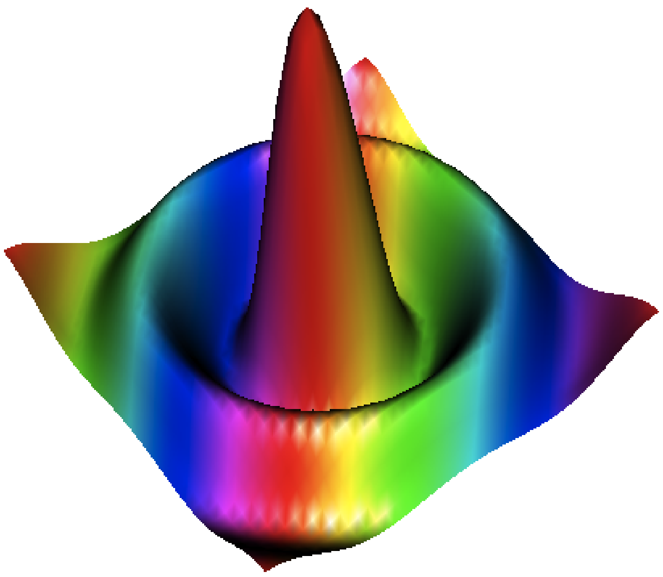
R Basics Tutorial for Ecologists

EEID Workshop: by Stu Field*

TimeStamp: 9:09 A.M., Sunday, 19 May 2012



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

Contents

1 What exactly is R?

R is an object-oriented scripting language that combines:[1]

- A dialect of the programming language S, developed by John Chambers at Bell Labs, that can be used for numerical simulation of both deterministic and stochastic dynamic models
- An extensive set of functions for classical and modern statistical data analysis and modeling
- Graphics functions for visualizing data and model output
- A user interface with a few basic menus

R is an *open source* project, available for free download via the Internet (`http://www.r-project.org`). Originally a research project in statistical computing, it is now managed by an international core development team that includes a number of well-respected statisticians, and is widely used by statistical researchers (and a growing number of theoretical ecologists and ecological modelers) as a platform for making new methods available to users. A standard installation of of R includes extensive documentation, including an introductory manual ($\sim$ 100 pages).[2]

2 Introduction to R

Reading a tutorial about the applications and capabilities of R is a great start, but until you do it yourself and are actually forced to solve problems on your own in R, you will become neither confident nor proficient. This is precisely what the following exercises are designed to do; give you practical programming experience in R. Before you start, I recommend you get used to running R from script files, rather than directly from the command window (i.e. The Console). It'll make your life much simpler, especially for more complex computations. For those of you using MacOS, R already comes with a fairly nice script editor. Alternatively, for those using Windows, I would recommend a supplementary script editor like RStudio, which is becoming rather popular in recent years (`http://www.rstudio.org/`).[3]

You may occasionally want to run the command `rm(list=ls())` to remove all objects from R's memory. However, it is *not* good practice to embed this in your script files by default, because it can trip up anyone else who uses your code and isn't expecting it. 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, e.g. `XYZdata.csv` and `XYZdata.R`, where the `*.R` file contains all the necessary commands, analyses, and plots for that particular data set in `*.csv`.

2.1 Installing Packages

Packages containing additional functions for R to use can be downloaded from the internet.[4] There are essentially two steps to using packages: (1) you need to download and install the

[1]From Ben Bolker (2008). *Ecological Models and Data in R*. Princeton Univ. Press, Ch. 1.

[2]See `R-intro.pdf` accompanying this tutorial.

[3]Really, any editor or GUI that can (1) hot-paste code chunks into R; (2) do syntax highlighting; (3) match closing with opening brackets will do. **Do not under any circumstances use Microsoft Word to edit your R code ...**

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

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 your CRAN mirror (somewhere within your current country is probably a good idea).
4. Select the package you want from the menu (in alphabetical order).
5. Click OK and wait for it to install (should auto-install).
6. The package is now installed and ready for you to call for it (i.e., load it as stated above).

Alternatively, if you know the name of the package you want to install you can just type `install.packages("pkgname")` at the R prompt.

Not all packages are independent (functions from one package may use functions from within other packages). Usually the required dependencies will load/install automatically; on the other hand, if you get an error message you will have to install them yourself. 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 your machine.

For MacOS, the process is very similar except you click on Packages & Data from the main menu and then Package Installer for package installation. There is an option to also install dependencies at this point. To see which packages are currently installed on the machine, go to Package Manager, where you can check the box of the package of interest to load it.

2.2 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 – just type `?functionname`. But be forewarned, it can take a little time to decipher these menus; at first they may seem like gibberish but with time this will pass. It is a very handy tool that I use almost daily. There are a few key sections of the Help Menu to highlight:

Table 1: Summary of the key sections of R's Help Menu.

Section	Description
`line 1`	Gives the function name and the package in `{...}`
`Description`	A short general description of what the function does
`Usage`	Illustrates proper syntax; with function arguments and defaults
`Arguments`	Details about the various function arguments (important!)
`Details`	Any additional notes and caveats that a user should know
`Value`	Description of what the function returns following execution
`References`	Who wrote it and their source if applicable
`See Also`	Related functions that might also be useful
`Examples`	Most useful . . . fully executable examples!

Let's take a look at the Help menu using the function `apply()` as an example. In the `R Console` 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 an 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 to include. 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, nrow = 4, ncol = 4)  # create a matrix called Mx
Mx

##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   13
## [2,]    2    6   10   14
## [3,]    3    7   11   15
## [4,]    4    8   12   16

apply(Mx, 1, mean)

## [1]  7  8  9 10

apply(Mx, c(1, 2), sqrt)

##        [,1]   [,2]   [,3]   [,4]
## [1,] 1.0000 2.2361 3.0000 3.6056
## [2,] 1.4142 2.4495 3.1623 3.7417
## [3,] 1.7321 2.6458 3.3166 3.8730
## [4,] 2.0000 2.8284 3.4641 4.0000
```

This tells R to calculate the mean of the *rows* `(,1,)` of `Mx` (by column is `(,2,)`). 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 fourth line above uses the `sqrt()` function to take the entry-wise square-root of `Mx`. **Note:** how you construct the command 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 – probably not what you had in mind). Likewise, the construction above is pretty redundant (and unnecessarily complex), since it's exactly the same as `sqrt(Mx)`. Lastly, take a look at the `Examples` section 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, try to move on, and we will take care of it during the workshop. Good luck and happy Rrrrring!

3 Walk before you run

Before we get into more advanced procedures in R, you will need to familiarize yourself with the basic input structure of R, completing interactive calculations, assigning variables, and becoming comfortable with the R console and visual interface.

At the R console notice the drop down menus and familiarize yourself with where you can find many of the options you will need during an R session. The console itself, upon startup, will state the version of R you are running and some basic information about the developers of the software. Also of importance here is the command to exit R, which is `q()`. Commands can be typed directly into the console at the `>` prompt and executed by pressing `Enter`. As stated previously, for shorter, non-repetitive commands, utilizing the console is fine, but for the most part I strongly suggest you make it a priority to learn to work from script files. You may then execute lines or chunks of R code by sending them to the console and in this way you are able to repeat a command by resending the code rather than retyping the entire line(s) in the console (which is extremely tedious). For Windows script editor: place the cursor on the line of interest, or highlight multiple lines, and hit `CTRL+r`. For the Mac: hit `Cmd+Enter` from the built in Mac editor. Of course, if using an auxiliary script editor (such as `RStudio`), you bypass this process and send code chunks directly from the external script editor. For now, lets use the console for the following simple interactive calculations.

3.1 Basic calculations and input

R uses most of the familiar mathematical operators you may be accustomed to from other computer languages. Addition, subtraction, multiplication, division, exponentiation, e^x, etc. are `+, -, *, /, ^, exp(x)`, respectively. In addition, when working from the console you may use the arrow up key (↑) to scroll through the previous few commands to the prompt and edit them (using the ←→ keys) as desired. Lets begin with some simple computations: type `6 + 9` and then `Enter` at the command prompt (`>`) of the R Console.

```
> 6 + 9

[1] 15
```

The answer is returned with `[1]` next to it, which represents the number of the first element of that output row (in this case only one entry). Try some simple calculator-type calculations on your own to get used to the input/output style of R. Rules of mathematical order of operations also apply. Try typing

```
> 7 - 12/3^2

[1] 5.6667
```

although, when making more complex calculations I recommend using parentheses, this is much less for R than for your own sake. Also note, you may separate an equation (line break) to a second line by hitting `Enter` and this will result in a *continuation prompt* (`+`), where R tells you that you haven't finished and prompts you to continue.[5]

[5]For the remainder of this tutorial I have removed the continuation prompt (`+`) in printed R code to simplify the cutting and pasting of code from the tutorial, but you will likely see it during your session, particularly if you directly use the Console.

```
> (18 * 9) * (2*(3 + 0.5)) -      # R knows you want to subtract something
+    1.5*(81)^0.5

[1] 1120.5
```

You may also separate multiple different commands on a single line using the semicolon (`;`).

```
> 7/2; 8*3; 9/3*10; 64^(1/3); log(45)

[1] 3.8067
```

Up until now the input and output are printed in this document more or less exactly as you will see it on the Console, showing the prompt (`>`) and continuation (`+`) characters; from now on the prompt and continuation characters will be missing, and the results will have comment characters (`##`) in front of them, to make it easier for you to cut and paste the results into R if you like.

3.2 Variable assignment

R carries out calculations with variables in the same manner as numbers (as above). You may assign values to variables (i.e. create objects) by using either the `=` or `<-` symbol, which is a "less than" symbol immediately followed by a "minus" sign.[6] This symbol can be viewed as "gets assigned to": `z` "gets assigned to" `y*(x/2 + 7) + sqrt(y)`. When assigning/naming variables, remember they must begin with a letter, but may contain numbers and decimal points thereafter, and that assignment is case-sensitive (`variable.1` $\neq$ `Variable1`). Try the following:

```
x = 7
y = 9
z <- y * (x/2 + 7) + sqrt(y)
```

Notice the result (`z`) was not displayed in the console. Once assigned, a variable (or object) is stored in R memory as an object and to view the result you must first "call" the variable by typing the variable name and pressing `Enter`. In calculating `z`, R remembers that `x` and `y` have already been assigned a numeric value and uses those values in calculating `z`.

```
x

## [1] 7

y

## [1] 9

z

## [1] 97.5
```

A trick you can use if you want to both assign and print the value is to surround the entire expression in parentheses:

[6]In practice `<-` can be used interchangeably with `=`, in almost every situation you will encounter. Opinions differ about which is better.

```
(Ex1 <- y^x - z^2)

## [1] 4773463
```

Try using the up-arrow (↑) to edit the value of `y=5` and then recalculate `z` and `Ex1`. We can now try a more practical example combining assignment and calculation using a list of numbers called a vector (more on vectors in § 6). Remember that R is case sensitive, so `X` $\neq$ `x`.

```
X <- 1:10
Y <- (1 - 1/X) + 2
X

##  [1]  1  2  3  4  5  6  7  8  9 10

Y

##  [1] 2.0000 2.5000 2.6667 2.7500 2.8000 2.8333 2.8571 2.8750 2.8889 2.9000
```

This will return two lists[7] of numbers which can then be plotted to display the relationship between x and y. We will discuss plotting in more detail in § 10.

