(R)MBL

Prepared for the Rocky Mountain Biological Laboratory Undergraduate Education Program
by William K Petry

Updated 28 June 2014

1. Tutorial introduction

R is a programming language for statistical analysis and data visualization. R is rapidly growing in popularity in part because R is:

- free
- open-source
- accommodating of new analyses

This tutorial will walk you through the basics of how to use R. Throughout, chunks of code will be displayed and followed by the results of that code. This will look like this:

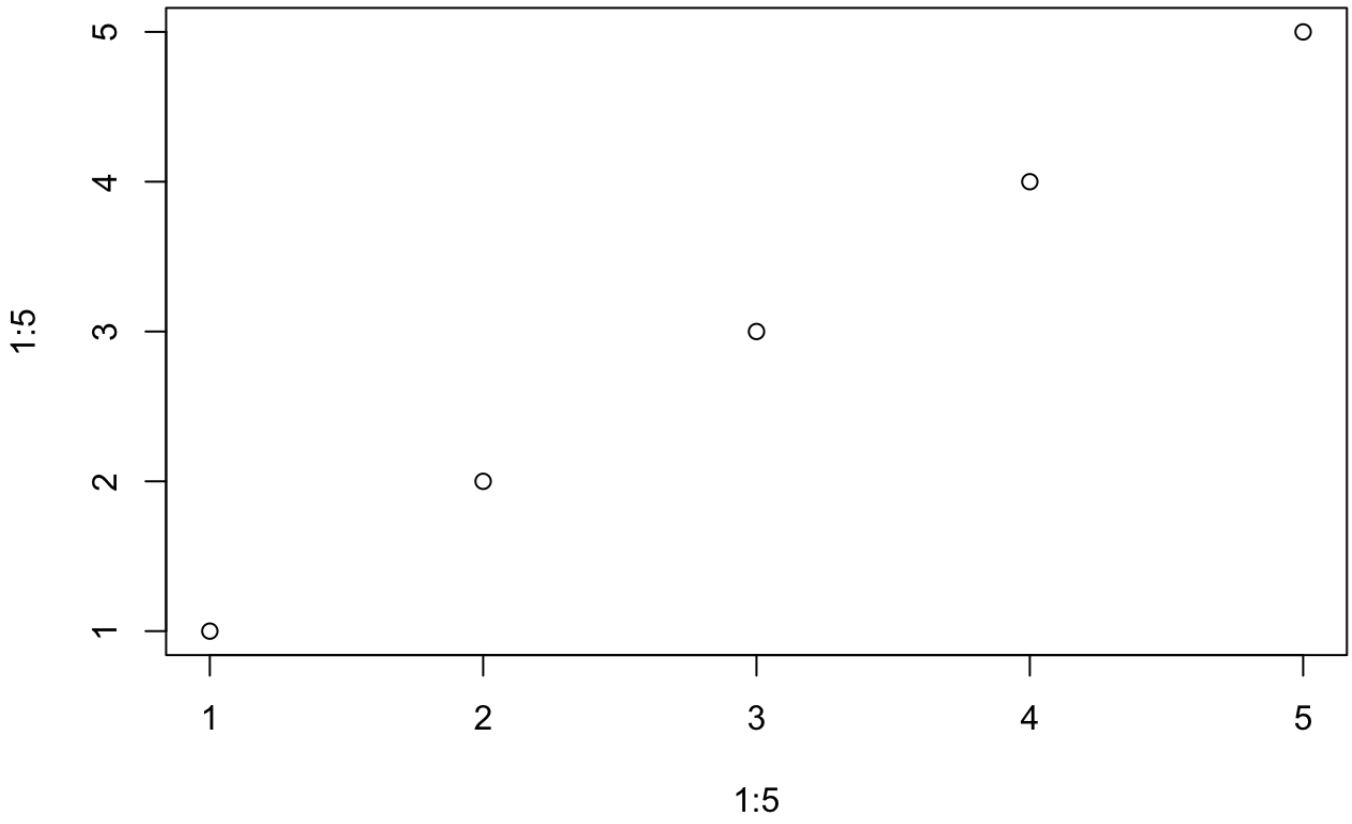
```
# comments and annotations will be preceded with the pound sign
# real code will look like these two simple additions:
2+2
```

```
## [1] 4
```

```
sum(2,2)
```

```
## [1] 4
```

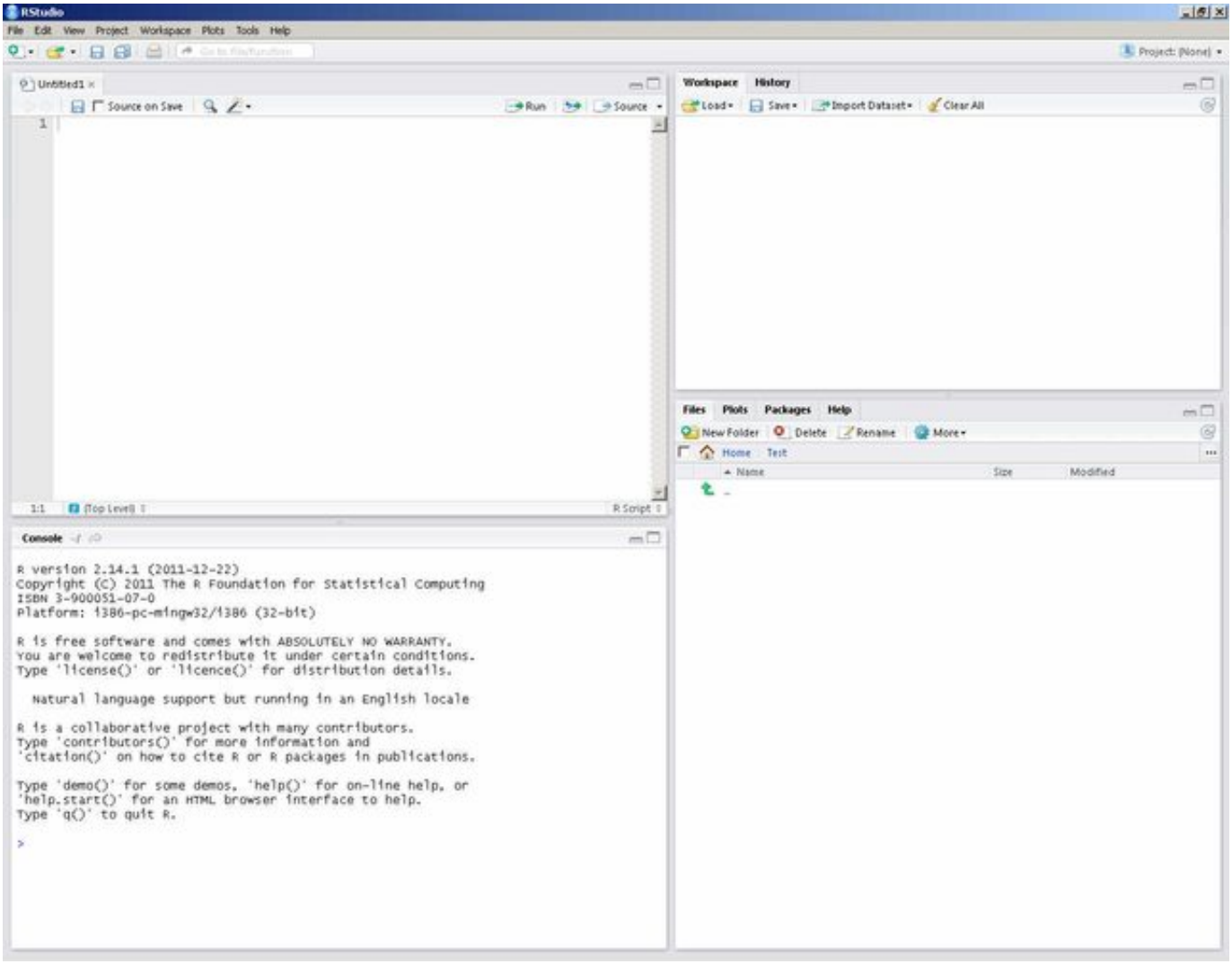
```
# and we can display plots too where the code will be followed by the plot itself
plot(1:5,1:5)
```



2. R and RStudio

R is both a computer language and program. RStudio is a GUI (graphical user interface) for R, meaning that it lets the user interact with R in a way that many find more user-friendly and efficient. For this tutorial, we'll interact with R exclusively through RStudio, emphasizing how the add-ons offered by RStudio .

To begin, download and install R (http://cran.r-project.org) and then RStudio (http://www.rstudio.com/products/rstudio/). Open up RStudio, and you'll see something that looks like this:



NOTE: If you don't have two panels on the left side already, click on the white square with a plus in the top left corner, and select 'R script' from the menu.

2.1 RStudio layout

RStudio has four panes, three of which support multiple tabs. By default, the 'Source' is in the top left and the 'Console' on the lower left. On the right side, the top contains 'Environment' and 'History' tabs. Finally, the lower right contains several miscellaneous tabs. We'll look into each of these one by one.

2.1.1 Console (lower left)

This pane is essentially R with all of the others supporting what goes on here. At start-up, you'll see the welcome text from R reporting which version you're using and offering a few tips. Below that, you'll see a ">" and blinking cursor, meaning that R is ready for our commands.

