Introducing R

S. P. Blomberg and J. A. Wells
University of Queensland
School of Biological Sciences
s.blomberg1@uq.edu.au

April 10, 2014

1 Welcome

This document is designed to introduce you to R, a free software environment for statistical computing and graphics. It also introduces RStudio as an environment for running R and organising your data and analyses. No previous computer programming experience is required. However, a desire to practice and learn (often from mistakes!) is necessary. This document is designed to be practical, hence we will only introduce programming concepts when necessary. This document is not designed to teach all aspects of R, merely to give new users the basic skills they need to get R to work on their problem.

The philosophy behind R is quite different from most other statistics packages:

- R is free as in 'free beer'. It costs nothing to install or use R or any of the extensions that its active community of developers provides. R is available for Windows, Mac and Linux. R is published by a not-for-profit foundation, and will always remain free. If you work for a cash-strapped NGO on a conservation project in sub-Saharan Africa, you can still have access to world-class analysis methods by using R. And so can all your collaborators - you can share code, results and even R sessions.
- R is free as in 'free speech'. R is an open-source project, so the entire source code is freely available for modification or study. The R community believes that the software is a 'public good', and that statistical results should be independently verifiable - so you can look at the "source" code to see exactly how R works. For proprietary software packages, you have to rely on the good name and reputation of the corporation producing the software, because the source code is secret.
- R is modular. This means there are many free packages that extend the basic R environment to implement a huge range of analyses and graphics, and to interface with other software such as GIS. Currently, there are over 5000 packages on the Comprehensive R Archive Network, known as CRAN. For any analysis or graphic you would like to make, it's quite likely that a package exists that can help you.
- R has a vibrant community of users and developers. This means that there is considerable support and documentation you can access (including websites, guides and email lists where you can look for help and advice - see 'getting help', below), and that the base software and packages are being actively improved and extended (developers are all volunteers, but their number and levels of activity are phenomenal).
- R has a programming language, and it is relatively simple but powerful. When you use R, you can write the code for the analyses in a text file, so you have a permanent record of exactly how you did the analysis (This is called your "R script"). You can use this code for future analyses or modify it to new applications. You can also share your code with others, or you may wish to automate analyses, for example inside simulations, or to run similar analyses many, many times. R makes this very easy. If R is about to become your first computer programming language, you have made a good choice because it is easy to write short programs in R that get useful things done very quickly. As background: The R language is a dialect of the "S" language, which is used by the commercial statistical software package S-PLUS. R code is usually compatible with S-PLUS, though the two languages are gradually diverging.

So welcome to R. I hope you find it useful as a tool for your research. The learning curve is somewhat steep, but the view from the top is well worth it.

2 Setting up the R environment

2.1 Obtaining R

The easiest way to get R is to download it from the R web site. Go to the left sidebar, and select the link to CRAN. Click on it and choose a local mirror site (e.g. http://cran.ms.unimelb.edu.au). Make a 'bookmark' or 'favourite' in your web browser for the CRAN site, so you can visit it easily.

R is available for Linux, Windows, and MacOS X. Download and install the appropriate version for your system. [1]

The basic R interface, and other GUIs: The look and feel of R is almost the same across different operating systems. For Windows and MacOS X, there are some pull-down menus at the top (e.g. 'File, Edit, Help) - but the vast majority of what R can do, requires code to be entered in the command line. Some GUIs (Graphical User Interfaces) have been developed for easy 'point and click' use of R (where you select things from menus rather than entering code). Examples include RCommander and RKward). You can try out a GUI and it may help you get into using R, but generally the power of R comes from learning to write scripts and using the command line.

2.2 R Studio

R Studio is free software that makes it much easier to write your code, run R (using code, and a few extra menus), and organize your work (project workspaces, datasets, analyses and graphics).

First install R (as above), and then install R Studio from: http://rstudio.org/.

2.3 Starting R and R Studio

If you are using R by itself (without R Studio), simply click on the R icon on the desktop, or launch it from the Applications menu or Start menu). R should start with a short copyright message, and leave you at the command prompt, which looks like this: >.

If you are using R Studio, you don't need to start R first, you can just open R studio (click on the desktop icon, or launch from the Applications or Start menu). You will see 3 windows: The left window shows you the R console (commands and results will be displayed here). The top right window has tabs that show 1. your workspace (this is a 'virtual folder' for everything you use or create in your R session - i.e. any data files, results objects etc) and 2. the history (every command, in the original order, including all your mistakes and repeats). Finally, the bottom right window has tabs that show your files, plots (i.e. graphics), packages menu, and help display.

Click the small square box in the top right corner of the left('console') window. This will give you a 4th window.

This window is for writing your 'R script' (your code), which is really helpful as your permanent record of exactly the commands and annotations you want to keep from your session. You can add 'comments'to annotate your code - this can be anything you want to note to yourself to help explain what you did or what the code means - just start by typing the # (hash symbol), and R will treat the rest of that line as a comment not a command.

Start by typing a title for your script, as a comment, e.g. type:

```
> # R script during the Workshop on (today's date)
```

and save the file (e.g. with the name: 'RscriptforCourse.R'). This will be a simple text file (meaning you can read it in any text editor), and if you give it the extension ".R", then you can easily identify it as an R script.

[1] If you do a lot of scientific computing, you should really consider moving to a Unix-based system, such as Linux or MacOS X. Unix environments have a lot of free scientific tools available (in addition to R), and you can automate analyses very simply by using shell scripting (or Perl or Python etc.).

You can send your commands to R from this script window very easily, by highlighting (selecting) any text or lines, and clicking the 'Run' button at the top right. (This means you don't need to cut and paste anything into the R console window; you can simply send it directly.)

Besides R Studio, there are some other free programs you can use to write code and interact with R. R Studio is generally our favourite, but you might like to check out: Tinn-R, JGR, or NppToR.

3 Notation and key symbols

`>` is R's 'command prompt'. It means that R is ready to receive a command. Any commands you type (or send from R Studio) will appear at this prompt.

In this document, the `">"` symbol is shown at the start of each command, because that is what you will see on the R console, and it easily identifies R code. However, you DON'T need to type it (R will get confused if you do)

`+` is R's 'continuation prompt'. It means the current command is not complete, and you can enter more information to finish it. (for example if you press 'enter' to make a new line, or if you haven't closed a final bracket)

If you want to cancel a command, just press ESC then ENTER, and a new command prompt will appear.

`#` is R's 'comments' character. If you type this, the rest of the line will be treated as a comment, not a command (R will not compute anything from it)

`<-` is the assignment symbol. Anything on the right is assigned to whatever is on the left. For example, x <- 2+3 means "x is 2+3", or equivalently: "assign the value 2+3 to an object called x"

spaces - You can add spaces between words in your R code to make them more easily readable, e.g. writing 2 + 3 with spaces between the numbers and symbols. However, do not add spaces within a word, or before brackets, e.g. use help(), not help ().

Note that R is *case sensitive* - so, for example, 'myData' is completely different from 'mydata', due to the uppercase 'D'. Be careful of this when you type, and remember to check this if you are getting errors.

To repeat or quickly edit a command in the R console, you can scroll up and down through previous commands using the 'up' and 'down' arrows on your keyboard. This is handy for correcting small errors. When you get it right, remember to include the command in your Script so you will have a permanent record of what worked!

c() is a function you will see often. c(1,2,3) means *concatenate* a set of elements (e.g. the numbers 1,2 and 3) into a vector. You can see more examples in the section on Vectors.