```
plot(X, Y)
```

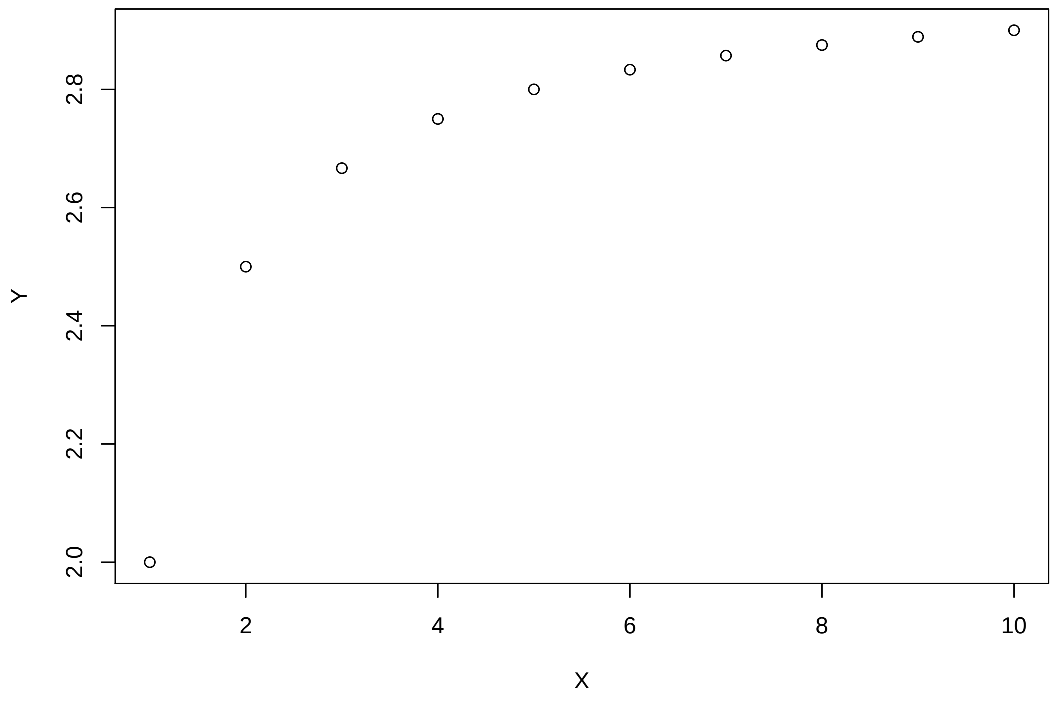


Figure 1: Your very first plot using R – the relationship between `X` and `Y`.

[7]I'm using the term "list" loosely here, in R the term `list` is a unique class of object. Technically, `X` and `Y` are *vectors*.

4 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. For the § 4 and 5 I suggest reading the supplement by Ben Bolker entitled *Importing data into* R. In addition, Chapter 2 of his book, *Ecological Models and Data in R*,[8] provides a more comprehensive discussion of how to shape your data into a compatible form for analysis in R (including troubleshooting tips for common data formatting issues). Importing data is 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`), using Save as. Two small hints:

1. make sure your home R directory is set to the location where your datafile is located.
2. 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 contain any combination of vectors (columns of data) that are of integer, numeric, character, or factor classes. This differs from a `matrix` object which can only contain one class of data.

One last note on data file organization: it's important to know the difference between *wide* format and *long* format. Table 2 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. Wide format is often easier to read, but many analyses require the data in long format. There is a function in the base installation of R, `reshape()`, that converts between the formats (somehow I can never remember how to use it! – `?reshape`). Lately I've been using the `melt()` and `cast()` functions (confusingly within the {reshape} *package*), instead of `reshape()`. Use whatever tool/function you like to get your data into long format.

Table 2: Long vs. wide format.

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

[8] Bolker, B. (2008). *Ecological Models and Data in R.* Princeton University Press. pp. 408. Graciously provided in `pdf` format by B. Bolker (ISBN: 0691125228; `http://www.math.mcmaster.ca/~bolker/emdbook/`).

Exercises

1. Read the `TreeData.csv` file into R memory using `read.csv("TreeData.csv"`. Add the argument `(..., row.names= 1)` and note what it does. Be sure to redefine (overwrite) this data frame back to the original (without the `row.names` argument) because you will use the original data below. 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. So you don't get hung up at this preliminary stage, if you are having difficulty reading `TreeData.csv` in to memory I have provided a supplementary "cheat" file called `TreeData.Rdata` which should help. Simply double-click this file and its contents (the object `mydata`) should be read into memory). Alternatively, type `load("TreeData.Rdata")` directly in the console.[9] Make sure this was done correctly by calling `mydata` in the console (or type `ls()` to make sure `mydata` is in R's list of objects in memory). I have also included the file `R_Tutorial_All_Data.Rdata` which contains **all** the data sets required for this tutorial. Try to read the data into memory using `read.csv()` but as a last resort use the `*.Rdata` files.

3. 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.

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

4. The first thing you'll need to do is make sure your data set is complete, that means no pesky `NA`s (the code for missing data in R). First, determine if some of those `NA`s 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)`.[10]

5. 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()`[11] function above), or some discrete variable rather than a continuous one. Try assigning a column of `mydata` a name[12] (e.g. `species = mydata$spp`) and using the `table()` function, `table(species)`. Repeat this with other variables within the data frame.

6. 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.

[9] Be sure that the working directory is set correctly!

[10] The Bolker references give a excellent and detailed description regarding how to deal with `NA`s.

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

[12] It is a good idea to avoid using the following: `c`, `q`, `t`, `C`, `D`, `F`, `I`, and `T`. These are already reserved for other meanings in R (`T` and `F` are abbreviations for `TRUE` and `FALSE`). It's also a good idea to avoid `l` and `O` (lower-case letter "L" and upper-case letter "O"), because they are easily confused with 1 (one) and 0 (zero).

5 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 more appropriate for arranging stand alone vectors (see § 6) as opposed to data frames or matrices (see § 8). Lets try a simple example:

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

The first line returns a vector indicating the row numbers of the entries in the `dbh` column in increasing order. The second line tells R to rearrange the data frame by increasing values of *dbh* and to order **all** columns in this fashion. The part before the comma determines the order of and which rows to include. 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 leave the column declaration blank which defaults to all columns:

```
mydata[order(mydata$dbh), ]
mydata[order(-mydata$dbh), ]  # decreasing
```

The default for `order()` is increasing; you may add the negative symbol (`-`) to arrange the data frame according to *decreasing* values of the desired column. This will work only for `numeric` or `integer` class variables; for non-numeric sorting variables you must use `rev(order())` – for example:

```
mydata[rev(order(mydata$Infected)), ]  # for a factor (character string)
```

Table 3: 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(mydata$dbh),]`). Also try sorting by the column *spp* (both ↑↓), 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 `NobarkArea`.

3. Now create a new data frame called `tree1` which contains only `tree #`, `SapDepth`, and `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, ]
```

   ... check how many cases were removed

```
BTzero <- mydata[mydata$BarkThick != 0, ]
nrow(BTzero)

## [1] 15

nrow(mydata)

## [1] 20
```

   In the first call above you are asking R to return **all rows** (notice the comma) where the column `BarkThick` is **not** equal to zero. These logical arguments (i.e. `!=`) can also be used in combination with others using the `&` symbol. For example, limit the data frame to only those trees which have a sapwood area ≤ 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 assign it a new variable name and work with that new data frame. Table 3 shows other operators for comparisons and selecting data.

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.

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

   Here is the R code for Fig. 2 (ensure that `mydata` is still in memory!):

```
par(mfrow=c(1,3)) ## create plot array of 1 row x 3 columns
hist(mydata$SapDepth, xlab= "Sapwood Depth",
     main= "Histogram: Sapwood Depth", col= "gray50")
boxplot(SapDepth ~ spp, data= mydata, ylab= "SapDepth",
        col= "darkslateblue", main= "Boxplot: Sapwood Depth by Species")
plot(mydata$dbh, mydata$Heartwood, pch= 17, col= "darkred",
     ylab= expression(paste(plain(Area)," ",(cm^2))), xlab= "DBH (cm)",
     main= "DBH vs. Heartwood & Sapwood")
points(mydata$dbh, mydata$SapArea, pch= 19, col= "darkgreen")
legend("topleft", legend= c("Heartwood", "Sapwood"),
       pch= c(17, 19), col= c("darkred", "darkgreen"), bg= "gray95")
```

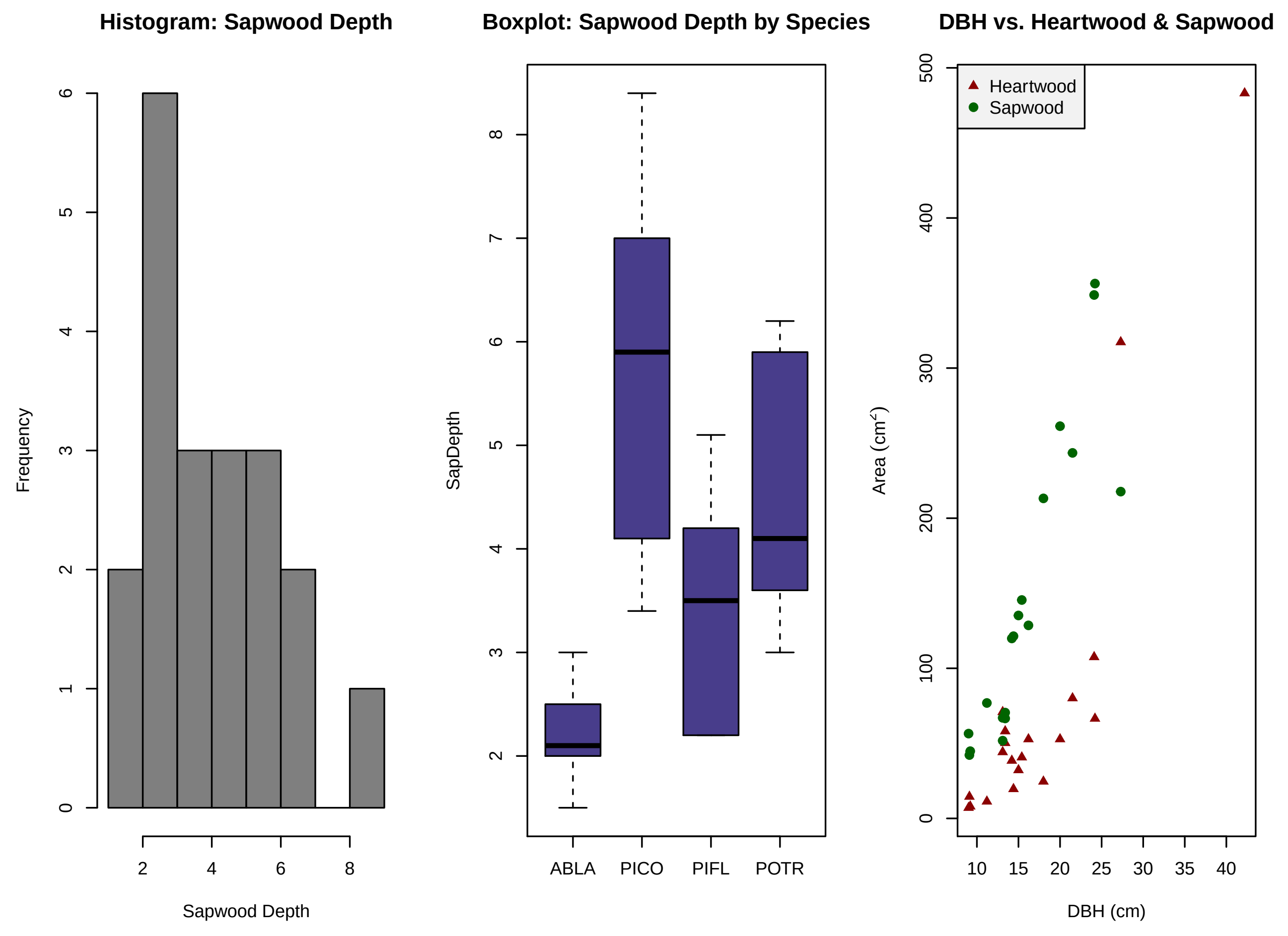


Figure 2: Quick and dirty visual exploration of data.