Commands can be typed directly into the Console, and hitting Enter/Return will run the command. Let's give it a try with a basic – but profound (http://en.wikipedia.org/wiki/Hello_world_program) – command.

```
print("Hello world")
```

Notice that RStudio helps out by automatically inserting the close quote and close parenthesis. If we were sloppy and left one out, R wouldn't know how to handle it. Try deleting the ")".

```
print("Hello world"
```

That `+` sign instead of the `>` means that R is expecting us to finish our statement. We can either type `)` and hit Enter/Return again, or use the Esc key to quit the current operation and start over.

Pro tip: If your console becomes too cluttered, move the cursor down there and press Ctrl + l (or Cmd + l on Mac) to clear everything away.

2.1.2 Source (top left)

Typing commands directly into the console is great for a quick and dirty run of your code. But usually we want to check that our code is working throughout and then save it all in a way that we can easily run everything again. The Source pane serves as a basic text editor with a few extra features that make this integration with the console very, very intuitive.

Let's try that same 'Hello world' script again, but this time copy it into the Source pane instead of the console:

```
print("Hello world")
```

Hitting enter goes to the next line, but doesn't run any of the code (i.e. nothing new shows up in the Console). We can go on to write more lines of code before running. Let's add another statement to the Source pane on line 2:

```
print("R is great!")
```

To run these lines, simply highlight both of them and press the 'Run' button on the top right of the Source pane. Alternatively, we can run one line at a time by highlighting it or skip highlighting by moving the cursor anywhere on the line we want to run and pressing 'Run'.

2.1.3 Environment/History (upper right)

This pane shows a record of all of the data that we've asked R to remember (Environment) and a running list of everything we've typed into the Console or sent to the Console from the Source pane (History). I usually keep my Environment tab displayed to keep track of what I've named all the pieces of data I am using in an analysis, what type of data they are, and how big they are. Like the Console, this too can be cleared entirely by clicking the 'Clear' button or selectively by checking which objects to remove.

Double clicking any of the commands in the History tab will run that command in the Console again. You can also access this list directly from the Console by simply using the up/down arrow keys. This is very useful if you need to correct a typo or re-import data.

2.1.4 Miscellaneous tabs (lower right)

This pane is the catch all for other helpful RStudio features. Files can be used just like Windows Explorer or Mac's Finder to navigate your computer. Plots shows any graphics you produce. Packages lists the add-on code installed and loaded on your computer (more on this later). Help provides a way to browse the help documentation.

3. Having a conversation with R

We've covered the basics of how to tell R to do something, but R forgets everything that we've said right after it returns the output of these commands. This isn't very helpful, especially when data analysis is bound to take several steps to get from data import to the final output. This section covers how to carry on a conversation with R where the output of previous commands is remembered and can be accessed later on.

3.1 Assignment

Everything we want R to remember must be given a name. A simple symbol, `<-`, is used to connect the name (on the side to which the arrow is pointing) and the result of the command (aka object, on the other end of the arrow). Reading this assignment command

```
x <- "Hello world"
```

as a sentence would sound like "x *is set to* Hello world." Go ahead and run this assignment command to set `x` to `"Hello world"`.

We can make vectors containing multiple values by using the concatenation command:

```
y <- c(1,2,3,4,5,6,7,8,9,10)
z <- c("a","b","c","d","e","f")
```

To display the contents of these objects, we can use the `print()` command, or simply type it's name:

```
print(y)
```

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

```
z
```

```
## [1] "a" "b" "c" "d" "e" "f"
```

Note: Some tutorials will offer the alternative of `=` for assigning names to objects. This works just as well in *most* cases, but has the potential to cause confusion for reasons we won't get into here. It's a good idea to get into the habit of using `<-` for all name assignments.

3.2 Object classes and functions

Objects are pieces of data and functions do things to these data and return other objects that hold the output. To use an analogy to language, objects are like subjects and functions are like verbs. Like spoken languages where subjects and their verbs must agree, in R objects must be compatible with the function applied to it.

Objects can be of many different types. We call these different forms classes, and the object's class will limit which functions can be used on it. For example, trying to multiply `"Hello world"` by `3` is pretty nonsensical, and R will return an error if you try. We can easily determine the class of an object like this:

```
class(x)
```

```
## [1] "character"
```

Here, `"character"` means that this object is made up of text. Here are some other basic classes of objects you'll encounter:

Class name	What it holds
character	text
numeric	numbers (can be decimal)
integer	numbers (only integers)
boolean	true or false
factor	categorical data
dataframe	data table (can contain variables of multiple classes)
matrix	a 2-dimensional matrix (data are all of the same class)
array	an n-dimensional matrix (data are all of the same class)
list	a collection of multiple objects of one or more classes

Additionally, each type of analysis will return an object of its own class. For example, an ANOVA will return an object of class `aov` and a simple linear regression will return an object of class `lm`. Unlike spoken languages, applying a function to an object of one class can result in a very produce an object of a very different class (e.g. the data going into an ANOVA or regression is often `numeric`, `factor`, or `dataframe`).

There is one time you're likely to run into problems with an object's class: importing data. When we get to that section of the tutorial, we'll see how to get around any problems with this using coercion. For now, suffice it to say there are lots of different ways that R can store data, functions can usually only work with a subset of these data classes, and functions may return a different class of object than the object to which the function was applied.

3.3 Simple manipulation of numbers

R uses all of the conventional symbols for basic arithmetic and logical operators.

Operation	Symbol in R
Addition	`+`
Subtraction	`-`
Multiplication	`*`
Division	`/`
Exponents	`^` or `**`
Less than	`<`
Greater than	`>`
Less than or equal to	`<=`
Greater than or equal to	`>=`
Equal to	`==`
Not equal to	`!=`

Or	`|`
And	`&`

What's nice about these functions is that they are not just limited to single values. Rather, they can be applied over whole vectors, dataframes, and matrices. For example,

```
# take a look at the vector of integers we've named y
y
```

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

```
# now multiply everything by 3
y_mult <- y*3
y_mult
```

```
##  [1]  3  6  9 12 15 18 21 24 27 30
```

```
# we can also logically test whether something is true across a whole vector
# (useful later to extract parts of a dataset)
y_mult > 7.5
```

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

4. Adding new functions

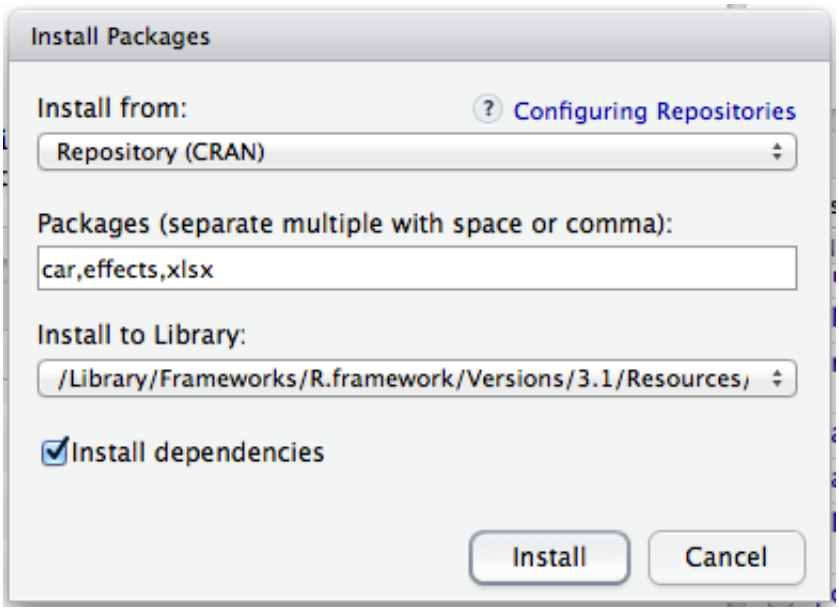
4.1 Packages

R users (aka useRs) building multiple functions will often share their work by publishing a package on CRAN (http://cran.r-project.org (http://cran.r-project.org)). These packages expand the capabilities of R, and there are currently >5000 packages available for download.

Frankly, the best way to sort through all of these is to use Google. A few suggested packages are included at the end of the document in the Appendix.

The first step is to download the package to your computer. On the 'Packages' tab, click 'Install Packages' and type in the name of the package you want. Let's download a couple at once that we'll use later:

- **car**
- **effects**
- **xlsx**



Make sure 'Install dependencies' is checked, and click install. Now these packages are on your computer, but aren't usable just yet. You'll need to click the check box next to the package name on the 'Packages' tab or use the `library()` command.

```
library(car)
```

```
library(effects)
```

```
library(xlsx)
```

You'll see that a few other companion packages are simultaneously loaded. Also, if there are packages with shared names, the last one loaded "owns" that shared name.

4.2 Writing functions

One really great aspect of R is that you can write your own functions. Not everything can be found in a package, and writing a function can save a lot of lines of code.

One function that is strangely absent from the base version of R is the calculation for standard error of the mean (SEM or just SE). Many of you may know that the SE for a sample is calculated as:

$$SE = \frac{s}{\sqrt{n}}$$

where s is the sample standard deviation and n is the sample size. Let's have a look at the syntax of function building:

```
# Let's start by considering three built-in functions:
# 1) sd(x) calculates the standard deviation of the vector x
# 2) sqrt(y) returns the square root of the number y
# 3) length(z) counts the number of elements in the vector z

SE <- function(x){
  # any intermediate calculations and data handling goes here
  # in this case, the function is simple enough to write on one line. We'll wrap it inside th
e
  # return() function so that this function spits out the result of the enclosed calculation.
  return(sd(x)/sqrt(length(x)))
}
```

Note that here we have nested several functions within one another. R will interpret this starting with the inner most function, then applying subsequent functions to the output of the previous function. This is a really powerful feature of the R language that allows you to accomplish multiple steps at once. As a beginner, it may be best to calculate all these steps as intermediates and check their accuracy before putting everything together.