4 Your Working Directory

The working directory is the place on your hard disk where R looks first (by default) when it looks for data files, and where R saves output files such as graphics etc. You can find out where your working directory is using:

```
> getwd()

[1] "/home/simonb/Desktop Stuff"
```

You can set the working directory to any other directory by giving a different path. The following is just an example - you will need to modify it to suit your computer and the folder you want to use.

```
> setwd("/new/working/directory")
```

You can also save your complete R session as a file in your Working Directory, so that you can return later on, and start again where you left off. (More on this later, in the section 'The Workspace').

It's a good idea to set up a new folder as your current working directory for each new project (so that each project will have its own separate working directory). To do this, create a new folder on your computer, and put any specific data files or scripts into it. Then when you are using R, use the 'setwd' function to tell R that this is the current working directory. This enables you to have all your scripts, data and saved R sessions (workspaces) in one place on your hard disk.

In R studio, another option is to set the working directory using the Menu. Go to the 'Session' menu, and choose 'set working directory', then 'Choose directory'. This allows you to navigate to your folder, and choose it. R will print the address on the console.

Make sure you save your setwd command into your Script, for future reference. It's usually a good idea to have this as the first command in your script, as an essential part of setting up your session with R.

5 Getting help

As a new user, the help facilities and search functions will be important tools to help you learn the language, and draw on the huge amount of experience and advice that's out there. (Even R experts need to get help, for example when using a new package). There are several sources of help in R:

- `> help.start()`

  A web browser should open up with access to the help pages in html.

- If you know the name of the package or the specific function you wish to use, then help() or ? can be used as in:

  ```
  > help(glm)
  > ?glm
  ```

  The two commands above are equivalent (help and ?). You can also view any examples that have been provided for a function by typing:

  ```
  > example(glm)
  ```

- If you don't know the precise name of the function, you can do a keyword search of the currently installed packages:

  ```
  > help.search("glm")
  > ??("glm")
  ```

  again, the two commands are equivalent, as ?? is short-hand for help.search

- You can search the R web site and mailing lists from the command line within R:

  ```
  > RSiteSearch("glm")
  ```

- The RSeek search engine is excellent.

- Many free help documents, tutorials, etc. are available online on CRAN. Go to the "Documentation" menu in the sidebar on the left, and look in the "Contributed" section.

- Many books have been published on the R system. A list can be found on the R web site.

- Graphics: Inspiration (and code) for R graphics can be found in the R Graph Gallery.

- Task Views: CRAN has a section called "Task Views". Each 'Task View' relates to a research area or method (e.g. 'Spatial', 'Genetics', 'Graphics') and gives a brief overview of the packages that may be useful for your analysis.

- WIKI for R - basically a portal to find and share information: R-wiki

- Help lists: The largest email list is r-help. However, you may get a really negative response if you haven't been very clear or have asked something you could have found elsewhere. Generally, it is best to 1. search for existing answers to a similar question (on email list archives, or using RSiteSearch or Rseek, above), 2. ask someone you know locally who knows R, and if you can't find what you need, then 3. post a new question to a help list, being as clear as possible about your problem, what you want to do, and any code you have tried to run.

6 Getting started

We will begin to build our R skills by exploring the language, with the aim of developing a vocabulary that will allow us to do some basic data manipulation, analysis and graphics. We will start with simple calculations and objects, and entering data.

Start by using R as a simple calculator

```
> 2*14
```

The result will display in the console. The [1] in front of the result identifies this as 'the first element' (this isn't very informative when there is only one, but is helpful to indicate the position in longer sequences of values.)

Next, create an object called x, and give it a value

```
> x <- 28
```

This simple command involves several concepts you will use all the time in R.

x is an object, and its value is 28. The assignment symbol "<-" is used to assign a 'value' to an 'object'.

In R, everything is an object. You will create objects to hold your data (as in this example), to hold analyses, and to draw graphs. Hence, R is described as an *object oriented* language. You can use R without thinking deeply about objects most of the time, but it helps to be aware of the "object <- value" idea, and the main types of objects (which we'll get to later on).

Next, use the object x to calculate a value for y

```
> y <- 4*x^2
```

You can see the value of any object by typing its name, and it will display on the console.

7 Data Entry: Getting your data into R

R can read data in a variety of different formats. The most common data format is a simple text file (e.g. tab-delimited, or comma-delimited). Other options include all the main commercial statistics package formats (e.g. SAS, SPSS, Stata, Systat). R can also read data directly from Excel (though this is often dangerous, due to Excel's hidden formatting and other quirks. It's much safer to save your Excel file as a text file, before you ask R to read it).

The read.table function is the most important command for reading data into R. It is typically used as:

```
> dat <- read.table("mydata.txt", header=TRUE, sep="\t")
```

Don't type this command just yet - we will do an activity to create the data first:

7.1 Activity 1

Create a new folder on your hard disk, called "Rworkshop" or any other name you choose. (No spaces in the name)

In R, type > getwd(). This shows you the current working directory, where R will look whenever you tell it to import or save files.

Set the working directory to the new folder you created, using:

```
> setwd("/mypath/Rworkshop") # change the first part `mypath' to your folder path
```

Next, create a simple dataset in Excel or a text editor. Give your dataset 4 columns, with 1 row for headings, and 8 rows of data. Give the columns names that are relevant to your work (e.g. Species, BodyMass, WingLength, BeakLength). In the first column, below the heading, type the numbers 1 to 8. Fill the other columns with random numbers. (To do this in Excel: type =rand() into the formula bar, then copy the formula to all cells. Then select all, copy, and 'paste special - values'. Pasting just the values means that Excel won't generate new random numbers over and over.

Save your file as a tab-delimited text file, in the folder you just created. (In Excel, use 'Save As', choose tab-delimited text, and click yes or continue to save as this file type)

Next, import your data to a new object called 'dat', using the command

```
> dat <- read.table("mydata.txt", header=TRUE, sep="\t")
```

This command reads a text file called "mydata.txt" and assigns it to the object called "dat" in your R workspace. See below to understand the parts of this command. If it doesn't work, try reading the following section on common errors.

Look at your data: When you have imported it, you can look at your data just by typing its name, or you can use summary(dat), and plot(dat). In R Studio, you can also look at your data by clicking on its name in the Workspace tab. Reflect on the way R imports data and relate it to your own, real datasets.

7.2 What does this command mean? Introducing functions, assignments, and arguments

```
> dat <- read.table("mydata.txt", header=TRUE, sep="\t")
```

As stated above, this command reads a text file called "mydata.txt" and assigns it to the object called "dat" in your R workspace. This introduces several concepts that you will use all the time:

'read.table' is a *function*, i.e. it acts on an object (or more than one) to produce something. The command above is a 'call' to the function. The result is called the 'value' of the function (Language note: this 'value' may contain many numbers or other elements, so it's not necessarily a single numeric value).