Fig. 2 shows examples of various types of exploratory plots. Use the code above to replicate it. Look at `mydata` and explore some relationships for yourself graphically (e.g. do species differ in sapwood depth?). Upon visual inspection of Fig. 2b it looks like they do. To confirm this, you would then have to show if this difference is statistically significant (see § 13). 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 § 13).

Lastly, the `qplot()` function (from the `ggplot2` package, which will be covered more later in the week) allows us (among other things) to plot each species in a separate color *or* in its own separate subplot. In the first case shown in Figure 3, we specify that `SapDepth` and `dbh` are the x and y coordinates respectively, we tell R to look within the data frame `mydata` for the variables, and we specify that the points should be colored according to species. In the second case, we say that the plot should be divided into *facets* according to species (we have to specify it as a formula, which is why we say `~spp` instead of just `spp`).

Try it out for yourself with different dependent and independent variables. What happens when if you use a two-sided formula for the facets (`Infected~spp`) instead of just `~spp` as your grouping factor? (You will need the `ggplot2` package installed and loaded to use `qplot`.)

```
qplot(SapDepth, dbh, data = mydata, color = spp)  ## left subfigure
qplot(SapDepth, dbh, data = mydata, facets = ~spp)  ## right subfigure
```

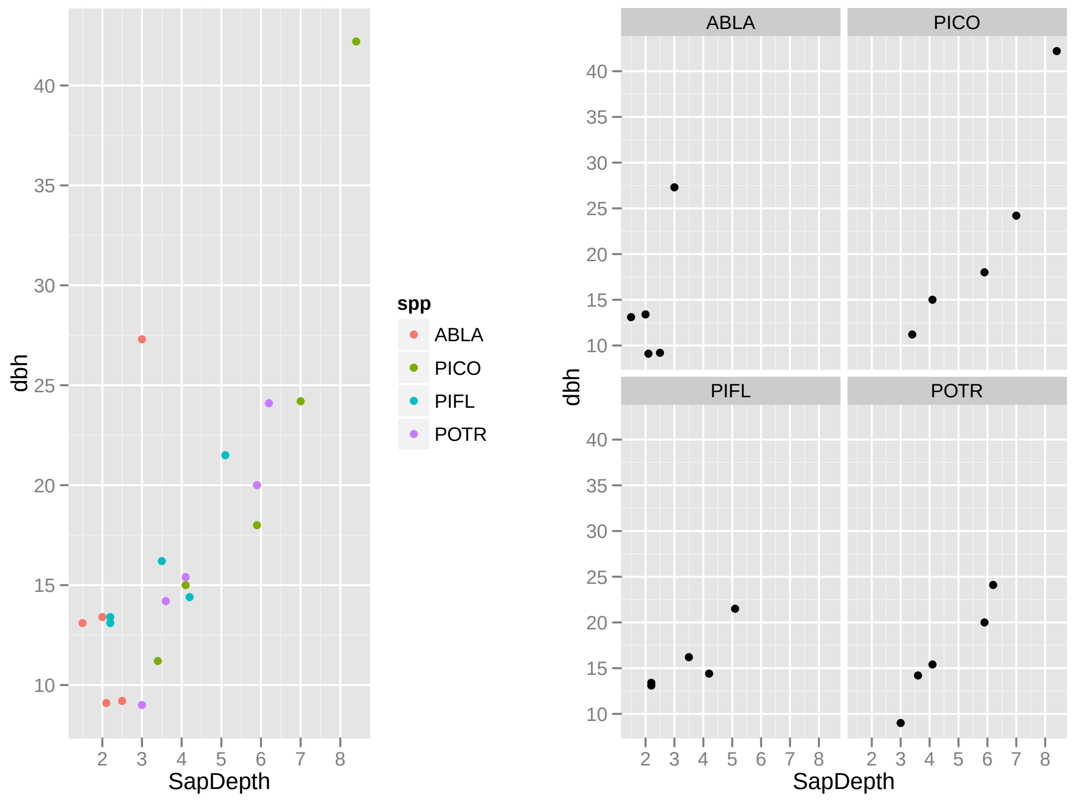


Figure 3: More quick and dirty visual exploration of data.

6. Make a Barplot of the mean *Area Without Bark* first by `season` and then by `species`. The conclusions from these results are meaningless, but you'll soon 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()`[13] is an extremely useful function that applies a function to the values in a vector and produces a table based on one or more grouping variables (e.g. factors) and shows the result of that function for each group. This sounds complicated but it really isn't. To illustrate, 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.

[13]Similar to `apply()` from above, but can be applied to 1D objects. It can be viewed as a combination of the `table()` and `apply()` functions.

```r
Nobark.season <- 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 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?).

6 Vectors

A vector is basically just a list of values, all the same type or class (numeric, logical, character). The numeric vector is a list of numbers and is one of the most often manipulated objects in R. Actually, a number in R is treated as a vector of length 1. They are used for everything from calculating means, plotting graphs, storing outputs of functions, or simply storing parameters for later use within a larger program. It is essential to learn to master their manipulation. Some R functions create vectors, but you can also create/assign vectors yourself. For example:

```
v1 <- c(1, 2, 7, 8, 9, 13)
```

Simply typing `1, 2, 7, 8, 9, 13` without `c()` results in an error. In addition, vector elements can be named. For example:

```
v2 <- c(first = 1, second = 1.2, third = 1.6, fourth = 2.1)
```

There are numerous ways to refer to elements in vectors (*vector addressing*):

- by position: `v1[1]` or (multiple elements) `v1[c(1,2)]`
- by name: `v2["first"]`; only if you've already assigned element names as in `v2`
- by exclusion: `v1[-1]` drops the first element
- with logical vectors: `v2[c(TRUE,TRUE,FALSE,FALSE)]` gives the first two elements only; more usefully, `v2[v2>1.2]` first computes a logical vector `[FALSE,FALSE,TRUE,TRUE]` and then uses it to select the last two elements only.

Most of R's functions are *vectorized*; if you give them vectors they automatically do the right thing (which can frustrate MATLAB users!). Functions of two vectors such as `(c(1,3,5,7) + c(2,4))` will automatically replicate the shorter vector until it is as long as the longer one, then carry out the calculation and give a warning message if the longer vector is not an even multiple of the shorter. For example:

```
v1

## [1]  1  2  7  8  9 13

3 * v1

## [1]  3  6 21 24 27 39

v1 + c(3, 2, 1)

## [1]  4  4  8 11 11 14

v1 * c(2, 4)

## [1]  2  8 14 32 18 52

v1 * c(1, 2, 3, 4)

## Warning message: longer object length is not a multiple of shorter object length
## [1]  1  4 21 32  9 26
```

Some of the most basic and useful functions for manipulating vectors are in Table 4. **Remember:** operations and functions are generally applied to vectors in R on an *element-by-element* basis.

Table 4: 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
`length(x)`	Number of entries/elements in vector x
`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` named `v3`.
2. Create a vector with the numbers: `2,2,2,5,5,9,9,8,8,8,8`. Try to use the `rep()` command and call it `v4`.
3. Create a vector called `v5` 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 do.[14]
5. Create a vector called `all.vecs` that includes all the numbers you just created (in `v3,v4,v5`) in one long vector (any order).
6. Create a vector of 15 random numbers from 50 $\rightarrow$ 150 and name it `mass`.[15] In this case assume a uniform distribution.
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. Now create a new vector called `mass.r` with these daily values rounded to two decimal places. What is the average daily weight gain?
8. As I said in § 5, I think `sort()` is more appropriate for stand-alone vectors, so try using it on `mass` (both increasing and decreasing).

[14]The `each=` argument within `rep()` can also be very useful.
[15]See Table 5 for more information about random number generation.

9. Create a new vector of length 7 with *i)* the minimum value *ii)* maximum value *iii)* mean *iv)* sum *v)* median *vi)* and range of `mass.r`.

10. Use the entries in the vector above to multiply the mean daily weight gain by the total weight gain, divided by the square root of its median, plus the log of the minimum daily gain. I cannot imagine why you would want to do this, but you get the point; you can use vectors to make calculations and then use those vector entries to make subsequent calculations. For example, you could place all the parameters of a model into one vector, then call them independently by vector addressing we learned above (i.e. `[]` brackets).

11. Now let's 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:

$$\text{Leaf Area} = \left(0.1 \cdot \delta^{(1-\beta)}\right) \; + \; \sqrt{\frac{\tau + 1}{\sigma}} \qquad (1)$$

    where δ, β, τ, and σ are tree diameter (`dbh`), bark thickness, area without bark, and 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 `^0.5`), but for cube roots or higher, you must use the "fractional exponents" syntax construction. Create an object called `LeafArea` that represents Eqn (1). You will use this object later in this tutorial. The resulting vector should look like this:

```
LeafArea

##  [1] 2.6192 1.8484 2.1508 1.9888 2.5730 3.0983 1.6385 1.3992 1.9445 1.6196
## [11] 1.8146 1.9693 2.6746 3.5551 1.5824 1.4610 1.5739 1.4959 1.6805 1.5938
```

7 Generating Random Numbers

R provides great flexibility and options for generating random sequences of numbers from various distributions. Table 5 shows the major functions used to produce numbers according to these distributions.

Table 5: 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 and standard deviation
`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: `mu` must be specified by na
`rgamma(n,shape,scale)`	Random n numbers from Gamma distribution

Of course you can also determine the probability density, distribution, and 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. 4 which shows what each of these functions do. Try to reproduce it using the `curve()` and `hist()` functions.[16] Below is the code for the first plot to get you started:[17]

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

[16]These functions are introduced in § 10. Look ahead for clues.

[17]Modified from Crawley, M.J. (2007). *The R Book*. John Wiley & Sons. pp. 950.

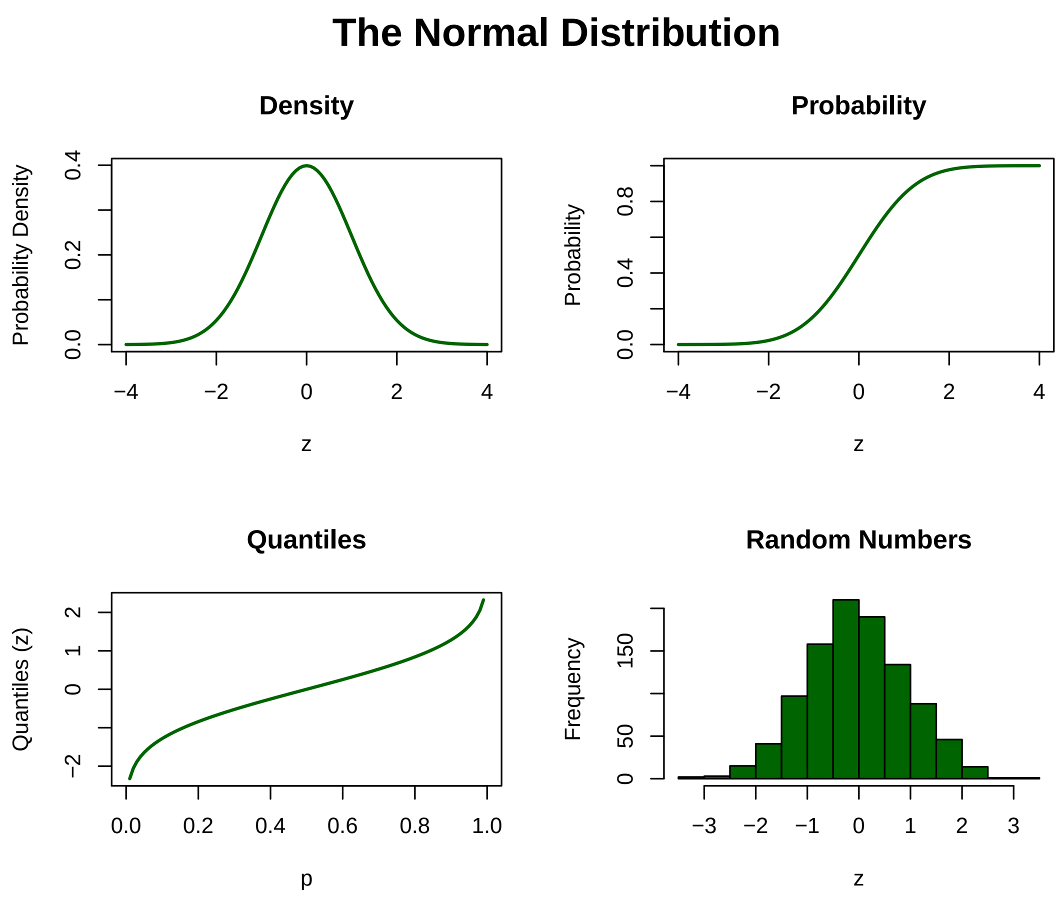


Figure 4: Reproduced shamelessly from *The R Book* (p. 219).

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 something similar, but with *SapDepth* as in Fig. 2a?

```
set.seed(666)
nD <- rnorm(1000, mean=400, sd=25)
hist(nD, main= "", xlab= "nD", ylab= "Probability Density",
   col= "gray88", freq= FALSE)
curve(dnorm(x, mean(nD), sd(nD)), from= min(nD), to= max(nD),
   col= "navy", lwd= 2, add= TRUE)
```

8 The Matrix

Besides evoking images of bullets flying in 3D surround slo-mo, the matrix (or 2-dimensional array) can be put to good use in R. Aside from their obvious use in population projections and classical matrix algebra. Be careful! R's default multiplication is set to the entry-wise product[18] using the `*` symbol. For proper matrix multiplication you will want to use `%*%` (see Table 6). If nothing else, matrices are useful in storing output of a given program/function, which is then to be used to some other purpose (e.g. plotting).

Table 6: 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 and 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 and 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 and right eigenvectors of matrix **A**

Exercises

1. Use the construction `matrix(v, nrow, ncol)` to create the matrix in Eqn (2) where v is a data vector created using the `seq()` function from Table 4. 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 Eqn (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

[18]The Hadamard product. Remember, the default in R is almost always entry-wise.

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 § 11).

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 § 2 to do the same thing.

5. Now use the commands `cbind()` and `rbind()` to attach these two vector of sums to create a new matrix **B** using the tools above which looks like Eqn (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 `LeafArea` 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`.[19] 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:

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

where $\vec{x}$ is a vector containing the number of individuals in a given age, stage, or class and **M** is the projection matrix. Consider the projection matrix **M** in Eqn (6) which contains 3 classes. The diagonal represents the survival rates of the classes, the sub-diagonal (0.1 and 0.3) represents the transition probability (i.e. class $1 \rightarrow 2$ and class $2 \rightarrow 3$) and the top row represents the fecundity of each class (the number of class 1 individuals produced per

[19] If you do export a modified data frame, be sure to rename the *new* modified data set with a unique file name. 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, as opposed to overwriting `*.csv` files.

individuals of class 2 or 3). Class 1 does not reproduce but if it did, entry $\mathbf{M}$[1,1] would be $s_1 + f_1$.

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

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

9 Loops

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 (see below) 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 an example of a simple `for`-loop:

```
Popn <- 1  ### create an initial population size
Gen <- 10  ### how many generations
for (n in 2:Gen) {
    Popn[n] <- Popn[n - 1] * 2
}
Popn

##  [1]   1   2   4   8  16  32  64 128 256 512
```

After calling `Popn`, this should look familiar, the population appears 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
}
pop.vec

##  [1]     1     2     4     8    16    32    64   128   256   512  1024
## [12]  2048  4096  8192 16384 32768
```

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, you can modify this code to replicate the classic expression, "double a penny every day for a month and you'll be a millionaire." Is this true? How long does it take to become a millionaire?

Lastly, 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 it could be any letter or even 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 generation, then roughly plot the results. We'll do more intense plotting in § 10. For now try:

```r
plot(1:Gen, Popn, type = "b")
```

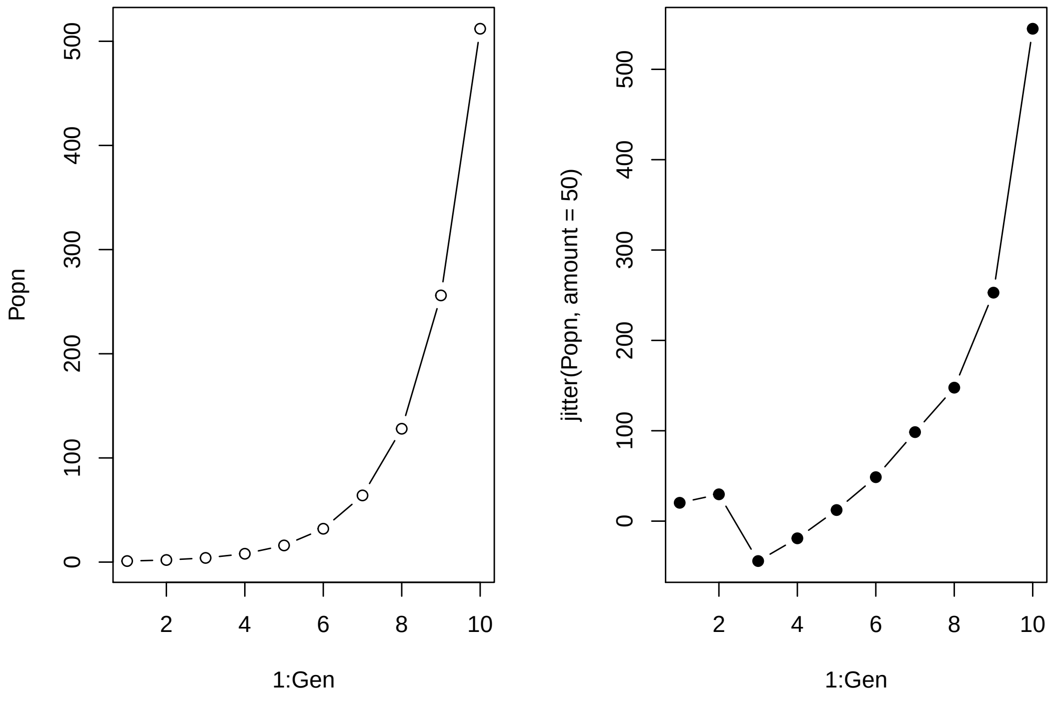


Figure 5: Quick graph of exponential population growth using a simple `for`-loop.

Alternatively, you could add noise to a data vector using the `jitter()` function.

2. Use a loop to create a vector representing a Fibonacci sequence running from 0 → 610. I suggest starting with the `Fib.v = c(0,1)`, then add to this vector at each iteration of the loop. In the end it should look like this:

```
Fib.v

## [1]   0   1   1   2   3   5   8  13  21  34  55  89 144 233 377 610
```

Then:

i) Use this vector to create the following matrix:

```
##      [,1] [,2] [,3] [,4]
## [1,]    0    3   21  144
## [2,]    1    5   34  233
## [3,]    1    8   55  377
## [4,]    2   13   89  610
```

ii) Use the `sample()` function to create a matrix of random Fibonacci numbers (see `?sample` to determine the proper syntax if necessary). For example:[20]

```
##      [,1] [,2] [,3] [,4]
## [1,]    8  233    0   21
## [2,]  144  377   89   13
## [3,]   55    1    1  610
## [4,]    5    3    2   34
```

3. Write a script using a *nested* `for`-loop which produces the following 5×5 projection matrix:[21]

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

4. Use a loop and Eqn (6) to project the 3 class population described above for 20 generations (i.e. $\vec{x}_i$, $i = 1 \ldots 20$). You will need to use a secondary matrix to store the results of the iterations (and then to plot them in § 10). You should use Exercise 8 in § 8 to guide you.

[20]Yours won't be *exactly* the same because of random sampling!

[21]It is not absolutely necessary to use a nested `for`-loop to accomplish this but it's a good exercise.

10 Plotting and Graphing

Remember what I said in § 5 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 7: List of the 8 numeric colors used by R. Thereafter, numeric colors are recycled, thus `col=9` is again `"black"`. There are many many other colors available using a string argument construction (i.e. `"midnightblue"`): see `colors()`.

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

One of the simplest ways view, visually explore, or summarize data is to plot them. The usual suspects are barplots, boxplots, scatterplots, and pie charts. However, depending on your audience they can be useful.