A quick check of our function shows us that it is working correctly:

```
# here's the vector that we want to know the standard error of:
y_mult
```

```
##  [1]  3  6  9 12 15 18 21 24 27 30
```

```
SE(y_mult)
```

```
## [1] 2.872
```

5. Data import

5.1 Dataset description

We'll use a real ecological dataset that has been made publicly available (Kartzinel et al. 2014) for the rest of the tutorial.

Briefly, the experiment was to exclude mammals of different sizes in eastern Kenya between 2008-2013. There were four treatment levels:

- Open (= control)
- Mega (= giraffes and elephants)
- Meso (= everything >40 kg)
- Total (= everything >5 kg)

The researchers replicated each treatment 3 times (called BLOCK in the dataset) at each of 3 sites, except for the control treatment that was only applied once per site. The data we'll use come from a single site and

focuses on a single tree species *Acacia drepanolobium* that is consumed by mammalian herbivores and defended by mutualistic ants. The landscape looks something like this:

5.2 Importing dataset into R

R knows how to understand a number of file types, but proprietary formats like Excel files are not within this repertoire. We'll start by loading a package that will serve as an interpreter between R and Excel.

```
# load the package xlsx that will let R talk to Excel
library(xlsx)
```

```
## Loading required package: rJava
## Loading required package: xlsxjars
```

```
# tell R to go get the data from the appropriate file and sheet
acacia <- read.xlsx("ACDR_survey.xlsx", sheetName = "ACDR_data")
```

Take a look at the "Environment" pane on the top right of RStudio. We can now see that there's an object in there that is holding our data. In the next section, we'll learn how to use R to make sure everything imported correctly and work with the data without going back to Excel.

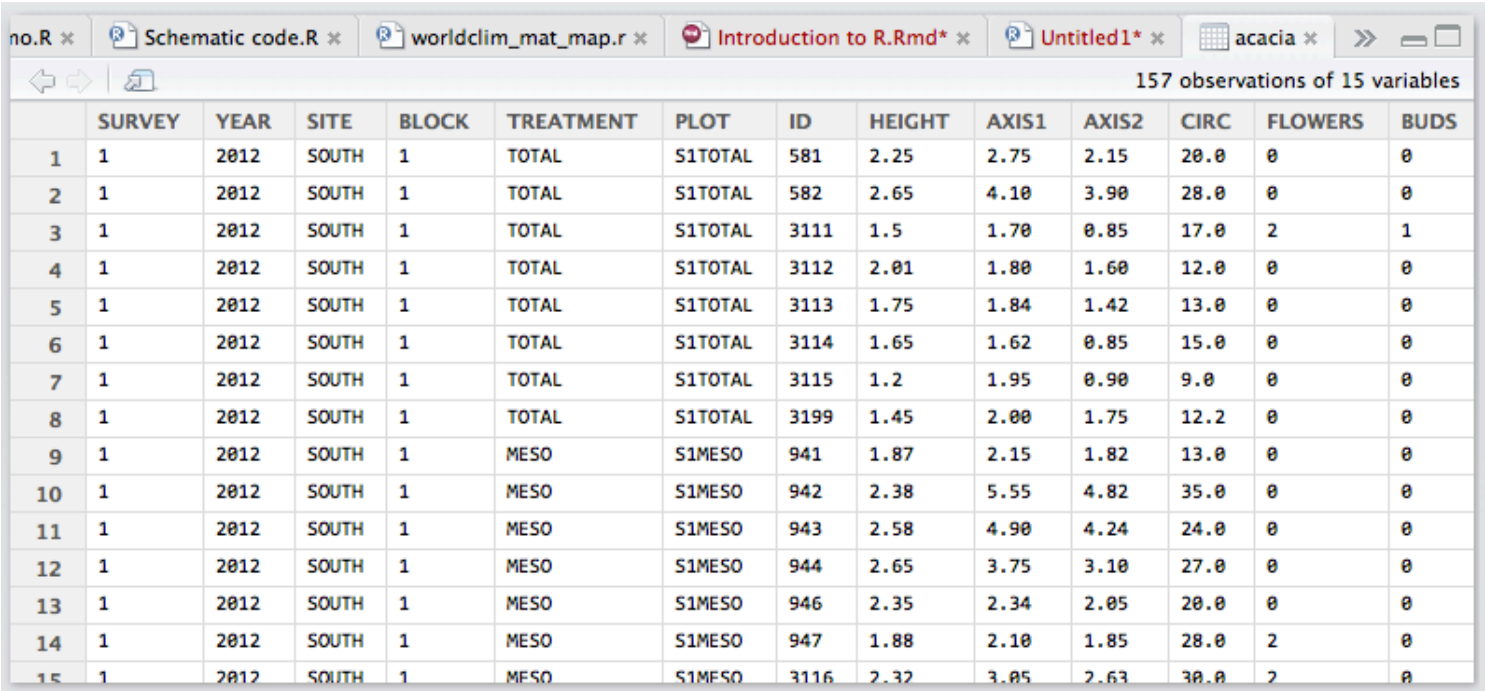
6. Data handling and plotting

In this section, we'll walk through some of the most common tasks that come up right after data are imported and prior to analysis.

6.1 Learning about your data

As is probably clear at this point, R is not a spreadsheet program like Excel: instead of pointing, clicking, and dragging, everything is done through code. That said, R and RStudio have some great built-in tools to learn about your data.

Let's start with the most Excel-like format that RStudio can display. Click on the little table icon associated with the object `acacia` in the "Environment" pane. We can now see the data displayed as a table in the "Source" pane, and RStudio has entered and run the command `View(acacia)` for us to make this data display happen.



	SURVEY	YEAR	SITE	BLOCK	TREATMENT	PLOT	ID	HEIGHT	AXIS1	AXIS2	CIRC	FLOWERS	BUDS
1	1	2012	SOUTH	1	TOTAL	S1TOTAL	581	2.25	2.75	2.15	20.0	0	0
2	1	2012	SOUTH	1	TOTAL	S1TOTAL	582	2.65	4.10	3.90	28.0	0	0
3	1	2012	SOUTH	1	TOTAL	S1TOTAL	3111	1.5	1.70	0.85	17.0	2	1
4	1	2012	SOUTH	1	TOTAL	S1TOTAL	3112	2.01	1.80	1.60	12.0	0	0
5	1	2012	SOUTH	1	TOTAL	S1TOTAL	3113	1.75	1.84	1.42	13.0	0	0
6	1	2012	SOUTH	1	TOTAL	S1TOTAL	3114	1.65	1.62	0.85	15.0	0	0
7	1	2012	SOUTH	1	TOTAL	S1TOTAL	3115	1.2	1.95	0.90	9.0	0	0
8	1	2012	SOUTH	1	TOTAL	S1TOTAL	3199	1.45	2.00	1.75	12.2	0	0
9	1	2012	SOUTH	1	MESO	S1MESO	941	1.87	2.15	1.82	13.0	0	0
10	1	2012	SOUTH	1	MESO	S1MESO	942	2.38	5.55	4.82	35.0	0	0
11	1	2012	SOUTH	1	MESO	S1MESO	943	2.58	4.90	4.24	24.0	0	0
12	1	2012	SOUTH	1	MESO	S1MESO	944	2.65	3.75	3.10	27.0	0	0
13	1	2012	SOUTH	1	MESO	S1MESO	946	2.35	2.34	2.05	20.0	0	0
14	1	2012	SOUTH	1	MESO	S1MESO	947	1.88	2.10	1.85	28.0	2	0
15	1	2012	SOUTH	1	MESO	S1MESO	3116	2.32	3.05	2.63	30.0	2	0

Clicking on cells to edit the data won't work like it does in Excel. This feature is strictly for viewing. We can do the same for smaller datasets by typing the name of the object in the Console:

`acacia`

That's a lot of output. Note that the text is a little too wide for the console, so it's wrapped in an awkward way that makes interpretation difficult. Also, we don't know how R is treating each piece of the dataset. Remember object classes in section 3.2? We'd like to know whether each of these is being read appropriately. For that, we can use the `str` command.

```
str(acacia)
```

```
## 'data.frame':    157 obs. of  15 variables:
##  $ SURVEY   : num  1 1 1 1 1 1 1 1 1 1 ...
##  $ YEAR     : num  2012 2012 2012 2012 2012 ...
##  $ SITE     : Factor w/ 1 level "SOUTH": 1 1 1 1 1 1 1 1 1 1 ...
##  $ BLOCK    : num  1 1 1 1 1 1 1 1 1 1 ...
##  $ TREATMENT: Factor w/ 4 levels "MEGA","MESO",..: 4 4 4 4 4 4 4 4 2 2 ...
##  $ PLOT     : Factor w/ 9 levels "S1MESO","S1TOTAL",..: 2 2 2 2 2 2 2 2 1 1 ...
##  $ ID       : num  581 582 3111 3112 3113 ...
##  $ HEIGHT   : Factor w/ 77 levels "0.67","0.68",..: 66 75 37 57 46 44 17 33 49 70 ...
##  $ AXIS1    : num  2.75 4.1 1.7 1.8 1.84 1.62 1.95 2 2.15 5.55 ...
##  $ AXIS2    : num  2.15 3.9 0.85 1.6 1.42 0.85 0.9 1.75 1.82 4.82 ...
##  $ CIRC     : num  20 28 17 12 13 15 9 12.2 13 35 ...
##  $ FLOWERS  : num  0 0 2 0 0 0 0 0 0 0 ...
##  $ BUDS     : num  0 0 1 0 0 0 0 0 0 0 ...
##  $ FRUITS   : num  10 150 50 75 20 0 0 25 0 50 ...
##  $ ANT      : Factor w/ 6 levels "AB_TP","CM","CN",..: 4 6 6 4 4 5 4 4 6 6 ...
```

The output reports that `acacia` is of class “data.frame” and has 15 columns and 157 rows. Additionally for each of those columns, we can see its name, class, and the first few data entries. Everything looks pretty good, except that the variable “HEIGHT” is being treated as a factor rather than a number. Also, BLOCK should probably be treated as a factor. Let’s take a look at why this imported this way and fix it in section 6.2.

