (b) What do the dynamics of population size look like when α evolves? What do the evolutionary dynamics of α itself look like? Try the following parameter values: $r = 0.2$, $\alpha_{opt} = 10$, and $\varepsilon = 0.03$ with initial conditions $x(0) = 2$ and $\alpha(0) = 9.15$ for a timespan from $0 \rightarrow 100$. **Hint**: make plots of x vs. t & α vs. t. How does evolution change the population dynamics? Now set $\alpha(0) = 10.85$ and observe the population dynamics of the system. Is this in agreement with your understanding of how stabilizing selection functions? Explore various parameter values both here and above in (a) while making predictions regarding the behaviour

[19]This is why I had you do the 2D SIR model above!

of the model. What happens when there is more genetic variation in the population?

11 Basic Statistics

Of course one of the great advantages of R is that you can perform many statistical analyses right here, within the R environment, without having to go to another program. The variety of analyses available in R is vast, unless you're pretty heavy into statistics, it is likely R will be able to fulfill your needs. So let's start with basic exploratory, comparative, & relationship statistics. Table 8 shows some of the more common functions you'll need. First let's return to an example we came across in Section 3. We were curious about a relationship we discovered in Fig. 1c, now we get to test whether this apparent relationship is statistically significant. Assume you've already determined a simple linear regression is appropriate, this is how we create the statistical model in R:

```
> plot(mydata$Heartwood ~ mydata$dbh)
```

Don't close this plotting window, we'll be using it in a moment (if you did, simply re-plot it using the ↑ key). Now we'll actually perform the regression analysis using a linear model via the `lm()` function (& giving it a name: 'fit').

```
> fit <- lm(Heartwood ~ dbh, data = mydata)
```

Notice that there is no immediate output of `lm()` as with most R functions; in order to see the results we must use `summary(fit)`. Now we can add the regression line from 'fit' to our existing plot using our old friend `abline(fit)`. Do this and see if you can reproduce Fig. 8 using various aesthetic arguments.

You can also peek under the hood of `fit` and see all its underlying components by typing `attributes(fit)`. You will see something like Table 7. Lastly, you can directly obtain and save these components as objects, perhaps for use in other analyses. For example, the coefficients of `fit` can be called using `coef()`. What does `coef(fit)[1]` produce? And `coef(fit)[2]`?

Table 7: The major components of a linear model.

```
$names
[1] "coefficients"  "residuals"  "effects"  "rank"
[5] "fitted.values" "assign"     "qr"       "df.residual"
[9] "xlevels"       "call"       "terms"    "model"

$class
[1] "lm"
```

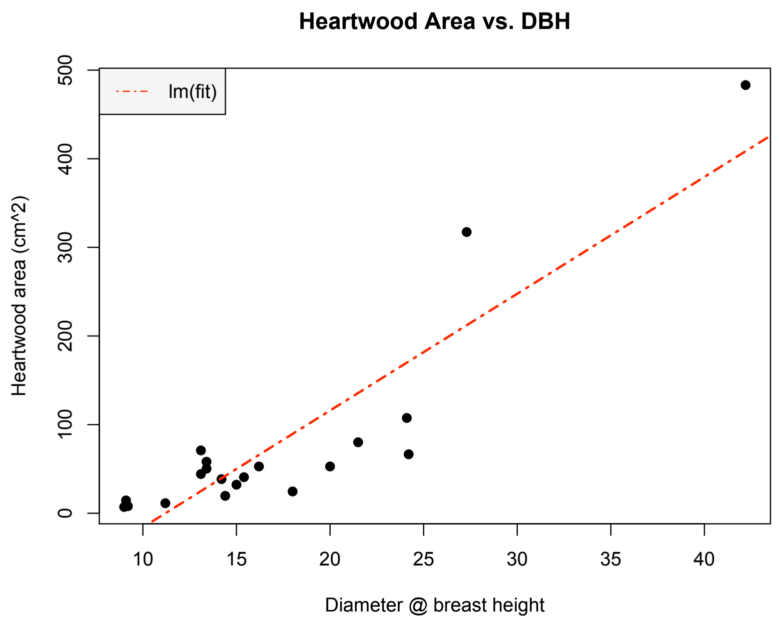

Figure 8: Graphical summary of the linear regression, `lm()`, performed on our tree allometric data (dbh vs. heartwood area).