I can never remember the individual numeric codes for the various **points**, **line types**, and **colors** available in R plotting functions, so Fig. 6 below is a cheat-sheet:

```
plot(1:25, 1:25, xlab="",ylab="",pch=1:25,col=1:25,cex=2)
grid(lty=1, col="gray90")
points(1:25, 1:25, xlab="",ylab="",pch=1:25,col=1:25,cex=2)
title("Plotting symbol, line type, & color codes")
legend("topleft", legend=1:6, lty=1:6, lwd=1.5, ncol=2, bg="gray95")
legend("bottomright", legend=1:8, col=1:8, ncol=3, pch=19, bg="gray95")
```

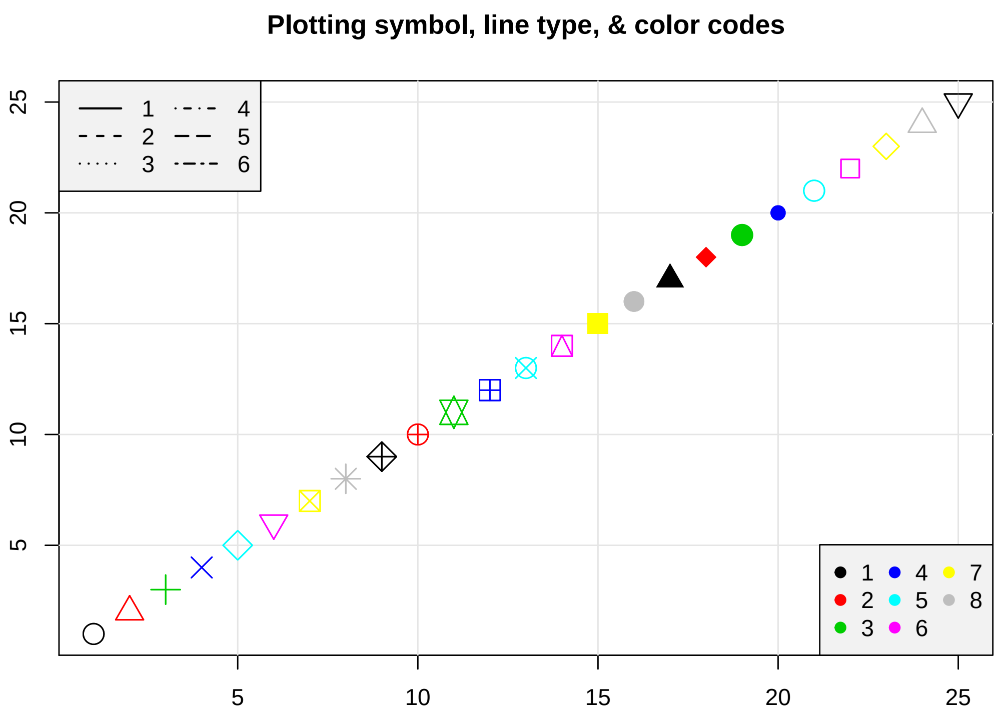


Figure 6: Basic numeric plotting codes used by R. For points, the symbols are accessed using the `pch=` argument and range 1 → 25 (some are redundant). Line types are shown in the legend, are accessed using the `lty=` argument, and range 1 → 6. Color codes (see Table 7) range from 1 → 8 and are accessed using the `col=` argument.

First create some dummy data to plot:

```
set.seed(876)
InfStatus <- factor(sample(c("Susceptible", "Infected", "Recovered"),
    size = 50, replace = TRUE))
I <- table(InfStatus)
I

## InfStatus
##    Infected   Recovered Susceptible
##          23          14          13

Genotype <- factor(sample(c("RR", "Rr", "rr"), size = 50, replace = TRUE))
G <- table(Genotype)
G

## Genotype
## RR Rr rr
## 17 22 11

table(Genotype, InfStatus)

##           InfStatus
## Genotype Infected Recovered Susceptible
##       RR       12         3           2
##       Rr       10         6           6
##       rr        1         5           5
```

```
par(mfrow = c(2, 2), mar = c(3, 2, 2, 1), oma = c(0, 0, 3, 0), bg = "white")
plot(InfStatus, ylim = c(0, 27))
box()  ### or barplot(I); box()
barplot(table(Genotype, InfStatus), ylim = c(0, 13), beside = TRUE)
box()
legend("topright", c("RR", "Rr", "rr"), fill = c("gray40", "gray70",
    "gray90"), ncol = 1, cex = 0.75)
boxplot(rnorm(50, mean = 15, sd = 3) ~ Genotype, col = "gray75")
pie(G, col = c("gray50", "gray70", "gray90"))
mtext("Basic R Plots", outer = TRUE, cex = 1.5, font = 2)  # main title
```

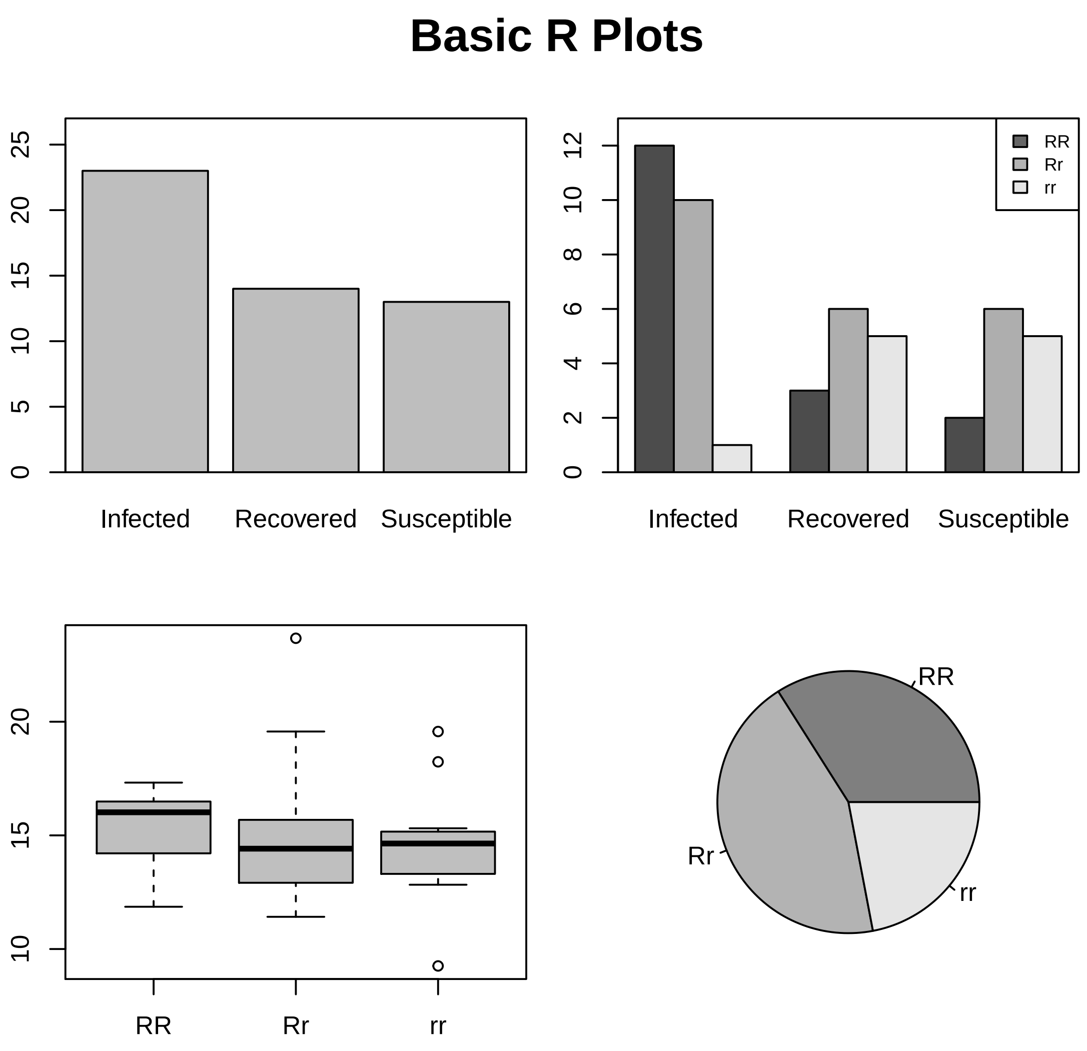


Figure 7: Examples of a few basic exploratory plots (and code syntax) for barplots, boxplots, and pie charts in R.

There are literally *endless* ways to modify graphics plots in R including colors, axis type, line type, adding a legend (as above), add a grid, background colors, labels, borders, etc. What happens if you change the beside argument = FALSE in Fig. 7b? The help pages for most of these functions can point you in the right direction for the syntax of what you would like to do. You may also want to use the par() to modify your plots; type ?par to see how truly massive this help file is! Odds are you'll find what you need buried in there, somewhere.

Exercises

1. Boxplots can be a particularly good way of presenting data and 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. 2b 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 the data points of a basic scatter plot. The `symbols()` function is great for this, and is especially good for geographical data where x and y are grid coordinates.[22] Read the data file `InfectionByCounty.csv` into R and make a geographical plot of infection counts for these data with blue solid circles and gray edges.

3. Now lets generate some fake data of a typical infectious disease outbreak. Lets assume that if an individual becomes infected they are infected for life (i.e. no recovery). We define the model by assuming some general underlying deterministic function, then assume some stochasticity about this deterministic function. We assume the variation around this model is normally distributed and will use the `rnorm()` function.

```
set.seed(125)
t <- seq(1, 30, by = 0.5)
y.det <- 0.4/(1 + exp(0.4 * (15 - t)))  # underlying deterministic model
p.inf <- rnorm(length(y.det), mean = y.det, sd = 0.02)  # add normal noise
plot(t, p.inf, main = "Outbreak XYZ", ylab = "Proportion Infected",
    xlab = "time")
```

[22]Especially longitude and latitude.

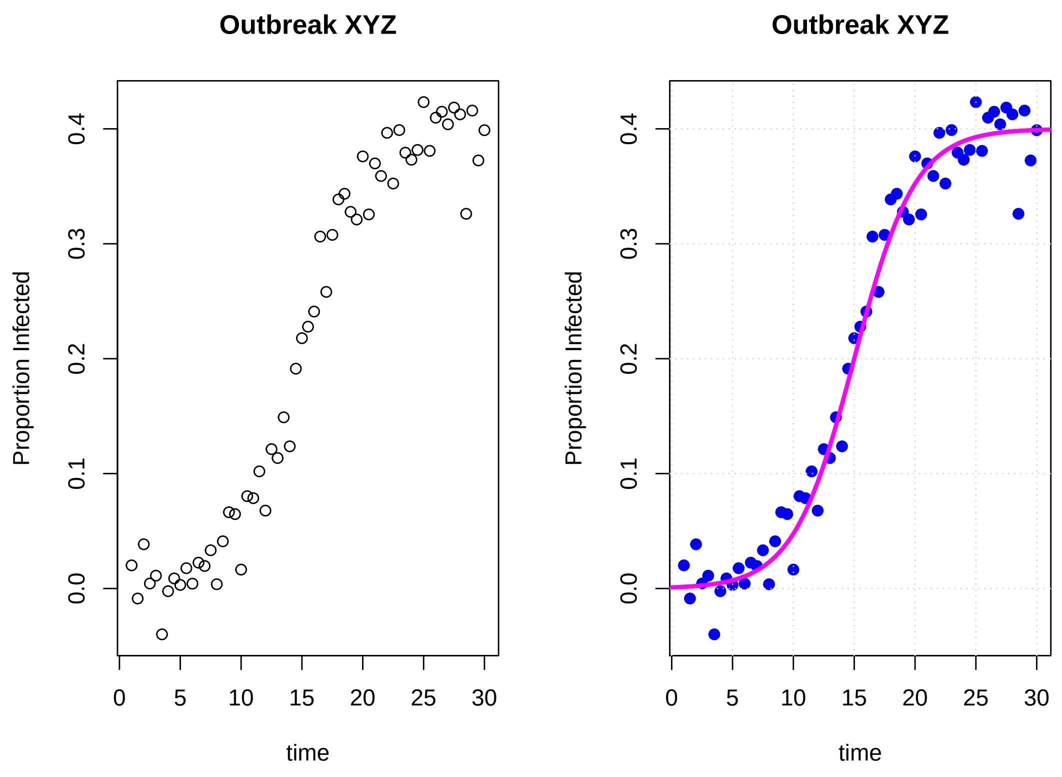


Figure 8: Some fake generated outbreak data.

Now plot these data, you should get something like Fig. 8a. You can use the `lines()` function to add lines to an existing plot as shown in Fig. 8b. See if you can replicate it using the `pch=` and `col=` arguments within the `plot()` function along with the `grid()` function for gridlines.