6.2 Subsetting and selective editing

Rather than looking at the whole dataset, let’s zoom in on the problem variable “HEIGHT”. The syntax to pull out a subset of the data uses the `$` symbol between the name of the object holding the data and the part to be extracted.

```
acacia$HEIGHT
```

```
##   [1] 2.25 2.65 1.5  2.01 1.75 1.65 1.2  1.45 1.87 2.38 2.58 2.65 2.35 1.88
##  [15] 2.32 2.39 2.2  1.05 2    1.28 dead 1.4  1.9  1.75 1.8  2.7  2.02 1.9
##  [29] 1.85 1.65 1.4  2.5  2.05 2.26 2.13 1.8  1.85 1.5  1.87 1.58 2.05 1.75
##  [43] 1.49 1.28 1.49 1.07 1.48 1.25 1.41 1.6  1.2  1.49 1.5  1.65 1.13 1.25
##  [57] 1.1  2.2  1.45 1.6  1.55 1.5  1.03 2.14 1.2  1.05 1.8  1.2  1.75 1.45
##  [71] 1.17 2.15 1.7  1.98 1.26 1.11 1.14 1.26 1.3  1.29 1.31 1.15 1.87 1.47
##  [85] 1.05 2.1  1.99 1.42 1.5  1.06 1.49 1.8  1.93 1.2  1.65 1.52 1.43 1.25
##  [99] 1.88 1.03 1.1  1.4  1.05 1.18 1.4  1.37 1.32 1.55 1.3  1.24 1.5  1.65
## [113] 2.17 1.28 1.07 0.67 0.68 1.87 1.35 1.75 1.75 1.64 1.42 dead 0.9  dead
## [127] 1.8  2.47 2.15 1.7  1.9  1.95 1.8  1.4  1    1.75 1.28 1    1.45 1
## [141] 1.03 1.51 1.17 1.33 1.3  1.13 1.58 1.06 1.05 1.45 1.15 1.42 1.02 1.4
## [155] 1.45 1.95 dead
## 77 Levels: 0.67 0.68 0.9 1 1.02 1.03 1.05 1.06 1.07 1.1 1.11 1.13 ... dead
```

Entries 21, 124, 126, and 157 are causing the problem because they are text: “dead”. R can treat any characters as text, but only numbers can be treated as numbers. R defaults to treating everything in `acacia$HEIGHT` as categories.

We could return to Excel, delete “dead” in that column, then re-import the data. But there’s a better way to do this just by using R. We’ll set those words to `NA` which will hold the place in the table without affecting calculations on the other numbers. Essentially this is a “Find & Replace” function.

```
# we can FIND the entries that are "dead" using brackets
acacia$HEIGHT[acacia$HEIGHT == "dead"]
```

```
## [1] dead dead dead dead
## 77 Levels: 0.67 0.68 0.9 1 1.02 1.03 1.05 1.06 1.07 1.1 1.11 1.13 ... dead
```

```
# to REPLACE them, just assign them to be something different, here NA
acacia$HEIGHT[acacia$HEIGHT == "dead"] <- NA
```

A quick look at `acacia` shows that all the “dead” entries have been replaced with `NA`, but `str(acacia)` shows the same problem as before. Because R initially read `acacia$HEIGHT` in as a factor, we need to coerce it into class numeric.

```
acacia$HEIGHT <- as.numeric(acacia$HEIGHT)
# now we can see that str(acacia) shows the problem is fixed
str(acacia)
```

```
## 'data.frame':    157 obs. of  15 variables:
##  $ SURVEY   : num  1 1 1 1 1 1 1 1 1 1 ...
##  $ YEAR     : num  2012 2012 2012 2012 2012 ...
##  $ SITE     : Factor w/ 1 level "SOUTH": 1 1 1 1 1 1 1 1 1 1 ...
##  $ BLOCK    : num  1 1 1 1 1 1 1 1 1 1 ...
##  $ TREATMENT: Factor w/ 4 levels "MEGA","MESO",..: 4 4 4 4 4 4 4 4 2 2 ...
##  $ PLOT     : Factor w/ 9 levels "S1MESO","S1TOTAL",..: 2 2 2 2 2 2 2 2 1 1 ...
##  $ ID       : num  581 582 3111 3112 3113 ...
##  $ HEIGHT   : num  66 75 37 57 46 44 17 33 49 70 ...
##  $ AXIS1    : num  2.75 4.1 1.7 1.8 1.84 1.62 1.95 2 2.15 5.55 ...
##  $ AXIS2    : num  2.15 3.9 0.85 1.6 1.42 0.85 0.9 1.75 1.82 4.82 ...
##  $ CIRC     : num  20 28 17 12 13 15 9 12.2 13 35 ...
##  $ FLOWERS  : num  0 0 2 0 0 0 0 0 0 0 ...
##  $ BUDS     : num  0 0 1 0 0 0 0 0 0 0 ...
##  $ FRUITS   : num  10 150 50 75 20 0 0 25 0 50 ...
##  $ ANT      : Factor w/ 6 levels "AB_TP","CM","CN",..: 4 6 6 4 4 5 4 4 6 6 ...
```

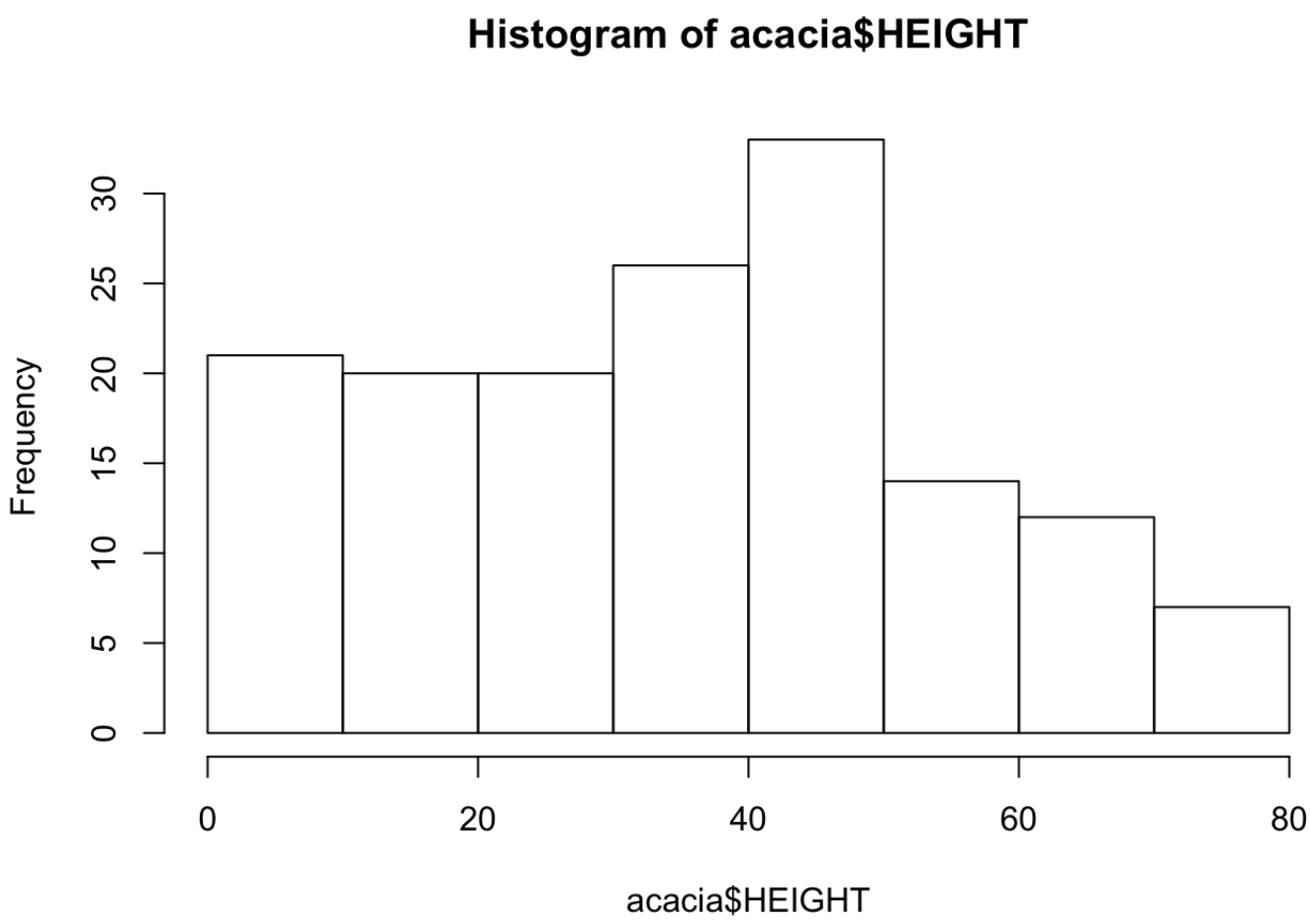
Coercing a number into a factor doesn’t require this last step because numbers can be treated as text without any problems:

```
acacia$BLOCK <- as.factor(acacia$BLOCK)
```