Assignments. As mentioned above, the assignment symbol `"<-"` is used to assign a 'value' to an object. In our example, the value (i.e. result) from the 'read.table' function is assigned to the object "dat". (If an object called dat existed before, it will be replaced. Otherwise, a new object called dat will be created.) The more familiar 'equals' sign `"="` is also used as an assignment symbol, but only inside function calls, as we did twice in the example above. (There is one other assignment symbol, `"<<-"`, which is not recommended if you are new to R. It's like a 'super assignment' that makes assignments in the workspace, rather than inside the body of a function. `"<<-"` should only be used by advanced users. *Functions* can be given *arguments.* Arguments are the 'pieces of information' that tell the function exactly what to do or what to act on. Functions usually have some 'essential' arguments (e.g. you often need to tell it which data to use), and some 'optional' arguments that specify particular options or settings (and will have default values that are used if you don't provide others).

In our example above, we gave values for three arguments.

The first argument is "mydata.txt". This tells R what file to look for, and where. This should be the name of a text file in your working directory. If a file is not in your working directory, you can still import it by providing the full path name, e.g. "C://myfolder/mydata.txt". Or, if you don't know the exact file path, you can use the file.choose() function, which creates a pop-up dialogue box that you can use to navigate your files to find the file you want. The file.choose() function returns the full path name as a value. You would use it to replace the name of your file, as in the following:

```
> dat <- read.table(file.choose(), header=TRUE, sep="\t")
```

The second and third arguments are 'named' arguments, so they use the `"="` symbol to assign their values.

"header" is the second argument, and is given the value TRUE. Here, we are telling the read.table function that "mydata.txt" has column headings that contain the names for your variables.

"sep" is the third argument. It tells read.table that the columns of data are separated by tabs. The "$\backslash t$" symbol is the symbol for the tab character. (If your data had a different separator, such as a comma, then you would change this symbol.)

So the above statement tells read.table where to look for the data, and the format of your text file. If any of the arguments are incorrect or missing, you might generate an error. Understanding function arguments is critical for getting R to work for you.

7.3 Common errors with read.table

1. R can't find the file. 1. Check that you have spelled the name correctly, including matching the upper or lower-case letters. 2. Check that you have included the file extension .txt. 3. Check that R is looking in the right folder, e.g. use getwd() and setwd() to change your working directory, or provide the full path name to the file.

2. Forgetting to set header=TRUE. Usually your data set will have column headings. If you don't tell this to R, this will cause your column headings to be interpreted as data values, which confuses everything.

3. Wrong separator. R can recognise several different column separators, but you need to tell it the right one. We specified tab characters above. R's default is "whitespace", which can be spaces or tabs. Another common option is commas, as found in .csv files.

4. Column headings that cause confusion. If you have spaces between words in your column headings, R will usually interpret each word as a variable, and you will get an error message. It is good practice to have one-word variable names. Use "." or capital letters to create multi-word variable names, and avoid spaces. For example: BodyLength, or body.length

5. Missing values. Any blanks, with no symbol replacing the missing value, will cause an error because the number of values will not equal the number of columns for any rows that have missing data. R uses "NA" as the missing value symbol. In your data file, you can easily use excel or a text editor to replace blank cells with the letters NA. Also be careful to never use 'zero' when you mean 'missing' NA. Zero is a known, definite value, whereas NA means the value is unknown, so it can be dangerous to confuse the two.

7.4 Friends of read.table

You may wish to explore alternatives to read.table, such as read.csv and read.delim. Try looking at ?read.table for further help.

8 Common Objects

Next we will learn about the most common types of objects in R: vectors, matrices, lists, dataframes, factors, and functions.

8.1 Vectors

Vectors are sequences of elements of the same type (i.e. real numbers, integers, true-false, or character strings). The most commonly encountered is a vector of numbers. Try the following code:

```
> vec <- c(1, 1.4, 2, 3.2, 5.6, 8, 14, 22.2) #example: exact numbers not important
> print(vec)

[1]  1.0  1.4  2.0  3.2  5.6  8.0 14.0 22.2
```

This uses the c() function to *concatenate* the individual elements into a vector, and assigns the result to the object "vec". This is a new object and we chose this particular name (vec) just to remind ourselves it's a vector. We could have called it almost anything. This object is a vector of length 8. (Scalar numbers (i.e. single numbers such as 2 or 200) are simply vectors of length one). Note that when R prints the vector on the console, it places a "[1]" in front of the vector. This indicates the first element of the vector. This gives us an 'index' for the position of each element in the vector, and is useful when printing very long vectors(as the index will tell you the position at the start of each line on your console). Try:

```
> print(rnorm(200))
```

Examples of simple vectors:

```
> c(1,2,3,4,5) # concatenate numbers to make a vector
```

```
[1] 1 2 3 4 5

> 1:5    # 1:5 is shorthand for generating the sequence of integers from 1 to 5

[1] 1 2 3 4 5

> -(1:5) # -1 times each element

[1] -1 -2 -3 -4 -5

> -1:5   # from -1 to 5

[1] -1  0  1  2  3  4  5
```

Indexing enables us to select specific elements from within an object. For example, we can use the square-bracket notation, "[", to index the elements of a vector. To access the 3rd element of vec, we could type:

```
> vec[3]

[1] 2
```

To access the 6th to 8th elements we could use:

```
> vec[6:8]

[1]  8.0 14.0 22.2
```

Remembering that 6:8 is short-hand for a vector of the integers from 6 to 8. Here, we are using the vector 6:8 as an 'index' to access the 6th to 8th values in another vector. Alternatively, we could access:

```
> vec[-(1:5)]

[1]  8.0 14.0 22.2
```

The minus sign "-" means that we did *not* want to include the values of vec[1] to vec[5]. (In words, you could say 'give all values from vec, except the values indicated by the minus sign)')

Indexing can be used to access the values, as above, or to replace them with new values, for example:

```
> vec[1]<-0
> vec[1:2]<-c(0,1)
```

You can also make vectors that contain characters, or character strings, rather than numbers, for example:

```
> vec2 <- c("x", "y", "z")  # quotation marks for character elements
> vec3 <- c("Monday", "Tuesday", "Wednesday")
```

Try out your own code to generate a few more vectors, to ensure you understand how to use the square-bracket notation "[]" to select elements of a vector, and how the colon, ":" can be used as a shorthand for generating vectors of integers.

8.2 Matrices

A matrix is simply a dataset with rows and columns (i.e. there are 2 dimensions), in which all the data values are of the same type (number, character or logical). (For datasets with columns of different types, see 'Data frames', below.) Matrices are one of the most useful mathematical objects for statistics. To create a matrix, do:

```
> mat <- matrix(vec, byrow=TRUE, nrow=2)
> print(mat)

     [,1] [,2] [,3] [,4]
[1,]  0.0    1    2  3.2
[2,]  5.6    8   14 22.2
```

We have used the matrix() function, with three arguments. The first (unnamed) is a vector of numbers that R will use to fill up the matrix. Here, we have used the vector we created earlier, called 'vec', which has 8 elements. The second argument, byrow=TRUE tells R to fill the matrix by row order (i.e. fill the first row, then fill the second row...), and finally, nrow=2 tells R that we want the matrix to have 2 rows. The matrix is then assigned to the object, "mat". Notice that when R prints the matrix, we have a series of square-bracket boxes corresponding to the positions of the rows and columns of the matrix.