4. Plot the population projection you produced in the final exercise of § 9. You should get something like Fig. 9. Play with the arguments to get it to look like Fig. 9, including the legend.

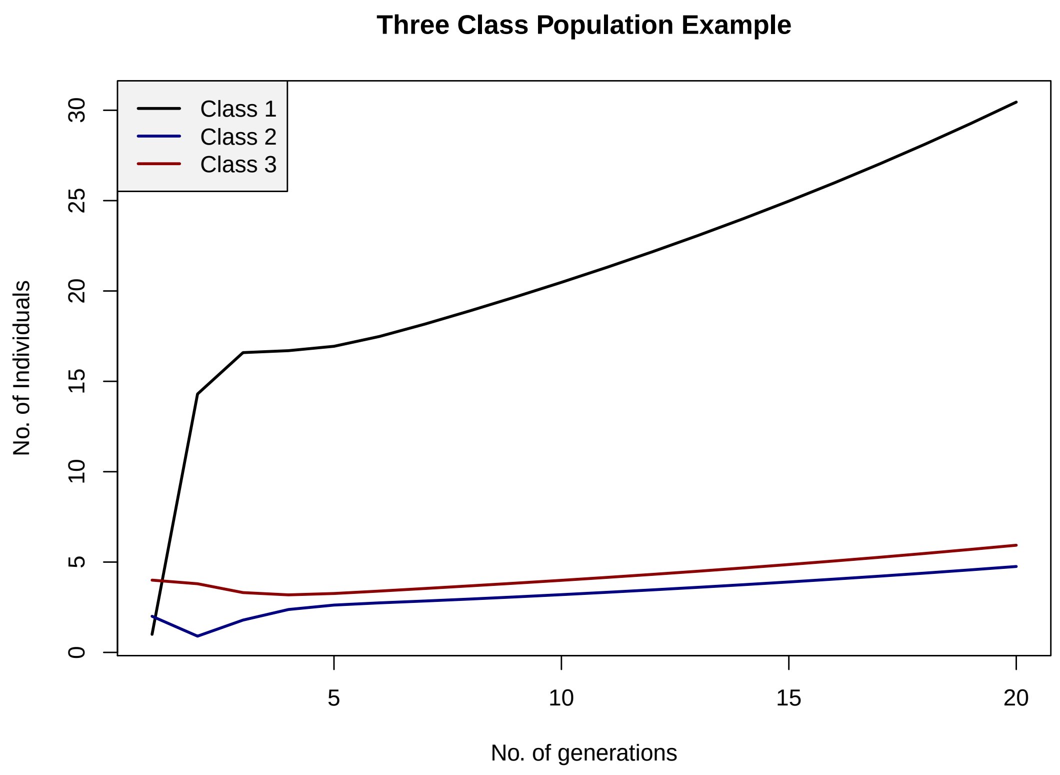


Figure 9: Quick population projection of 3 class model.

5. Matplot is a very good command for quickly graphing data multiple lines in the same plotting window. Your data frame or matrix 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. 10. If you are having trouble importing `*.csv` files I have also included an R script file of these data (`Pop6C.R`). Don't forget the gridlines and legend. Are the horizontal gridlines[23] on top of the legend and plot-lines or beneath them?

[23]See `?grid`

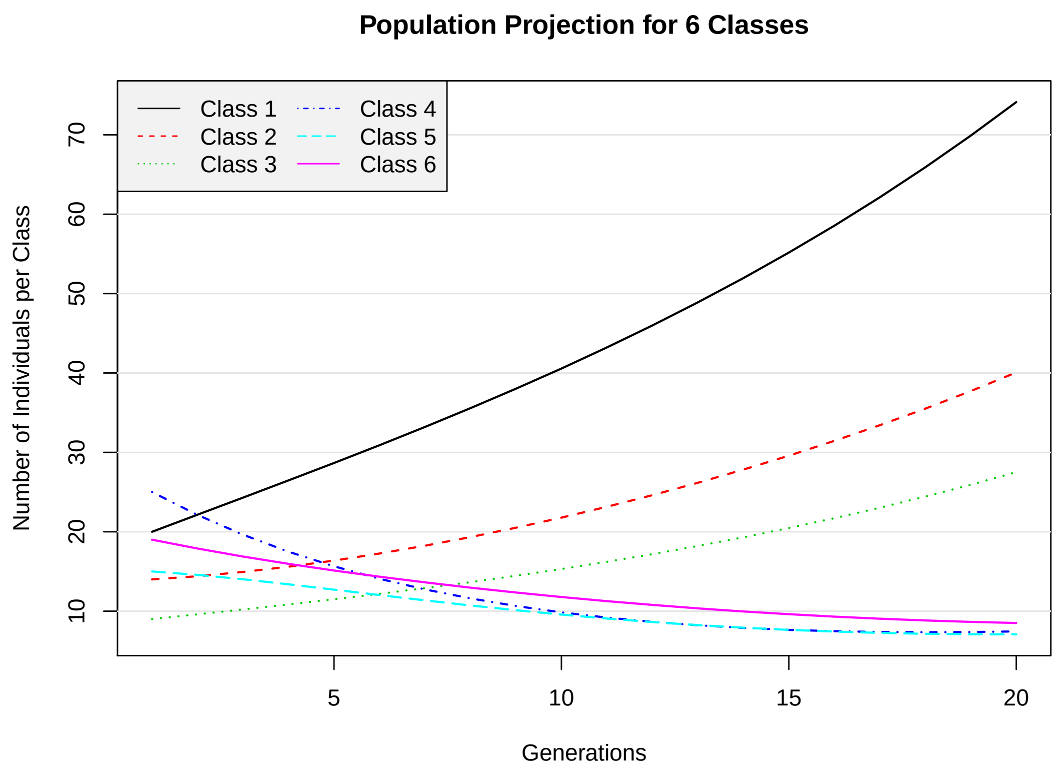


Figure 10: Using `matplot` to quickly graph simultaneous variables with the same independent variable (time/generations in this case).

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 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. 8b.

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

7. Yet another useful plotting function is `abline(a,b)`. This nifty little function adds a straight line with intercept = a and slope = b to an existing plot (in school I learned **m** = slope and **b** = intercept, so now I find it personally confusing to have 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()`, `curve()`, or `plot()`. We'll be using `abline()` again later in § 13.

11 Creating Functions

Writing your own functions (aka subroutines) in R is in my opinion the most useful tool/strategy in R and much of it's power stems from this flexibility. Let's start simple with the generalized format of a function:

```
myfun <- function(x, y, z) {
    expression_1
    expression_2
    expression_n
    Output  ## if desired
    ## or list(Output)
}
```

In this very generic example, `x`, `y`, and `z` are the arguments for the function and will have to be provided by the user (you). They are essentially the ingredients the function requires to perform its job and can be vectors, scalars, matrices, or data frames (any supported R object). It is also possible to provide the function with 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 `y=5`.

The expressions within the function will describe what `myfun` will do to `x`, `y`, and `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 object you wish to output, use `list()`. The function `cat()` can be especially useful for printing some output during the completion of a function (useful for debugging or tracking a long simulation). In addition, an output line is not absolutely *necessary*, only if you want R to return something when you hit `Enter`.

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 that is included in the `base` package installation of R.

```
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.[24] Type `ls()` and find a vector (or matrix) already in memory and execute your new function, then double check it against the `mean(vector)` function provided with R. Here it is applied to `LeafArea` from § 6.

```
xbar(LeafArea)

## [1] 2.0141

mean(LeafArea)

## [1] 2.0141

xbar(Mx)

## [1] 8.5

xbar(21:49)

## [1] 35
```

[24]This function will also work on matrices because they can be easily converted into vectors.

Also notice 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`, and `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* use a function to reassign the value of a variable.

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)
}
rMat(3, 7, 5, 37)

##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,]   12   20   16   15   10   26   14
## [2,]   33    9   34   26   14   35   18
## [3,]   23   34   12   16   18   30   26
```

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 95% confidence intervals (assuming a normal distribution). It uses Eqn (7) to calculate the standard error of the mean, and returns the upper and lower CI95s along with the mean of any vector (`x`) provided as an argument.

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

```r
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)))
    names(CI.vec) <- c("2.5%", "Mean", "97.5%")
    CI.vec
}
```

Explore this function for a few minutes then use it on a few of the vectors in § 6 or on any of the variables in `mydata` (e.g. Heartwood area). Remember, the vector does not have to be named `x` per se, it can have any name since `x` above is only stored locally *within* the function itself (and therefore not a global variable).

```r
CI95(mydata$Heartwood)

##    2.5%    Mean   97.5%
##  28.162  79.005 129.848
```

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 allele frequencies of the population (lets assume no sampling error). These are your bog standard p's and q's from Hardy-Weinberg.

3. Write a function that incorporates the steps you used above in Eqn (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).

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 to explore areas of your own interest (BinomialFun?).

```r
NormFun <- function(n=1000, mu=400, sdv=25, brks=25) {
   Y <- rnorm(n, mu, sdv)
   hist(Y, main="", xlab="", freq= FALSE, col= "gray88", breaks= brks)
   curve(dnorm(x, mu, sdv),
         col="navy", add=TRUE)
}
```

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 § 9. You will be doing this often in § 12 below.

12 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 `deSolve` package since we will be relying heavily upon `ode()` in this section. This function is the workhorse for ODEs and is peculiar in that it, as mentioned briefly in § 11, takes a function as one of its arguments.

- `ode`'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)` where the `item1` is the vector of derivatives calculated at the current time and system state (see below).

Here are two examples to get you started:

12.1 Logistic growth

We will start with logistic growth partly because it has only one dependent variable (and 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 `ode` a function which describes the model Eqn (8), some values for r and K, and some initial conditions (e.g. the initial values of time and population size). `ode` likes its arguments to be as follows:

```
ode(initial_values, time_interval, gradient_function, parameters)
```

... so lets first create our function describing Eqn (8):

```
LogisGrow <- function(t, y, parms) {
    N <- y[1]
    dN <- parms[1] * N * (1 - N/parms[2])
    list(dN)
}
```

Notice our function has the format that `ode` likes. The `t` will be used as a time step by `ode()`, 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 `ode()`:

```
logistic <- as.data.frame(ode(y = c(N = 0.1), times = seq(0, 10,
    by = 0.1), func = LogisGrow, parms = c(r = 0.9, K = 5)))
head(logistic)  # take a peek at the first few rows

##   time       N
## 1  0.0 0.10000
## 2  0.1 0.10921
## 3  0.2 0.11925
## 4  0.3 0.13019
## 5  0.4 0.14210
## 6  0.5 0.15507
```

Notice the start values, time interval, and parameter values were defined directly as arguments within `ode`. The starting value of $N = 0.1$ and r and K have been set to 0.9 and 5 using the `parms=` argument. The time interval has been set using the `seq()` function we learned back in § 6 and goes from $0 \rightarrow 10$ by increments of 0.1 (it's simply a vector of time steps). Lastly, `ode()` will produce a matrix of the solutions with `time` as Col 1 by default. Notice I have asked R to produce a `data frame` instead of a matrix which is slightly more convenient for plotting as in Fig. 11.

```
plot(N ~ time, data = logistic, main = "Simple Logistic Growth Model",
    col = "navy")
```