6.3 Summarizing and visualizing raw data

Looking at data as numbers isn't a good way for most people to understand patterns in their data. Let's take a look at a few plots and tables of some of the variables in `acacia`. Let's build a histogram of `acacia$HEIGHT`.

```
hist(acacia$HEIGHT)
```

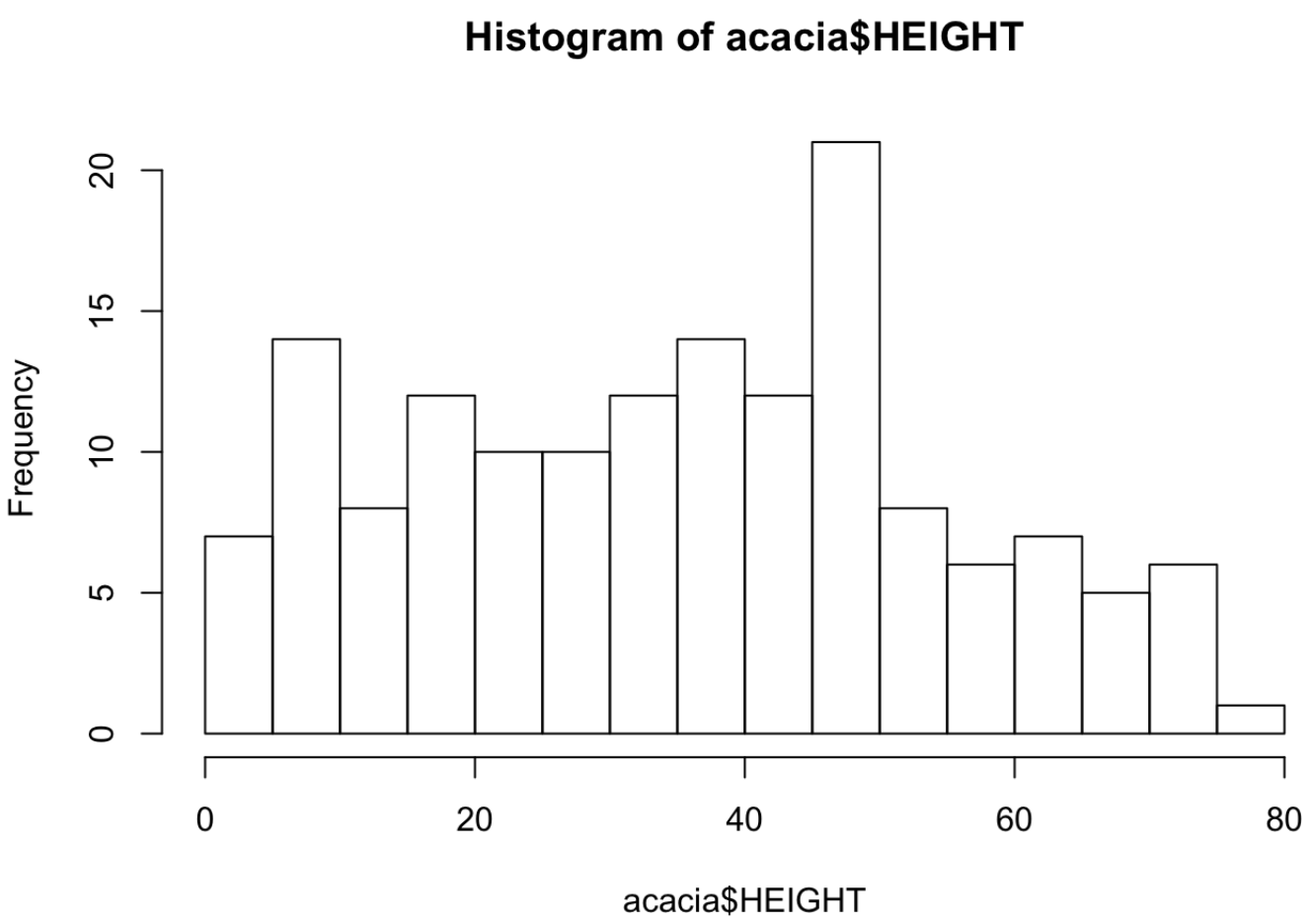


Those bins are pretty chunky and would be worth examining in greater detail. Almost all functions in R have some options associated with them. To see what these are, let's open the help file using the `?` command:

```
?hist
```

R documentation can be pretty heavy sledding to interpret, so at the beginning Google may be a better friend for troubleshooting. The option that we want is "breaks" that by default is "Sturges", but we can set to any number we like.

```
hist(acacia$HEIGHT, breaks = 15)
```



Histograms are great for continuous data, but tables are more useful for categorical data. Let's look at the breakdown of ant species found living on the trees.

```
table(acacia$ANT)
```

```
##
## AB_TP    CM    CN    CS     E    TP
##     1    12     2    56     2    80
```

7. Basic statistical tests

In this section, we'll test a simple hypotheses using the example dataset:

H_a: Herbivore exclusion increases tree growth.

7.1 Checking assumptions

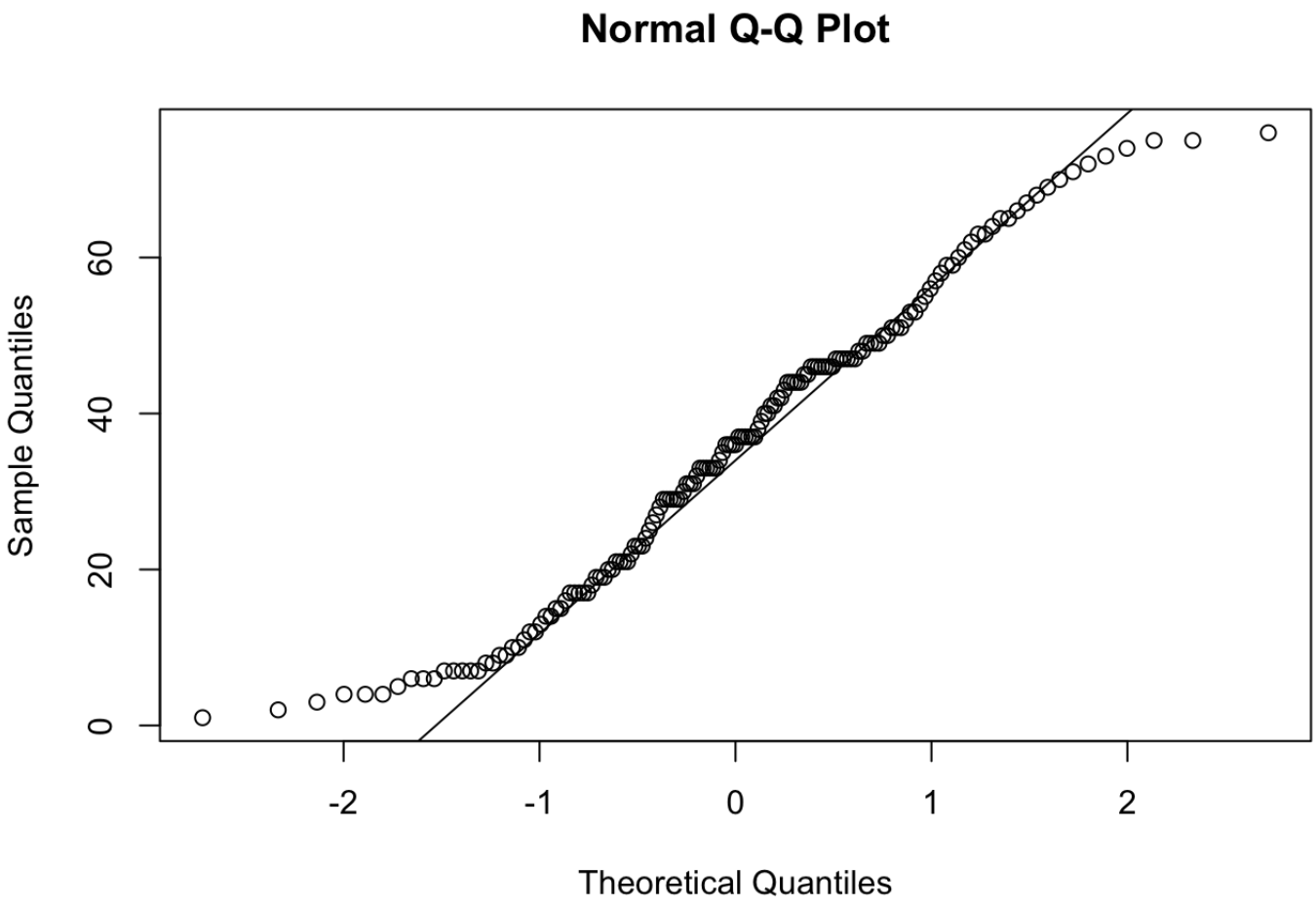
All statistical tests make some sort of assumption, some of which involve how the data were collected while others involve characteristics of the collected data. For the latter, R has some functions to test whether our data are suitable for use in a given statistical test. For parametric analyses like ANOVA and regression, the two assumptions we should check are:

1. normality of the residuals – the amount each replicate deviates from the mean follows the normal

distribution
2. homoscedasticity – the data in each group have equal variances around the mean

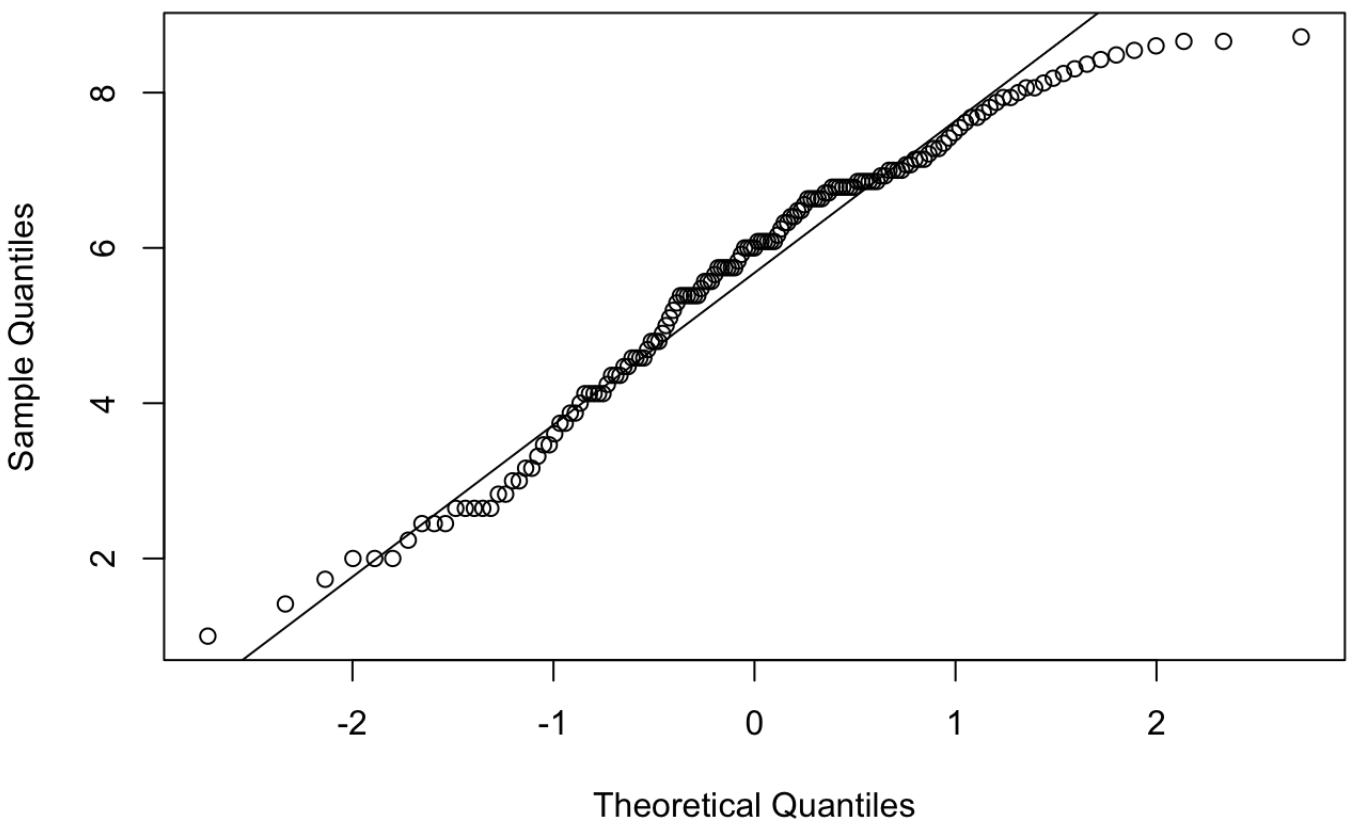
For the first, we can inspect the data visually using a quantile plot of the response variable (though other more quantitative methods also exist, see the function `shapiro.test()`).

```
# the first function plots the data, the second shows the line on which points should fall
# if the data are normally distributed
qqnorm(acacia$HEIGHT)
qqline(acacia$HEIGHT)
```



This isn't too big a departure from normality, but we can try out a data transformation (specifically, the square root transformation) to try to reel in these residuals.

```
# make a new column in "acacia" and fill it with the transformed data
acacia$HEIGHT.sqrt <- sqrt(acacia$HEIGHT)

# now make a new quantile plot
qqnorm(acacia$HEIGHT.sqrt)
qqline(acacia$HEIGHT.sqrt)
```



The new quantile plot isn't perfect, but the analyses we'll use are pretty robust to a violation of this magnitude.

A quick check of the homogeneity of variance shows that we're going to have some problems with the control plots: there were very few OPEN plots and the trees were destroyed in all but one.

`bartlett.test(HEIGHT.sqrt~TREATMENT, data = acacia)`

We can check on all the other treatments by subsetting the data. While it is far from ideal to leave the control treatment out of the analysis, we can still ask questions about how the different exclusion treatments compare relative to each other. We'll subset the data to exclude the OPEN treatment:

```
# can you trace the logic of this subset (hint: check the table in section 3.3)?
acacia_exclude <- acacia[acacia$TREATMENT != "OPEN",]
bartlett.test(HEIGHT.sqrt~TREATMENT, data = acacia_exclude)
```

```
##
##  Bartlett test of homogeneity of variances
##
## data:  HEIGHT.sqrt by TREATMENT
## Bartlett's K-squared = 1.301, df = 2, p-value = 0.5218
```

Everything looks ok from the Bartlett's test. Note that the `data = acacia_exclude` tells R to look for the objects `HEIGHT.sqrt` and `TREATMENT` in the object `acacia_exclude` (not to mention saving some typing).

7.2 t-tests

One of the most basic tests to compare means between two groups is to use a t-test. This test has a long and interesting history (involving Guinness beer!), but is still widely used.

Let's test for a difference between the MEGA and TOTAL herbivore exclusion plots.

```
# let's start by subsetting the data to extract only these two parts
MEGA.HEIGHT <- acacia$HEIGHT.sqrt[acacia_exclude$TREATMENT == "MEGA"]
TOTAL.HEIGHT <- acacia$HEIGHT.sqrt[acacia_exclude$TREATMENT == "TOTAL"]

# now open the help file so we can see the order to enter commands
?t.test

# we know our variances are equal, so we can set that option to TRUE for a more powerful test
t.test(MEGA.HEIGHT, TOTAL.HEIGHT, var.equal = TRUE)
```

```
##
##  Two Sample t-test
##
## data:  MEGA.HEIGHT and TOTAL.HEIGHT
## t = -1.462, df = 134, p-value = 0.146
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -1.776  0.266
## sample estimates:
## mean of x mean of y
##     4.867     5.622
```

7.3 Linear models

Testing each of these treatment comparisons is statistically problematic not to mention tedious. Let's use an ANOVA to do a single test across all treatments.

```
# we can use an optional data argument to avoid writing "acacia$" over and over again
growth.anova <- lm(HEIGHT.sqrt ~ TREATMENT, data = acacia_exclude)
anova(growth.anova)
```

```
## Analysis of Variance Table
##
## Response: HEIGHT.sqrt
##            Df Sum Sq Mean Sq F value Pr(>F)
## TREATMENT   2     39   19.42    6.32 0.0023 **
## Residuals 149    458    3.07
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

This is a very simplistic model to understand a complicated system. Let's add in some more information on these trees as covariates. Specifically, let's account for the BLOCK the tree grows in and also the ANT in residence on the tree.

```
growth.anova2 <- lm(HEIGHT.sqrt ~ TREATMENT + BLOCK + ANT, data = acacia_exclude)
# load a package that will calculate the right p-value when there are multiple factors in the
 model
library(car)
Anova(growth.anova2, type = 3)
```

```
## Anova Table (Type III tests)
##
## Response: HEIGHT.sqrt
##             Sum Sq  Df F value  Pr(>F)
## (Intercept)     47   1   15.76 0.00011 ***
## TREATMENT       10   2    1.70 0.18600
## BLOCK           17   2    2.88 0.05916 .
## ANT             10   5    0.69 0.63411
## Residuals      427 142
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

We can see that TREATMENT is no longer significant, essentially all the TREATMENT differences among trees are really caused by differences among BLOCK.