Matrix dimensions tell us how many rows and columns we have. We made 'mat' from an 8-element vector, so when we specified 2 rows, we obtain a matrix of 2 rows x 4 columns. Check this using the 'dimensions' function:

```
> dim(mat)

[1] 2 4
```

Alternatively, we could have made a matrix of 4 rows x 2 columns, from the same data. What command would we use to do this? (Try modifying the command we used to create mat, and use it create a new matrix called mat2)

Indexing matrices. Just as with vectors, we can access matrix elements using the "[" notation, but this time in two dimensions. We refer to the elements using [row,column] notation. So the element in row 2, column 1 is:

```
> mat[2,1]

[1] 5.6
```

You can also look at many more examples of using square brackets to index your data by typing ?"[" (the quotation marks are necessary here).

Until now, we have been dealing with numeric vectors and matrices. However, we can have matrices and vectors of character variables too:

```
> letters[1:8]

[1] "a" "b" "c" "d" "e" "f" "g" "h"

> mat3 <- matrix(letters[1:16], nrow=4)
> mat3

     [,1] [,2] [,3] [,4]
[1,] "a"  "e"  "i"  "m"
[2,] "b"  "f"  "j"  "n"
[3,] "c"  "g"  "k"  "o"
[4,] "d"  "h"  "l"  "p"
```

Importantly, *the elements of vectors and matrices must be of the same type.* We cannot have a matrix of mixed character and numeric elements, for example. (Try making a vector that includes numbers and characters, for example, and it will convert all the numbers to characters so they can be treated as the same type.)

8.3 Lists

Lists can contain objects of any type. We create a list here:

```
> my.list <- list(vec, vec2, mat)
> print(my.list)

[[1]]
[1]  0.0  1.0  2.0  3.2  5.6  8.0 14.0 22.2

[[2]]
[1] "x" "y" "z"

[[3]]
     [,1] [,2] [,3] [,4]
[1,]  0.0    1    2  3.2
[2,]  5.6    8   14 22.2
```

Notice several things about my.list:

- The list contains objects of different types: a numeric vector, a character vector, and a numeric matrix.

- The elements of the list have different lengths or dimensions.
- The "[[" notation is used to access elements of a list:

  ```
  > my.list[[2]]

  [1] "x" "y" "z"
  ```

  The above code extracts the second element of the list my.list, which corresponds to a vector of three letters.

Lists are a useful data structure for grouping disparate objects together.

8.4 Data Frames

A data frame is a very common object in R, and is probably very familiar to you as the format for datasets you've used in your own research. A data frame has rows and columns (like a matrix), but the columns can be of different types. Usually, your rows will correspond to individual observations, and the columns represent the different variables you have measured.(Formally, a data frame is a 'list' of vectors. These vectors form the columns of your data frame, and all have the same length, so that the columns all have an equal number of rows).

Creating a data frame - You created a data frame in the Activity earlier, when you used read.table(). There are also other ways to create data frames. Here we construct a data frame with 3 columns and 20 rows, where the columns are the vectors x, y, and z, and the data values are random Normal variates (generated using the function called rnorm).

```
> dat <- data.frame(x=rnorm(20), y=rnorm(20), z=rnorm(20))
> head(dat)

           x          y          z
1  0.8110090  0.1943589  0.9410126
2 -1.1165617  1.4203680 -0.3568443
3 -0.6372097  0.4358469  0.3676833
4  0.9377177  0.3756026  0.8919044
5 -0.2597355  0.4481600  0.2706534
6  0.2244069 -1.5532755  0.4300426
```

The head() function enables us to look at the first 6 rows of the data frame. This is useful when working with large data frames (so you don't have to print the whole thing). Alternatively, you could look at the first 10 rows, using head(dat, n=10), or the last 10, using tail(dat, n=10)

Indexing data frames. Because data frames are lists, we can use the "[[" notation to extract columns. And because the columns of a data frame are vectors, you can use the "[" notation on them, too:

```
> dat[[2]]

 [1]  0.1943589  1.4203680  0.4358469  0.3756026  0.4481600 -1.5532755
 [7]  0.3435042 -1.7980250  1.1517761 -1.4174554  0.8180157 -0.5845328
[13]  0.8490907 -1.3760359 -0.6967972  1.9681305 -1.1230396  0.1852976
[19] -0.7976197  1.2322044

> dat[[3]][1:5]

[1]  0.9410126 -0.3568443  0.3676833  0.8919044  0.2706534
```

We can also refer to columns by their heading names, rather than the column number. We can do this with the "$" notation:

```
> head(dat$z)
> head(dat$x)
```

We can use these symbols to access the data in several different ways. For example, here are 4 different ways of selecting rows 2:4 from the first column, x:

```
> dat[2:4 , 1]
> dat[[1]][2:4]
> dat[["x"]][2:4]
> dat$x[2:4]
```

We can also select just those elements that meet some requirement, for example selecting values from column y, based on values in column x:

```
> dat[["y"]][dat$x>0.2]
```

Or we could use the subset() function to make a new dataset that contains only records that meet some requirement(s):

```
> dat2 <- subset(dat, x>0.2)
> # gives a new data frame made from all rows that values >0.2 in column x
> dat2
> dat3 <-subset(dat, x>0.2 & y>0.1) # similarly, with two criteria
> dat3
```

8.5 Factors

Factors are a type of vector that is used to represent categorical variables (where you have a few possible categories, and they may be repeated many times each - for example representing treatment levels, or gender, or species). Factors are especially useful when fitting ANOVA-type models. To make a 'factor', first we create a character vector representing 3 treatment levels, with 10 observations each. We then convert it to a factor.

```
> vec1 <- rep(c("a", "b", "c"), each=10)  # rep function `replicates'
> vec.fac <- factor(vec1) # convert into a factor
> summary(vec.fac)

 a  b  c
10 10 10

> levels(vec.fac)

[1] "a" "b" "c"
```

We used the rep() function ("repeat") to create our vector with 3 levels, 10 observations each. We then used factor() to convert our character vector to a factor. Using summary(), we see that there are indeed 3 levels, and each level has 10 observations. Using levels(), we see just the information on the names and order of the levels. Here, "a" is the first level. If you run an ANOVA, the first level will be treated as the "baseline" category for linear contrasts. Alternatively, you could choose another category for the baseline (perhaps "c" for "control"?), by using the relevel() function:

```
> vec.fac <- relevel(vec.fac, "c")
> levels(vec.fac)

[1] "c" "a" "b"
```

Now "c" is the baseline level, as indicated by its position in the output of the levels() function.

Let's put this new knowledge to work in a simple analysis: a one-way ANOVA with 3 treatments. For the analysis, we can simulate some data (use your own data if you have it!)

First make a dataframe with the columns 'treat' and 'resp':

```
> dat <- data.frame(treat=rep(letters[1:3], each=10), resp=rnorm(30))
> dat$resp <- dat$resp + c(rep(7,10),rep(5,10),rep(0,10))
> head(dat)

  treat     resp
1     a 6.230449
2     a 5.942930
3     a 4.977390
4     a 7.130676
5     a 5.976213
6     a 4.781768
```

The first line creates the data frame with random normal values for resp, between 0 and 1. The second line adds 7, 5 or 0 to these random values. This gives a moderately realistic dataset, where the levels have different means, and some random variation around them. You can visualise this by typing plot(dat)

Now we will fit a traditional ANOVA, using the aov() function:

```
> fit <- aov(resp~treat, data=dat)

> summary(fit)

            Df Sum Sq Mean Sq F value   Pr(>F)
treat        2 198.58   99.29   88.36 1.42e-12
Residuals   27  30.34    1.12
```

Models and Formulas. When fitting the anova model with aov(), we have used a *formula* to represent the model. We know it is a formula because it has a ~ (tilde) in it. The response variable is on the left, and the explanatory variable is on the right of the tilde. Formulae are used to describe statistical models for many statistical functions in R, such as anova, linear models, glms and many others. We can read the formula above as "resp is modelled by treat".