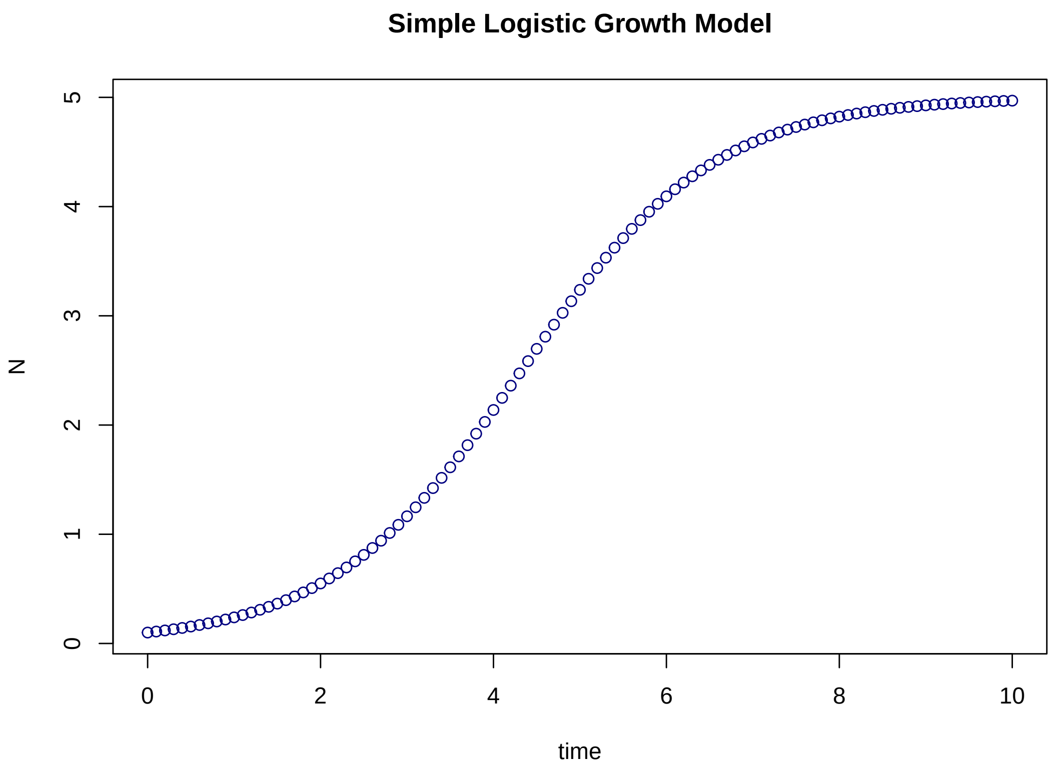


Figure 11: Logistic growth model with $r = 0.9$ and $K = 5$.

12.2 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[25] which stands for the **S**usceptible and **I**nfected classes within a host population (we're not concerned here about parasite populations ☺).

In Eqn (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 (frequency-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 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?[26] 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 used for the SI model above:

```r
tInt <- seq(0, 25, by = 1/2)
pars <- c(beta = 0.75)
Initial <- c(S = 4999, I = 1)
# The function #
SIfun <- function(t, y, parms) {
    with(as.list(c(y, parms)), {
        dS <- -beta * S * (I/(S + I))
        dI <- beta * S * (I/(S + I))
        ODEs <- c(dS, dI)
        list(ODEs)
    })
}
```

The `with()` function is a *magical* functions: the point here is that it enables you to write out your ODE equations in a more familiar way, as opposed to having to refer to parameters with respect to the `parms` vector (i.e. `beta = parms[1]`, . . .) or your state variables with respect to the `y` vector (however, in order to use `with` in this way you must have given appropriate names to your parameter and initial-state vectors, and (of course) the sets of names must not overlap). Compare this construction with the `logistic` ODE function above with I didn't use `with()`.

[25]Loads of assumptions but still quite reasonable for some systems.
[26]Just for fun, google SI model and see what you get ☺

```r
SIout <- as.data.frame(ode(Initial, times = tInt, func = SIfun, parms = pars))
head(SIout)

##   time      S      I
## 1  0.0 4999.0 1.0000
## 2  0.5 4998.5 1.4549
## 3  1.0 4997.9 2.1165
## 4  1.5 4996.9 3.0789
## 5  2.0 4995.5 4.4786
## 6  2.5 4993.5 6.5136
```

```r
par(mfrow = c(1, 2))
plot(S ~ time, data = SIout, ylab = "Number", col = "navy")
points(I ~ time, data = SIout, col = "darkred")
legend("right", bg = "gray95", c("S", "I"), pch = 1, col = c("navy",
    "darkred"))
plot(I ~ S, data = SIout, ylab = "Susceptible", xlab = "Infected")
```

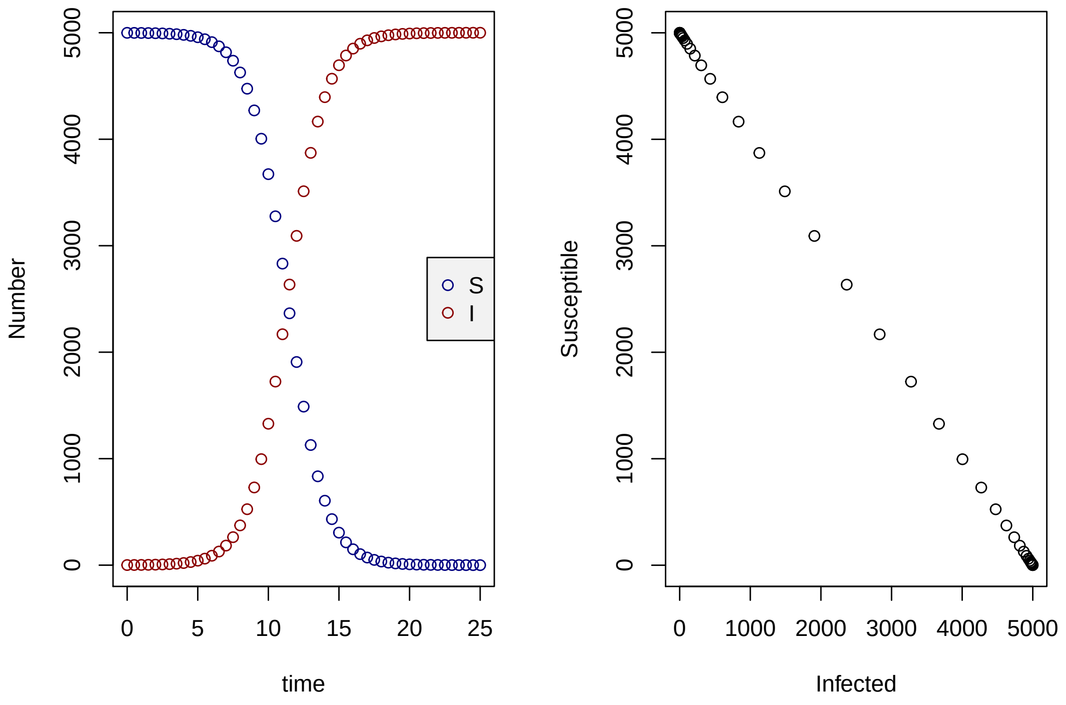


Figure 12: Results of the simple SI model system of ODEs of Eqn (9).

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 `ode`. 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've included my summary plot of the system (Fig. 12) but I'll leave it to you to code for yourself using the matrix `SIout` (since S and 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?

Exercises

1. Now it is your turn to expand on Eqn (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.[27]

   i) 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.

   ii) 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 and 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 (*i*) while making predictions regarding the behaviour of the model. What happens when there is more genetic variation in the population?

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

13 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, and 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 § 5. We were curious about a relationship we discovered in Fig. 2c, now we get to test whether this putative 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(Heartwood ~ dbh, data = mydata)
```

Don't close this plotting window, you will be using it in a moment (if you did, simply re-plot it using the ↑ key). Now actually perform the regression analysis using a linear model via the `lm()` function and give the object the 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()`. Do this and see if you can reproduce Fig. 13 without using the provided code. Can you use the various aesthetic arguments of `plot()` to reproduce it accurately?

```
summary(fit)

##
## Call:
## lm(formula = Heartwood ~ dbh, data = mydata)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -104.8  -30.8    5.2   34.9  105.2
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -147.66      29.61   -4.99  9.6e-05 ***
## dbh            13.18       1.57    8.38  1.3e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 53.8 on 18 degrees of freedom
## Multiple R-squared: 0.796,Adjusted R-squared: 0.785
## F-statistic: 70.2 on 1 and 18 DF,  p-value: 1.26e-07
##
```

```
plot(Heartwood ~ dbh, data= mydata,
   main= "Heartwood Area vs. DBH",
   ylab= expression(paste(plain(HeartwoodArea)," ",(cm^2))),
   xlab= "Diameter at breast height (dbh)",
   pch= 19, col= 1)
grid(NA,NULL, lty= 4)
abline(fit, col= 2, lty= 4, lwd= 2)
legend("topleft", legend=c("lm(fit)"), col= 2, lty= 4, bg= "gray85", box.lty=0)
```

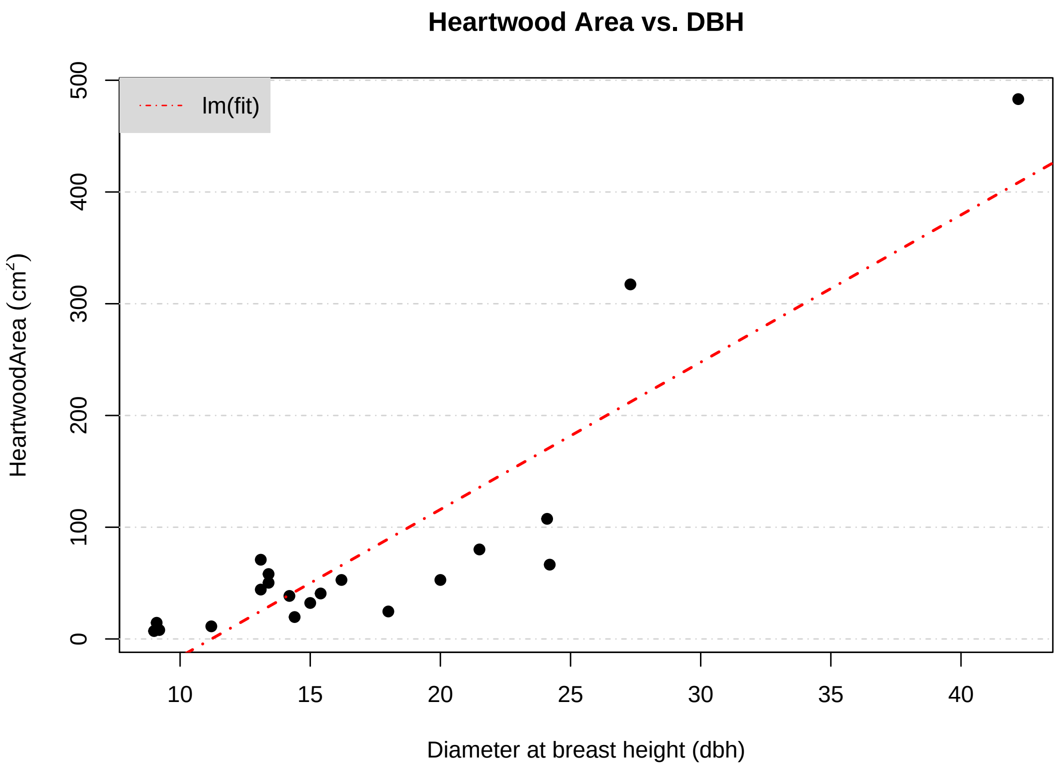


Figure 13: Graphical summary of the linear regression, `lm()`, performed on our tree allometric data (dbh vs. heartwood area).

In general, R provides *accessor* methods, such as `coef`, for extracting the information from complicated objects like linear model fits. You can see what methods are available for a given class, such as `lm` (`class(fit)` tells you what class you should be inspecting), by typing `methods(class="lm")`. If you try this[28], you'll see that there is (for example) a `residuals.lm`, telling you that you can type `residuals(fit)` to extract the residuals (and possibly `?residuals.lm` for more information).

It is always best to use the accessor methods (`coef`, `residuals`, etc.) if you can. However, sometimes you have to peek under the hood and see all the underlying components by typing `attributes(fit)`.