To call other components, use the syntax `fit$name`. Determine the **residuals** of the model using this syntax. If you want to use the residuals vector you've just called, perhaps for graphing purposes, you can do so by simply giving it a name and saving it as an object, however you should prefer `residuals(fit)` over `fit$residuals` and likewise, `coef(fit)` over `fit$coefficients`.

Exercises

1. In the linear model above, lets assume new information comes available which violates the assumptions of a linear regression. How might you now explore the relationship between tree diameter & heartwood area? How might you look at parametric and non-parametric alternatives?

2. Create a new factor variable (i.e. column) to be added to 'mydata' called `Temp` which groups the seasons spring/summer & fall/winter into levels of either 'Warm' or 'Cool' (two levels for Temp). Now conduct a simple *t*- test for any of the continuous variables in 'mydata'. For example, is there a difference in bark thickness depending on whether it is warm or cool? Or does bark thickness correlate with whether a tree is infected or not? I suggest you first check for a normal distribution and equal variances. Depending on the variable you choose, you may need to find a non-parametric rank test.

Table 8: A few of the functions in R for statistical modeling and data analysis. There are **many** more, but you will have to learn about them somewhere else. The statistical functions such as var & sd assume values are samples from a population and compute an estimate of the population statistic (e.g. for example sd(1:3)=1).

`hist(x)`	Histogram plot of value in v
`mean(x),var(x),sd(x)`	Estimate of population mean, variance, standard deviation based on data values in x
`median(x)`	Median value of x
`cor.test(x,y)`	Correlation between the two vectors x & y
`t.test`	One & two sample Student's t-test
`pairwise.t.test`	Pairwise t-test
`chisq.test`	χ^2 test; differences in frequencies
`binom.test`	Binomial test for differences in proportions, binomial samples
`aov, anova`	Analysis of variance or deviance
`var.test`	Test of equal variance of sample means (Levene's test)
`ks.test(x,pnorm)`	Kolmogorov-Smirnov test of x to Normal distribution
`ks.test(x,y)`	K-S test, do x & y come from $different$ distributions?
`wilcox.test`	Mann-Whitney or Wilcoxon U non-parametric rank tests
`bartlett.test`	Bartless's test for equal variance, multiple treatments
`kruskal.test`	Kruskal-Wallis non-parametric rank test, multiple treats
`lm`	Linear models (regression, ANOVA, ANCOVA)
`glm`	Generalized linear models (e.g. logistic, Poisson)
`nls`	Fit nonlinear models by least-squares
`optim`	Minimize (or maximize) a function over one or more parameters
`lme, nlme`	Linear and nonlinear mixed-effects models (repeated measures, block effects, spatial models)
`manova`	Multivariate analysis

3. Based on Fig. 1 it seems as though species differ in sapwood depth. Test this hypothesis using a simple ANOVA. Then look at other continuous variables via boxplots, look for other differences by species (or season), and test for differences. Come up with your own questions and hypotheses, then test them statistically.

4. For any of the comparisons above, produce a barplot including error bars (SEMs or CI95s).[20] You can use the function you created in Section 9 and draw the lines yourself or you can use the function `barplot2()`[21] which can simplify the process.

[20]As mentioned, I prefer boxplots for this purpose because it shows the actual data points, but some advisors/journals/fields traditionally prefer barplots, so it's good practice to know how.

[21]Within the 'gplots' package.

By now you should have a pretty firm introduction to the practical applications of R for ecologists. You will be able to import and explore data and plot it to produce pretty graphs, perform some simulations (via loops), write your own functions, and carry out basic statistical analyses. You will rely heavily on these techniques over the next few days during other parts of the EEID 2009 Workshop.

Contributors

Stu Field, Dept. of Biology, Colorado State University, Fort Collins, CO.

Ben Bolker, Dept. of Zoology, University of Florida, Gainesville, FL.

Colleen Webb, Dept. of Biology, Colorado State University, Fort Collins, CO.

Mike Antolin, Dept. of Biology, Colorado State University, Fort Collins, CO.

Aaron King, Depts. of Ecology & Evolutionary Biology and Mathematics, University of Michigan, Ann Arbor, MI.

John Drake, Odum School of Ecology, University of Georgia, Athens, GA.