The summary() function gives us the ANOVA table. This function works differently on different objects). Often, a particular type of object will have particular summary(), print() and plot() functions associated with it (different *methods* in computer science terms). As the user, you just have to remember one command, and R will run the function using the method that is appropriate to the object.

8.6 Functions

In R, functions are objects, so you can create them and manipulate them, like any other object. A large part of the power of R comes from the ability to easily create your own functions. To create a function, we use the "function" keyword in the assignment. The following function is called my.func, and has three arguments. The value of the function is called result, which is given by the expression $x^a + b$:

```
> my.func <- function (x, a, b) {
 my.result <- x^a+b
 my.result            # last line tells R what to return
 }
```

When you type this: press ENTER at the end of each line (so R knows they are separate steps), and remember that R places the + symbol as a 'continuation prompt', for you to add more commands. (It doesn't mean 'plus' as in addition.) Also note that you need to write 'my.result' on the last line, so that R knows what you want the function value to be. If you leave this out, you will get no result.

```
> my.func(4,3,2)  # runs the function

[1] 66

> # here, the arguments are matched by position 1,2,3.
> # my.func(x=4, a=3, b=2) would be equivalent, matching by name.
```

Functions can also contain other functions, for example the mean function:

```
> my.func2 <- function (x, a, b) {
 result <- mean(x)^a+b
 result
 }
> dat1 <- my.func2(x=vec,3,2)
> # runs the function, and saves the result to a new object called dat1
> dat1

[1] 345
```

8.7 Summary of common objects

The 6 most frequently used types of *data* objects in R are vectors, matrices, lists, data frames, and factors.

- Vector - a vector represents a set of elements of the same type (numbers, character, logical).
- Matrix - a matrix is a set of elements of a single mode, with rows and columns (i.e. 2 dimensions). Note that a matrix is a 2-dimensional example of an n-dimensional 'array' object.
- List - a list is a generalisation of a vector and represents a collection of any data objects. For example, a list could contain a vector, a matrix, a dataframe, and another list.
- Dataframe - a data frame is similar to a matrix object in having row and column dimensions, but the columns can hold different types of data (e.g. a character column, then numeric columns).
- Factor - a factor is a specific type of vector that stores categorical data.

Finally, we remember that *functions* are also objects. Functions act on other objects to give a result.

9 The Workspace

R has a "workspace", in which all objects are stored - this is like a 'virtual folder' that R uses to contain all the objects created or imported in your session. R gives you easy access to the workspace, using the ls() and rm() functions.

9.1 What's in the workspace?

The most commonly used function for examining the workspace is ls(). The ls() function *lists* the objects in the workspace:

```
> ls()  # note this starts with a lower-case L, not a number 1

 [1] "dat"      "dat1"     "fit"      "mat"      "mat3"     "my.func"
 [7] "my.func2" "my.list"  "vec"      "vec1"     "vec2"     "vec3"
[13] "vec.fac"
```

You can see that all the objects we have created so far (data objects, results objects, functions) are in the workspace. (Yours may look slightly different from that above, if you named things differently etc).

We can also use ls() to examine a specific subset of objects:

```
> ls(pat="vec")

[1] "vec"     "vec1"    "vec2"    "vec3"    "vec.fac"
```

The use of the "pat" (for "pattern") argument is useful when there are a large number of objects in the workspace, and you just want to select the ones that share part of their name.

R keeps all the objects in the workspace, which is in the RAM memory of the computer. R can use as much memory as present in your computer, so the size of R problems is largely limited by the amount of RAM. The current workspace is *not* stored on disk (unlike S-PLUS). This gives R a speed advantage. But it also means that if your computer crashes, you will usually lose the contents of your workspace. *So save often!*

R can save the workspace to disk, using save.image():

```
> save.image(file="sessionName.RData")
```

This enables you to begin another session with all the objects you have created during this one. You can even send the .Rdata file to someone else, or open it on a different operating system.

When you quit R, it automatically asks whether you want to save your workspace. Remember that this will be saved (as with any other files) to your current working directory. If you want to save to a new directory, create the folder on your computer, then set the working directory using setwd(), then save the workspace.

So, we have a "Workspace" in the computer's active memory, which is a virtual storage for objects for the current session, and we have a "Working Directory", which is the physical location of files etc. on your computer's Hard drive.

9.2 Tidying up the workspace

The rm() function is used to (*remove*) unwanted objects from the workspace. This deletes them from the memory. rm() takes the names of objects as arguments:

```
> foo <- 1
> bar <- 3.14159
> baz <- 2.71828
> ls()

 [1] "bar"      "baz"      "dat"      "dat1"     "fit"      "foo"
 [7] "mat"      "mat3"     "my.func"  "my.func2" "my.list"  "vec"
[13] "vec1"     "vec2"     "vec3"     "vec.fac"

> rm(bar, baz)
> ls()

 [1] "dat"      "dat1"     "fit"      "foo"      "mat"      "mat3"
 [7] "my.func"  "my.func2" "my.list"  "vec"      "vec1"     "vec2"
[13] "vec3"     "vec.fac"
```

To clear the entire workspace, use the "list" argument to rm():

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

character(0)
```

Be careful however, because this command is impossible to undo, and you will not get any warning from R.

10 Packages

R is modular, and many packages are available on CRAN. Each package contains many functions that enable you to perform particular analyses or actions - for example, 'scatterplot3d' is a package of functions for drawing 3D graphs. R comes with several powerful packages built in. To find others, first look at the 'Task views' on CRAN for an overview of packages that relate to your field of research.

To install a new package, do:

```
> install.packages("scatterplot3d")
```

When we want to use a package, we need to tell R to load the package's library of functions. For example, to load the MASS package, which is built-in so you don't need to install it first:

```
> library(MASS)
```

Then we can use all the functions and data sets in the MASS package:

```
> help(package=MASS)
> ?glmmPQL
```

11 Manipulating Data

R offers you many different tools for manipulating data. One of the simplest and most powerful is the "[" notation. We will create a data frame of random numbers and perform some simple manipulations:

```
> dat <- data.frame(x1=rnorm(25), x2=rnorm(25), cat=sample(c("a", "b"), 25, replace=TRUE))
> head(dat)
```

```
           x1         x2 cat
1 -0.12059626 -1.1568619   a
2  1.30559363 -2.9279947   b
3  0.89848137  1.5784460   a
4 -0.02901424  0.3402646   a
5  1.82507479  0.6761959   b
6  1.07343807  1.3231281   b
```

We have used the data.frame() function to create our data frame, and within that function, we have created the columns named, "x1", "x2", and "cat", using the rnorm() and sample() functions. The sample() function performs random sampling. We have told it to sample from a and b, 25 times, with replacement (so sampling an 'a' doesn't change the probability of sampling it again).

We can find out more about the data using the summary() function:

```
> summary(dat)
```

```
       x1                  x2            cat
 Min.   :-1.1550   Min.   :-2.927995   a:17
 1st Qu.:-0.4717   1st Qu.:-1.156862   b: 8
 Median : 0.3249   Median :-0.006727
 Mean   : 0.1576   Mean   :-0.129487
 3rd Qu.: 0.6411   3rd Qu.: 0.845754
 Max.   : 1.8251   Max.   : 2.027037
```

```
> dim(dat) # dimensions
```

```
[1] 25  3
```

We see that for the continuous variables, we get a 6-number summary of their distributions. For our categorical variable, we simply get the number of cases in each category. The dim() function tells us the dimensions of the data frame (numbers of rows and columns). Now let's sort the data to get the "a"s and "b"s together:

```
> dat2 <- dat[order(dat$cat),]
> head(dat2)
```

```
           x1         x2 cat
1 -0.12059626 -1.1568619   a
3  0.89848137  1.5784460   a
4 -0.02901424  0.3402646   a
7  0.43345738  0.6099755   a
8 -0.30285783  0.2025611   a
9 -0.99310898 -1.5098320   a
```

The order() function orders the data according to its argument(s). Notice that we have used the "[" notation. Even though dat is a data frame, it can also be used like a matrix, with respect to the "[" notation. Also, we have created a new data frame called "dat2" that is a sorted copy of "dat". So now we have two data frame objects containing the same data. Now let's subset the data according to the categorical variable:

```
> dat.a <- dat2[dat2$cat == "a",]
> head(dat.a)
```

```
           x1         x2 cat
1 -0.12059626 -1.1568619   a
3  0.89848137  1.5784460   a
4 -0.02901424  0.3402646   a
7  0.43345738  0.6099755   a
8 -0.30285783  0.2025611   a
9 -0.99310898 -1.5098320   a
```

the "==" operator is the Boolean "is equal to" operator. So we are choosing only rows in dat for which cat is equal to "a". We can do the same for the "b"s:

```
> dat.b <- dat2[dat2$cat != "a",]
```

This time we have used the "not equal to" operator, "!=".

Alternatively, we could have used the subset() function:

```
> dat.b <- subset(dat2, cat=="b")
```

Sometimes the subset() function is clearer to understand. The "[" notation is faster but terser. Feel free to use subset() if you want. In R, there are many ways to skin a cat. Suppose we wanted to select all the negative values of "x1" for cat "b":

```
> dat.neg.b <- dat2[dat2$cat == "b" & dat2$x1 < 0,]
> dat.neg.b
```

```
           x1          x2 cat
10 -0.1098492 -1.02734656   b
18 -0.6030111  0.07891244   b
22 -0.9152528  1.22237306   b
```

the output shows us that there are NO rows that meet both these conditions. Here we have used the & operator, which is Boolean "and", and < for "less than". Another option is to use | meaning "or", in place of the &.

The which() function is very useful for locating rows (observations) that have some specified property. For example, if we wanted to know which rows were positive for "x2":

```
> which(dat2$x2 > 0)
```

```
 [1]  2  3  4  5  9 10 11 13 19 20 22 24
```

Finally, we can split the data frame into two parts, based on "cat" values, using the split() function. This makes a new object called "dat.split", which is a list of two components, called "a" and "b". Each component is a data frame, containing the observations for a particular value of "cat".

```
> dat.split <- split(dat2, dat2$cat)
> names(dat.split)
```

```
[1] "a" "b"
```

```
> head(dat.split$a)
```

```
           x1         x2 cat
1 -0.12059626 -1.1568619   a
3  0.89848137  1.5784460   a
4 -0.02901424  0.3402646   a
7  0.43345738  0.6099755   a
8 -0.30285783  0.2025611   a
9 -0.99310898 -1.5098320   a
```

12 Flow of Control

Flow of control refers to the order in which R executes statements. Normally, R proceeds in a linear fashion through the script, executing line by line. However, there are times when we may need R to process differently. The two most common ways to change the flow of control is with if-else constructs and via loops.

12.1 If you have finished your rice, then wash your bowl.

The if-else construct allows R to make decisions about how to proceed with the analysis at runtime. (Runtime is the time after you press the Enter key to run your script). Here is an example of if-else inside a function:

```
> my.func2 <- function (x, a, b) {
 if (x < 0) print(a) else print(b)
 }
> my.func2(7, "x", "y")
```

```
[1] "y"
```

Our function has three arguments, and prints the second argument if the first is negative, else it prints the third argument. If-else constructs can be quite large. You can include multiple statements in each clause using the "}" notation:

```
> z <- 2
> if (is.na(z)) {
 z <- 5
 print(z)} else {
 cat("Here is the output\n")
 cat("z is", z, "\n")
 }

Here is the output
z is 2
```

Here we have introduced the is.na() function to test whether z is a missing value, and the cat() function which can be used to prepare pretty output. The `\n` character is the "newline" character in R.

12.2 Loop the loop

Frequently we may need to execute some commands more than once. For example, in a simulation study we may wish to create random data sets thousands of times. Loops are used to repetitively execute code at runtime. R has several loop constructs. The most commonly used is the "for" loop. Here is an example:

```
> for (i in 1:10) {
 print(i^2)
 }

[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
[1] 36
[1] 49
[1] 64
[1] 81
[1] 100
```

"i" represents the counter index of the loop. We can refer to i inside the loop like any other variable. We can also have "nested" loops:

```
> vec <- vector("numeric")
> for(i in 1:10) {
 for (j in 1:5) {
 vec <- c(vec, i+j)
 }
 }
> vec

 [1]  2  3  4  5  6  3  4  5  6  7  4  5  6  7  8  5  6  7  8  9  6  7  8  9
[25] 10  7  8  9 10 11  8  9 10 11 12  9 10 11 12 13 10 11 12 13 14 11 12 13
[49] 14 15

> length(vec)

[1] 50
```

The inner loop j executes for each value of the outer loop i. So vec has a length of 50. Loops, and especially nested loops, are relatively slow in R. They are best avoided if possible.

It is often possible to write R code so that loops are unnecessary. For example, it's really efficient to do calculations with vectors, rather than individual numbers. This means R goes through each vector 'element-wise' (i.e. element by element, one after the other), and performs the calculation.

```
> x <- vec[1:10]; y <- 1:10  # make vectors of the same length
> z <- x + y  # add element-wise
> z           # element z[i] = x[i] + y[i]