```
attributes(fit)

## $names
```

[28]Note that there is a bug in the current version of the `gdata` package, which the `gplots` package loaded automatically; in order to make `methods(class="lm")` work you'll first have to `detach("package:gplots",unload=TRUE); detach("package:gdata",unload=TRUE)` (and later reload them if you need them again).

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

coef(fit)

## (Intercept)         dbh
##    -147.658      13.178
```

You can then extract the individual components via `fit$name`, where `name` is the name of the bit you want. Using this syntax, convince yourself that you get the same values for the residuals by extracting the component with `$` or using the accessor method.

Lastly, you can manipulate these objects or save them, perhaps for use in other analyses. What does `coef(fit)[1]` produce? And `coef(fit)[2]`?

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 and 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 and y
`t.test`	One and 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 and 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

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 and 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 and fall/winter two into levels of either *Warm* or *Cool*. There should be 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.

3. Based on Fig. 2 it seems as though species differ in sapwood depth. Test this hypothesis using a simple ANOVA. Then examine other continuous variables via boxplots, look for differences by species (or season), and test for statistical significance. Come up with a few of your own questions and hypotheses, then test them.

4. For any of the comparisons above, produce a barplot including error bars (SEMs or CI95s).[29] First try for yourself using the function you created in § 11, and adding the appropriate lines the the barplot. If you run into trouble I have included a function in the script file `R Tutorial Script.R` for creating barplots with error bars called `BarplotBars()` along with an example in TextBox 1. Alternatively, you can "cheat" (I do it all the time) and use the function `barplot2()`[30] which can simplify the process.

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 (usually via loops), write your own functions, and carry out basic statistical analyses. You will rely heavily on these tools over the next few days during the **2012 EEID Workshop**.

[29]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.

[30]Within the gplots package.

Tutorial Contributors

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

Ben Bolker, Depts. of Math & Stats and Biology, McMaster University, Hamilton, Ontario.

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.

Last updated: 19 May 2012

Box 1 Here is the sample code for producing error bars for the exercise in § 13.4 above. You must *first* load the function for making error bars, `BarplotBars()`, which is provided in the R code script file `R_cheat_script.R`.

```
##     Barplots with Error Bars    #
BarplotBars()   # load this function first



##         Produce necessary arguments for error.bars
f <- factor(rep(LETTERS[1:4],each=20))
x <- runif(80)
data <- data.frame(x,f)
N <- as.numeric(table(data$f))
FactorMeans <- tapply(data$x, data$f, mean)
sd <- tapply(data$x, data$f, sd)
sem <- sd/sqrt(N)
ci <- (FactorMeans + (sem * 1.96)) - (FactorMeans - (sem * 1.96))
labels <- as.factor(levels(data$f))

##        Make the Barplot with Error Bars (SEMs & CIs)       #
par(mfrow=c(1,2))
BarplotBars(FactorMeans, sem, labels, axis=2)
title("Barplot with SEM bars")
BarplotBars(FactorMeans, ci, labels, bcol=c(1,8), lcol=2, type=4, w=50)
title("Barplot with CI95 bars")

##        Points with Error Bars
require(gplots)
f <- factor(rep(LETTERS[1:4], each=20))
x <- runif(length(f))
data <- data.frame(x,f)
N <- as.numeric(table(data$f))
Means <- tapply(data$x, data$f, mean)
sd <- tapply(data$x, data$f, sd)
sem <- sd/sqrt(N)
ci <- (FactorMeans + (sem * 1.96)) - (FactorMeans - (sem * 1.96))
Treatments <- c(A=1,B=2,C=3,D=4)

par(mfrow=c(1,2))
plotCI(Treatments, Means, uiw=sem, pch= 19, xaxt="n")
   axis(1, 1:length(Treatments), LETTERS[1:length(Treatments)])
plotCI(Treatments, Means, uiw=ci, pch= 19, xaxt="n")
   axis(1, 1:length(Treatments), LETTERS[1:length(Treatments)])
```

Box 2 The R code below was used to project the imaginary population described in § 10.5 while exploring the `matplot()` function. The population projections are stored in the files `Pop6C.csv`, `Pop6C.R`, and `Pop6C.Rdata` and are plotted in Fig. 10. By the way, this is an example of a stage-structured *SI* model in discrete time.

```
##       Used for creating Pop6C.csv

require(popbio)
s1 = 0.97
s2 = 0.08
s3 = 0.93
s4 = 0.06
s5 = 0.99
FA = 0.14
FJ = FA * 0.66
B = 0.015
F.sel = 0.12
S.sel = 0.9
I.pop = c(20, 14, 9, 25, 15, 19)
#
PM <- matrix(c(s1*(1-B), FJ, FA, 0, FJ*F.sel, FA*F.sel,
               s2*(1-B), s3*(1-B), 0, 0, 0, 0,
               0, s4*(1-B), s5*(1-B), 0, 0, 0,
               s1*B, 0, 0, s1*S.sel, 0, 0,
               s2*B, s3*B, 0, s2*S.sel, s3*S.sel, 0,
               0, s4*B, s5*B, 0, s4*S.sel, s5*S.sel),
       nrow=6, ncol=6, byrow=TRUE)
pop.projection(PM, I.pop, iterations= 20)
popdata <- pop.projection(PM, I.pop, iterations= 20)$stage.vectors
write.csv(round(t(popdata), 2), file= "Pop6C.csv")
```

Box 3 The following R code was used to create the figure on the title page of the tutorial (it also shows several variations on this theme). The `curve3d()` is a plotting function within the emdbook package. The `rgl` environment opens a special interactive graphics window allowing the user to rotate the image in 3D – very cool!

```
x <- seq(-10, 10, length = 50)
y <- seq(-10, 10, length = 50)

rotsinc <- function(x, y) {
   sinc <- function(x) {
      y <- sin(x)/x
      y[is.na(y)] <- 1; y}
   10 * sinc(sqrt(x^2+y^2))
}
z <- outer(x, y, rotsinc)
##  using persp()
persp(x, y, z, theta = 30, phi = 30, expand = 0.5, ltheta= 0,
   shade=0.4, col = "navy", axes=FALSE)
persp(x, y, z, theta = 30, phi = 30, expand = 0.5, col = "navy",
   ltheta = 120, shade = 0.5, ticktype = "detailed", xlab = "X",
   ylab = "Y", zlab = "Z")
##      using curve3d()
require(emdbook)
curve3d(outer(x, y, rotsinc), from= c(-10,-10), to= c(10,10),
        zlab="Z", ylab="Y", xlab="X", col="gray35")  ## with persp
curve3d(outer(x, y, rotsinc), from= c(-10,-10), to= c(10,10),
        zlab= "Z", ylab= "Y", xlab= "X", sys3d= "rgl", col= "navy")
curve3d(outer(x, y, rotsinc), from= c(-10,-10), to= c(10,10), zlab="", ylab="",
        xlab="", sys3d= "rgl", col= rainbow(40), axes= FALSE, box= FALSE)
```

Box 4 Summary of some critical and useful functions in R.

Function	**Description**
`ls`	lists contents of R workspace/global environment
`rm`	removes objects from R workspace
`save`	save selected objects
`+,-,*,/,^`	arithmetic operators
`%*%`	matrix multiplication
`t`	matrix transpose
`solve`	matrix inverse (and solving linear equations)
`c`	combines (concatenates) objects, simplest way to make vectors
`seq`	creates vectors that are regular sequences
`rep`	replicates vectors
`length`	returns length of a vector
`sum`	returns the sum
`mean`	returns the mean
`median`	returns the median
`sd`	returns the standard deviation ($n-1$ in denominator)
`min`	returns minimum
`max`	returns maximum
`sort`	sort a vector (rearranges the vector in order)
`order`	returns indices of vectors that will order them
`rank`	returns rank of each element in vector
`==`, `<`, `>`	comparison operators
`<=`, `>=`, `!=`	
`|`, `&`	`OR`, `AND`
`is.na`	tests for missing value `NA`
`which`	does logical comparison and indicates which elements are `TRUE` that is, gives the `TRUE` indices of a logical object
`any`	does logical comparison returns 1 (`TRUE`) if any of the comparisons are `TRUE`, i.e. is at least one of the values true?
`exp`	returns `e` to that power
`log`	returns natural logarithm (to the base e)
`log10`	returns logarithm (to the base 10)
`sqrt`	returns square root
`table`	does frequencies and cross-tabs
`help`	help page on specified function
`cbind`	combine by columns
`rbind`	combine by rows
`matrix`	create a matrix
`vector`	create a vector
`nrow`	number of rows in an array or data frame
`ncol`	number of columns in an array or data frame
`dim`	dimensions of an array or data frame
`array`	create an array

Box 5 Summary of some critical and useful functions in R.

Function	**Description**
`is.vector`	answers the question, is this a vector `TRUE` or `FALSE`
`as.vector`	attempts to coerce object into a vector
`read.table`	reads data from a text file
`read.csv`	reads data from a text file with comma separated data
`write.table`	writes a data frame to a text file
`is.data.frame`	tests object to see if it is data frame
`as.data.frame`	coerces object into data frame
`is.factor`	tests object to see if it is a factor
`as.factor`	coerces object into a factor
`head, tail`	list the first, last six rows
`names`	returns names of elements of object
`colnames`	returns or sets column names of object
`rownames`	returns or sets row names of object
`subset`	select part of a vector, matrix, or data frame
`merge`	merge two data frames
`lm`	multiple linear regression
`glm`	generalized linear regression
`anova`	analysis of variance
`chisq.test`	Pearson's Chi-squared test for count data
`summary`	shows results of various model fitting functions
`predict`	predicted results from model
`hist`	histogram
`boxplot`	box plot
`plot`	scatterplot
`lines`	connects points sequentially with lines (added to a plot)
`segments`	add lines to a plot (between pairs of points)
`text`	add text to a plot
`legend`	add a legend to a plot
`abline`	add a line to a plot by specifying its slope and intercept
	passing an lm object will result in adding the predicted line to the plot
`x11`	open another graphics window (PC)
`pdf`	open a pdf file for recording graphics
`dev.off`	close graphics device
`par(mfrow)`	arranges multiple plots on same page (by row)
`sample`	produces a random sample of the specified values
`set.seed`	sets seed for next random sample (repeat random sample)
`rnorm`	produces a random sample from a normal distribution
`qnorm`	quantiles (percentiles) of normal distribution
`pnorm`	CDF of normal distribution
`dnorm`	PDF of normal distribution
`rbinom`	produces a random sample from a binomial distribution