7.4 Least square (marginal) means

Often we want to report the means of each treatment without including the effect of other treatments. You may hear these means referred to as least square or marginal means. We'll use a package called `effects{}` to calculate these values appropriately from the regression we just ran.

```
# load the package that calculates least square means
library(effects)
```

```
## Loading required package: lattice
## Loading required package: grid
## Loading required package: colorspace
##
## Attaching package: 'effects'
##
## The following object is masked from 'package:car':
##
##     Prestige
```

```
# Let's look at the effect of BLOCK while controling for everything else.
summary(effect("BLOCK", mod = growth.anova2))
```

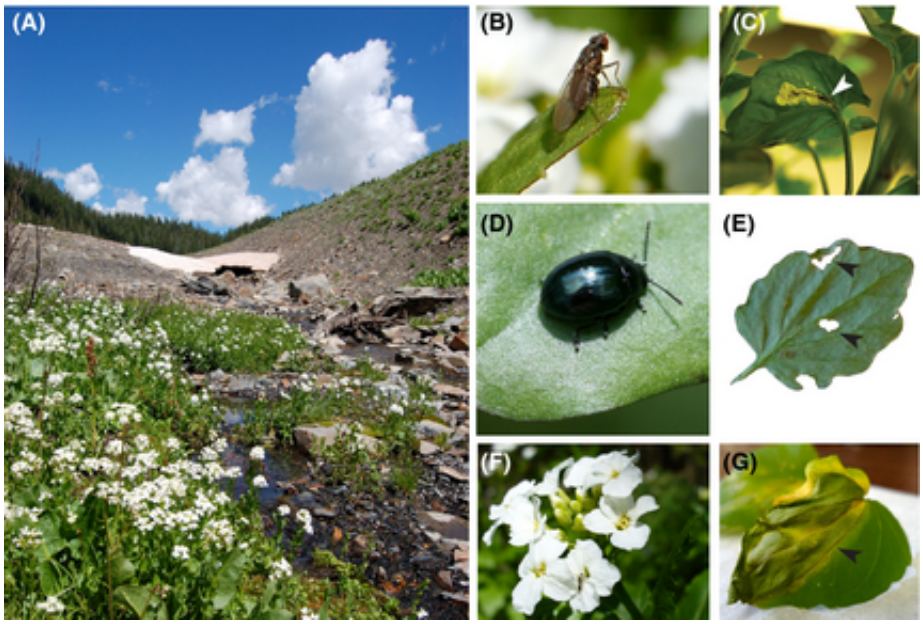
```
##
##  BLOCK effect
## BLOCK
##     1     2     3
## 6.630 5.704 5.127
##
##  Lower 95 Percent Confidence Limits
## BLOCK
##     1     2     3
## 5.657 5.343 4.430
##
##  Upper 95 Percent Confidence Limits
## BLOCK
##     1     2     3
## 7.603 6.065 5.824
```

```
# Compare these to the raw means using the aggregate() function:
aggregate(HEIGHT.sqrt ~ BLOCK, acacia_exclude, mean)
```

```
##   BLOCK HEIGHT.sqrt
## 1     1       7.058
## 2     2       5.626
## 3     3       5.106
```

8. Putting it all together

Noah Whiteman's lab here at the RMBL recently published a study where they were trying to understand how the macro (=herbivores) and micro (=bacteria) enemies of plants interact through their effects on plant defense systems. The work took place at Emerald Lake just up valley from RMBL. Here are the main players in the system (image and caption from Humphrey et al. 2014):



Overview of study organisms and types of leaf damage. (A) Subalpine study population of bittercress near the Rocky Mountain Biological Laboratory from which the leaves in this study were sampled (near outflow of Emerald Lake, elevation 3182 m). (B) *Scaptomyza nigrita* adult female. (C) *S. nigrita* larva mining bittercress leaf (white arrow indicates larva). (D) *Phaedon* sp. chrysomelid (leaf) beetle (*Phaedon aeuruginosa* depicted; photograph by Sandy Rae). (E) *Phaedon* sp. damage (black arrows indicate removed leaf area). (F) Bittercress inflorescence. (G) Chlorosis in a bittercress leaf (arrow indicates border between chlorotic and nonchlorotic leaf tissue).

One of their questions was whether the intensity of bacterial infection was driven by insect herbivory. They collected data on the amount of bacteria on a set of leaves and related it to measures of herbivore damage and a few other potentially important covariates based on their deep knowledge of this system. In this exercise, we'll bring together a bunch of things we've just learned to do in R – from data import to analysis – to answer this question. Find and import the data file **leaf.master.199.xlsx** and run a regression to test whether **Total Bacteria (tot.cfu in the dataset)** is driven by the following:

- Leaf miner (*S. nigrita*) damage (called **prop.mined**)
- Beetle (*Phaedon* spp.) damage (called **prop.beetle**)
- Chlorosis (= yellowing of the leaf, called **prop.yellow**)
- Leaf position on the plant (called **leaf.num**)

Throughout, you'll have to troubleshoot as these data are meant to mimic the type of data that you'll encounter. Specifically, you should think about:

- Is the working directory set correctly?
- Did all the data import appropriately?
- Are the variables of the appropriate class?
- Do your data meet the assumptions of the statistical test you want to use (hint: try a $\log_{10}$ transform if not)?

You should report:

1. a histogram of the response variable
2. a quantile plot of the data used in the analysis (e.g. transformed if needed)
3. the statistics (F and p-values) for each independent variable

Bibliography

I am indebted to the authors of several sources used to compile this tutorial. Each provides excellent further reading on topics covered here as well as more advanced topics we were unable to cover here:

- Benjamin Blonder's 'An Introduction to R for Ecologists' – http://www.benjaminblonder.org/rworkshop/ (http://www.benjaminblonder.org/rworkshop/)
- Josh Grinath and Zak Gezon's R tutorials run at the Rocky Mountain Biological Laboratory in 2012 & 2013
- QuickR – http://www.statmethods.net (http://www.statmethods.net)
- The R Book – http://www.wiley.com/WileyCDA/WileyTitle/productCd-0470973927.html (http://www.wiley.com/WileyCDA/WileyTitle/productCd-0470973927.html)
- An Introduction to R – http://cran.r-project.org/doc/manuals/R-intro.pdf (http://cran.r-project.org/doc/manuals/R-intro.pdf)
- R Language Definition – http://cran.r-project.org/doc/manuals/r-release/R-lang.html (http://cran.r-project.org/doc/manuals/r-release/R-lang.html)

The datasets used in this tutorial came from the following:

- Kartzinel, T. R., J. R. Goheen, G. K. Charles, E. DeFranco, J. E. Maclean, T. O. Otieno, T. M. Palmer, and R. M. Pringle. 2014. Plant and small-mammal responses to large-herbivore exclusion in an African savanna: five years of the UHURU experiment. Ecology 95:787. http://dx.doi.org/10.1890/13-1023.1 (http://dx.doi.org/10.1890/13-1023.1) [photo from: http://thepalmerlab.com/ (http://thepalmerlab.com/)]
- Humphrey, P. T., T. T. Nguyen, M. M. Villalobos, and N. K. Whiteman. 2014. Diversity and abundance of phyllosphere bacteria are linked to insect herbivory. Molecular Ecology, 23: 1497–1515. doi: 10.1111/mec.12657

Appendix A – Useful packages

Below is a list of packages I count among my personal favorites.

Package name	Brief description
ape	Phylogenetic tools and analyses
car	Helper functions for regression and ANOVA
effects	Calculation of model effects
ggplot2	Publication quality and highly customizable graphics
lme4	Analysis of mixed effect models
nlme	Analysis of mixed effect models
popbio	Analysis of matrix population models
picante	Phylogenetic tools and analyses
reshape2	Data reorganization
vegan	Ordination techniques for community analysis
xlsx	Data import/export between R and Microsoft Excel

Appendix B – Exercise answers

Here are the answers for the excercise on the bacteria and herbivores of *Cardamine cordifolia*. It takes the form of commented code.

```
# first begin by pointing your working directory to where you've stored the data
# this is easiest to do in RStudio using the Files tab in the lower right corner
# and clicking 'More' > 'Set As Working Directory'
# (note: your file path may be different, and the syntax differs between PC and Mac)
setwd("~/Documents/Teaching/R_Tutorial_RMBL_2014")

# now read in the data file from Excel using the package xlsx, then subset to the columns we're using
library(xlsx)
cardamine <- read.xlsx("leaf.master.199.xlsx", sheetName = "community_data")
cardamine.sub <- subset(cardamine, select = c("tot.cfu", "prop.mined", "prop.beetle", "prop.yellow", "leaf.num"))

# check that everything imported correctly
str(cardamine.sub)
```

```
## 'data.frame':    43 obs. of  5 variables:
##  $ tot.cfu    : num  9250 160000 26250 28000 42500 ...
##  $ prop.mined : num  0.105 0.141 0 0 0 ...
##  $ prop.beetle: num  0 0 0 0 0 0 0 0 0 0 ...
##  $ prop.yellow: num  0 0.139 0 0 0 ...
##  $ leaf.num   : Factor w/ 12 levels "1","10","11",..: 5 9 6 11 5 5 11 6 10 3 ...
```

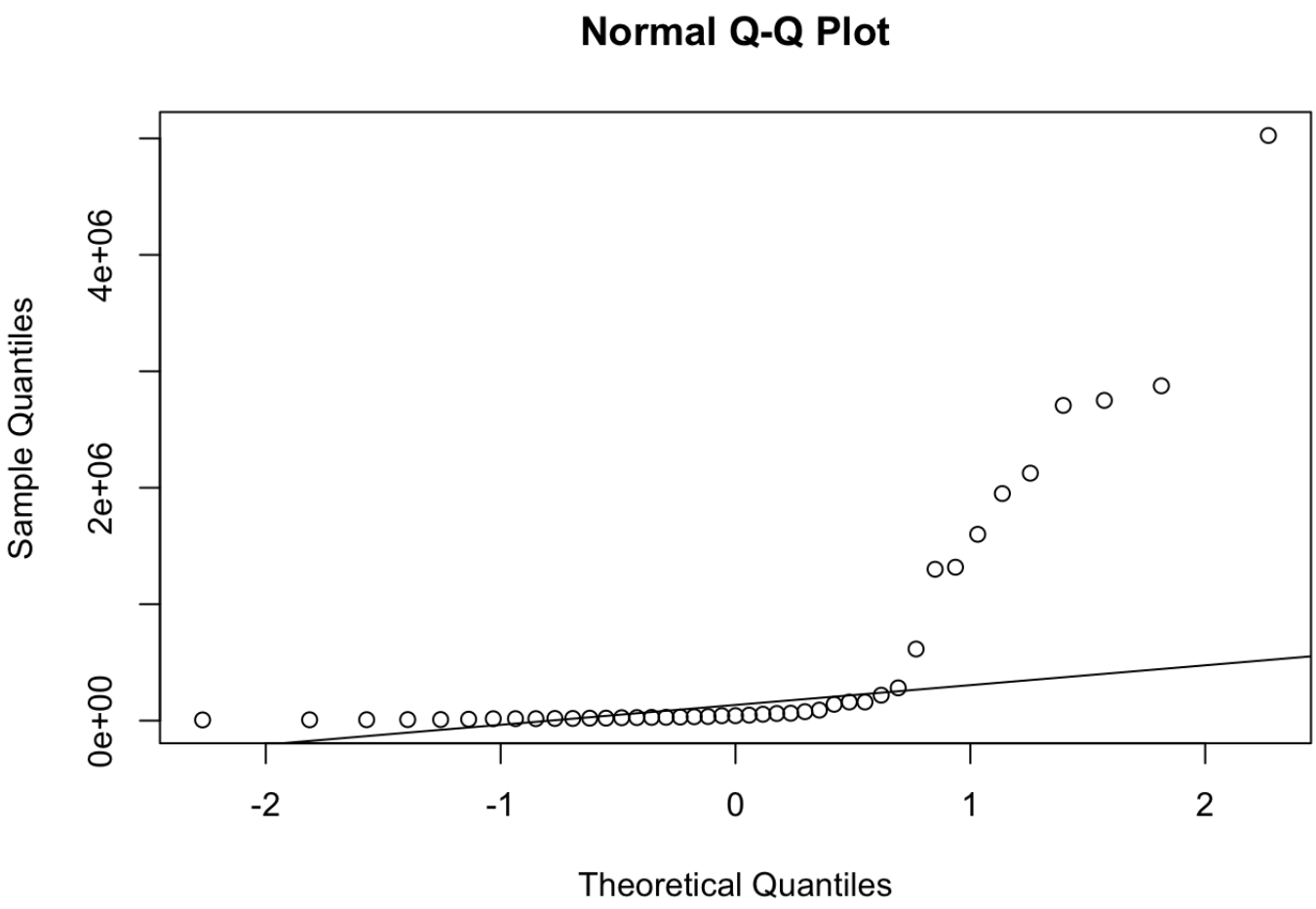
```
# leaf.num came in as a factor because some values were listed as NA
cardamine.sub$leaf.num
```

```
##  [1] 2  6  3  8  2  2  8  3  7  11 5  8  4  6  5  7  NA 3  5  2  4  5  7
## [24] 6  7  2  8  8  11 3  7  6  8  7  1  5  13 5  1  10 6  NA NA
## Levels: 1 10 11 13 2 3 4 5 6 7 8 NA
```

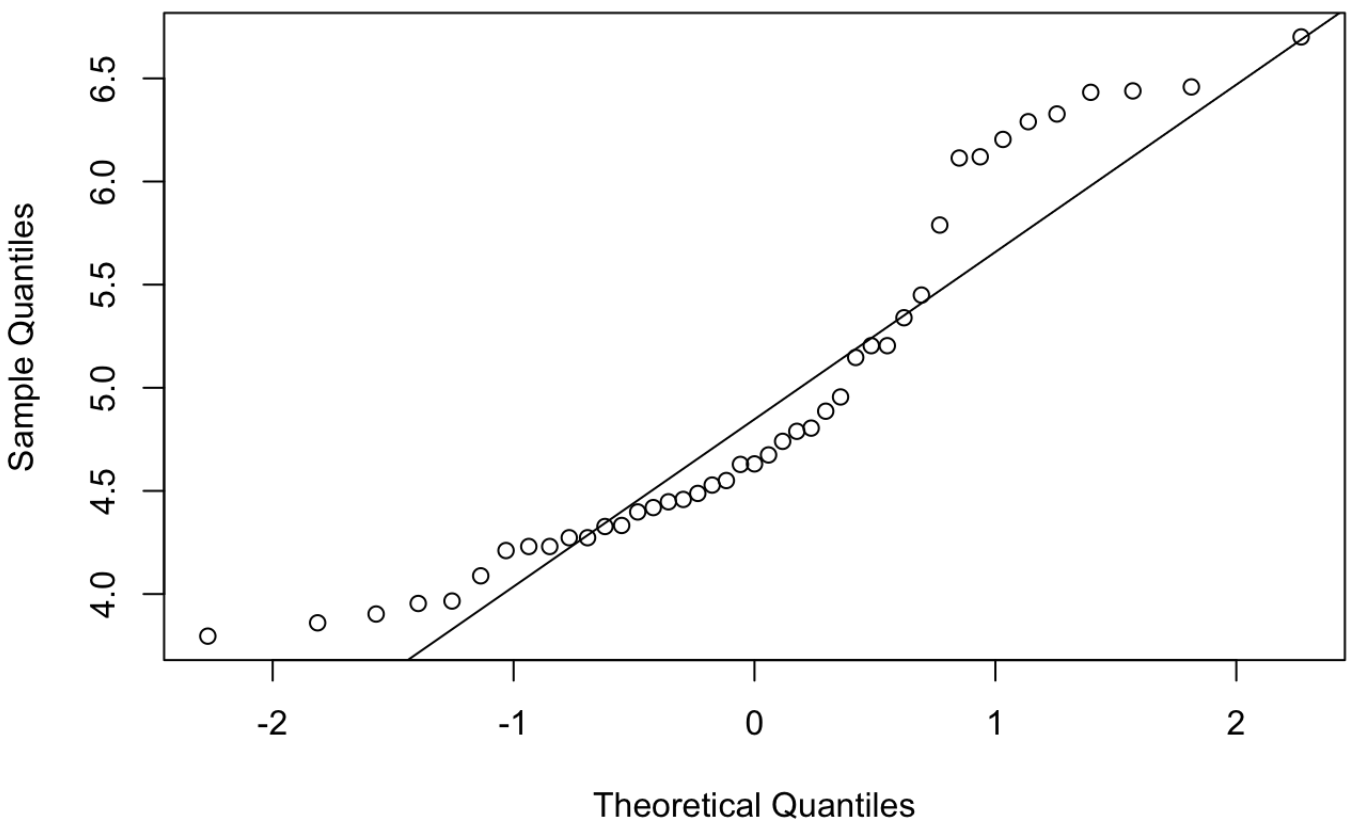
```
# it should be a number (i.e. leaf 2 is always between leaves 1 and 3)
# first set the text "NA" to the R default for missing values, NA
# then coerce the vector to class numeric
cardamine.sub$leaf.num[cardamine.sub$leaf.num == "NA"] <- NA
cardamine.sub$leaf.num <- as.numeric(cardamine.sub$leaf.num)

# one more check show's everything is now ready for data analysis
str(cardamine.sub)
```

```
## 'data.frame':    43 obs. of  5 variables:
##  $ tot.cfu    : num  9250 160000 26250 28000 42500 ...
##  $ prop.mined : num  0.105 0.141 0 0 0 ...
##  $ prop.beetle: num  0 0 0 0 0 0 0 0 0 0 ...
##  $ prop.yellow: num  0 0.139 0 0 0 ...
##  $ leaf.num   : num  5 9 6 11 5 5 11 6 10 3 ...
```

```
# check for normality using quantile plot
qqnorm(cardamine.sub$tot.cfu)
qqline(cardamine.sub$tot.cfu)
```




```
# a transformation is definitely needed, we'll use a log base 10
cardamine.sub$log.tot.cfu <- log10(cardamine.sub$tot.cfu)
qqnorm(cardamine.sub$log.tot.cfu)
qqline(cardamine.sub$log.tot.cfu)
```




```
# now build the model
bact.herb.mod <- lm(log.tot.cfu ~ prop.mined + prop.beetle + prop.yellow + leaf.num, data = c
ardamine.sub)

# calculate parameter estimates and p-values
summary(bact.herb.mod)
```

```
##
## Call:
## lm(formula = log.tot.cfu ~ prop.mined + prop.beetle + prop.yellow +
##     leaf.num, data = cardamine.sub)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -1.5211 -0.3805 -0.0723  0.3380  1.5287
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   4.3472     0.3111   13.97    7e-16 ***
## prop.mined    3.4289     1.2529    2.74  0.00968 **
## prop.beetle  15.9190    10.2241    1.56  0.12847
## prop.yellow   2.5766     0.6948    3.71  0.00072 ***
## leaf.num      0.0243     0.0367    0.66  0.51226
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.644 on 35 degrees of freedom
##   (3 observations deleted due to missingness)
## Multiple R-squared:  0.49,   Adjusted R-squared:  0.431
## F-statistic: 8.39 on 4 and 35 DF,  p-value: 7.39e-05
```