 [1]  3  5  7  9 11  9 11 13 15 17
```

which is much simpler and faster than writing a loop to do the same thing.

13 Graphics

13.1 Graphics Systems

R has three systems for doing graphics, and all of them can produce publication quality plots in several different graphics formats:

1. Base Graphics: This was the original graphics system for R. The approach is to use the graphics functions to build a basic plot, and then use other functions to add finishing touches to the plot. The plot is incrementally constructed by one or several plotting functions.

2. Lattice: Lattice graphics takes a different approach. The entire format for each plot is contained within the one function call. Lattice is especially suited to situations where we want multiple plots from a large data set, particularly multivariate data. Lattice plots can be difficult to prepare because the syntax is difficult, and if a mistake is made, usually no plot is produced.

3. ggplot2: ggplot2 implements the "Grammar of Graphics" approach of Wilkinson (2005). It has features similar to both Base Graphics and Lattice.

We will only consider Base Graphics here. Base Graphics makes extensive use of R's object-oriented features. In particular, the main plotting function, plot(), can operate in several really different ways, depending on the arguments we give it. We will give several examples:

13.2 Scatterplots

Scatterplots in R Base are created using the plot() function on bivariate, continuous data. Consider the following data frame:

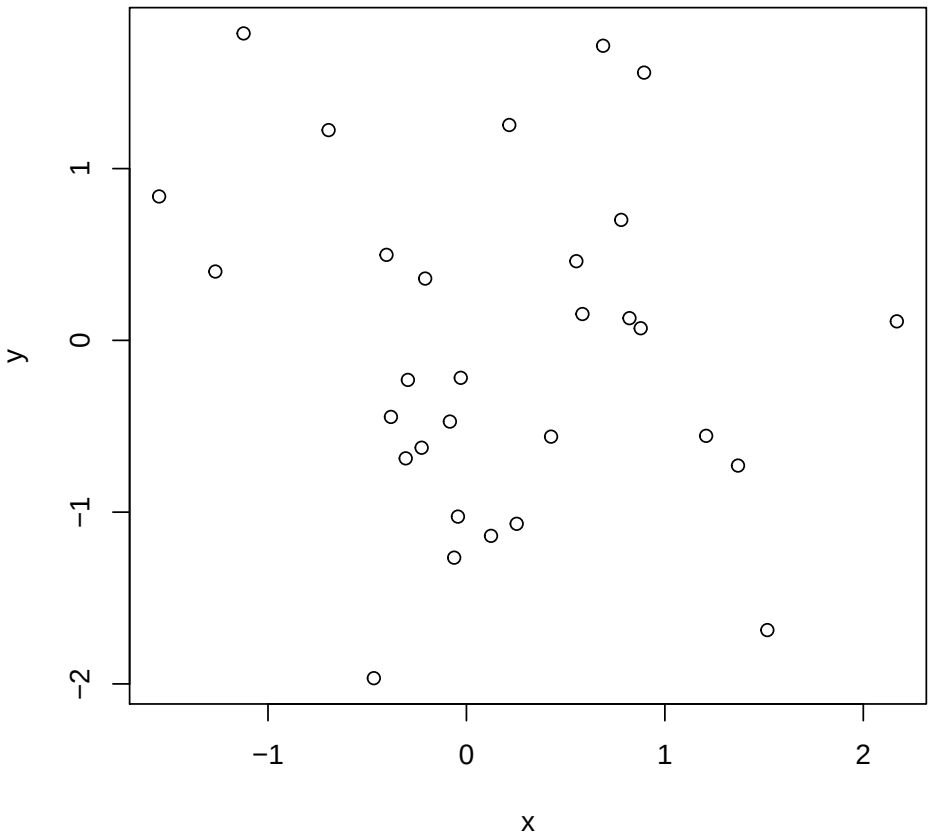
```
> set.seed(123)
> dat <- data.frame(y=rnorm(30), x=rnorm(30), Sex=factor(rep(c("Male", "Female"), each=15)))
> head(dat)

            y          x  Sex
1 -0.56047565  0.4264642 Male
2 -0.23017749 -0.2950715 Male
3  1.55870831  0.8951257 Male
4  0.07050839  0.8781335 Male
5  0.12928774  0.8215811 Male
6  1.71506499  0.6886403 Male
```

The set.seed() function sets the 'seed number' of the inbuilt random number generator in R. If we set the seed number, we will get the same random numbers every time (so your dataset will be the same as in this document). Try NOT setting the seed, or set the seed to a number other than 123. See ?set.seed for further information.

We have two variables, "x" and "y", and they both hold random variates from the standard normal distribution. We also have a factor variable, "Sex", which has two levels, "Male" and "Female", 15 of each. Suppose we wish to draw a scatter plot. We can do this easily:

```
> with(dat, plot(x, y))
```

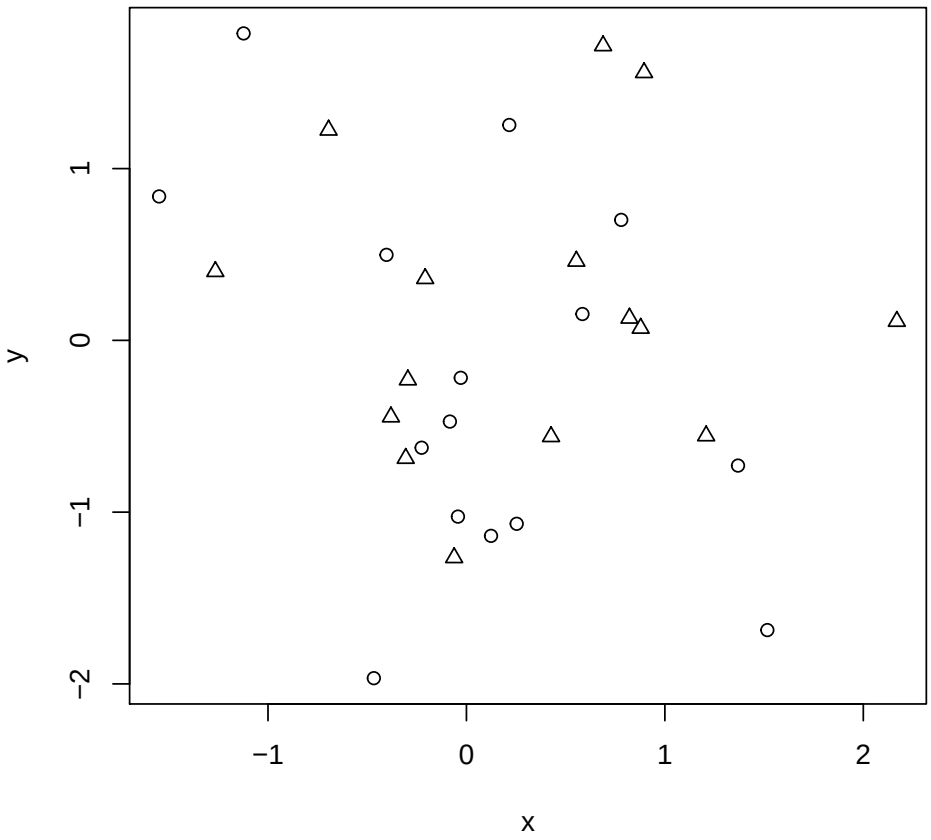


The with() function is a way of telling R what to act on. It uses the dataframe in its first argument, to execute the command given by its second argument. It is a simple way of saying 'look for x and y within dat', as an alternative to using the $ notation. We could have used:

```
> plot(dat$x, dat$y)
```

Notice that our plot is somewhat spartan. We may wish to plot different symbols for the different sexes:

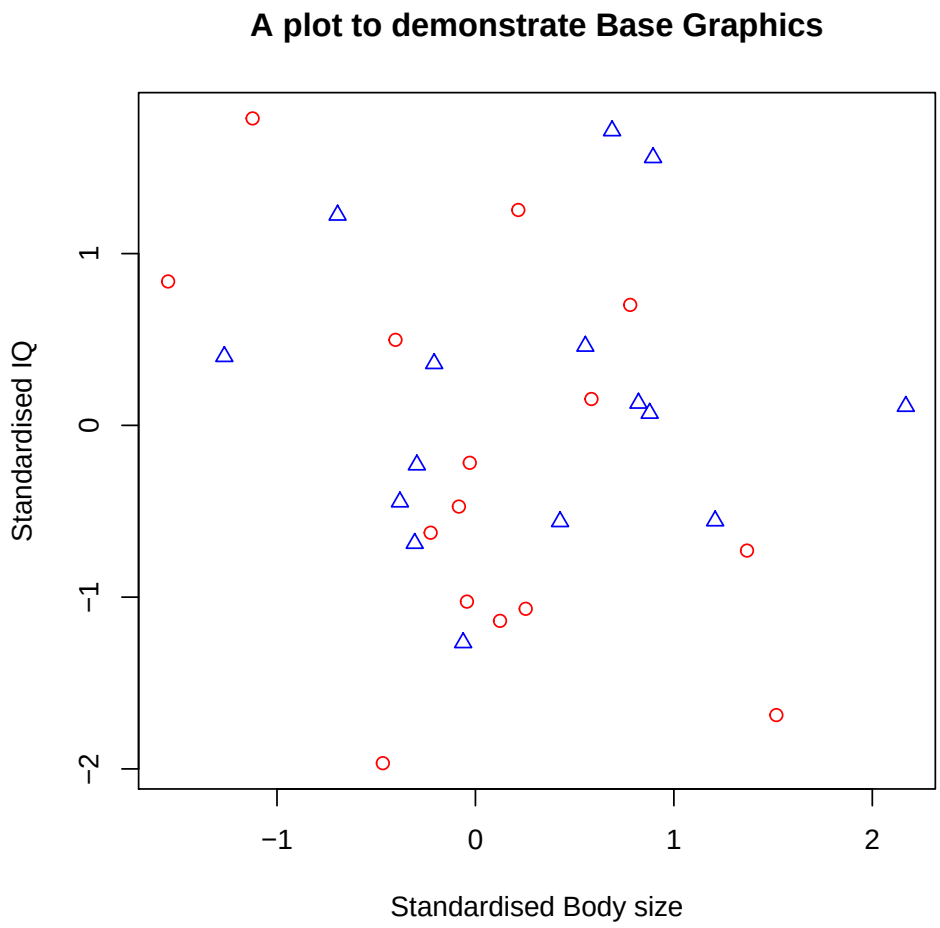
```
> with(dat, plot(x,y, type="n"))
> dat.split <- split(dat, dat$Sex)
> with(dat.split$Female, points(x,y))
> with(dat.split$Male, points(x, y, pch=2))
```



With this plot, we have first drawn the plot, but without plotting the points (using type="n", meaning "none"). Then we used the points() function to draw the two sets of points onto the blank plot, using different symbols for each set. Notice that we did not specify the "pch" argument for the Female points. Not specifying an argument means that the default condition is applied. Here, the default is `pch=1`.

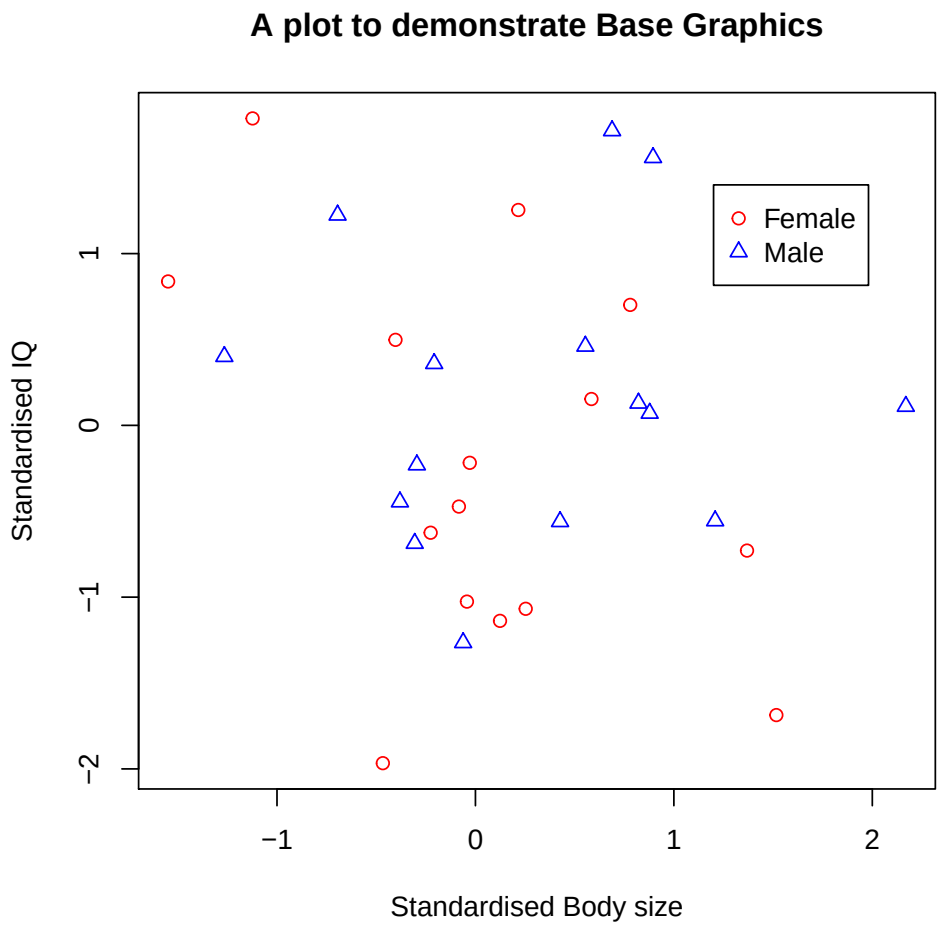
We may wish to put a title on our graph, and give the axes more intelligible names. This is easy. We simply have to adjust the original call to plot():

```
> with(dat, plot(x,y, type="n", xlab="Standardised Body size",
 ylab="Standardised IQ", main="A plot to demonstrate Base Graphics"))
> with(dat.split$Female, points(x,y, col="red"))
> with(dat.split$Male, points(x, y, col="blue", pch=2))
```



Here we have introduced the "xlab" and "ylab" arguments, used to change the x labels and y labels. We have also used the "main" argument to set a main title for the plot. Finally, we can put a legend in the top corner:

```
> with(dat, plot(x,y, type="n", xlab="Standardised Body size",
 ylab="Standardised IQ", main="A plot to demonstrate Base Graphics"))
> with(dat.split$Female, points(x,y, col="red"))
> with(dat.split$Male, points(x, y, col="blue", pch=2))
> legend(1.2,1.4, legend=c("Female", "Male"), pch=1:2, col=c("red", "blue"))
```



13.2.1 Graphics output formats

Once you have plotted your figure, you will want to save it, for example to include in a paper, report, or thesis. One easy way to do this is to simply right-click on the figure and save it to disk. On Windows, this produces a Windows metafile. On a Mac, the graph is converted to pdf. You can control the format of the figure output by setting the graphics device (to a file, rather than a plot window. For details see ?device). For example, the following code produces a pdf file:

```
> pdf("MyFig.pdf")
> set.seed(123)
> with(dat, plot(x,y, type="n", xlab="Standardised Body size",
 ylab="Standardised IQ", main="A plot to demonstrate Base Graphics"))
> with(data.split$Female, points(x,y, col="red"))
> with(data.split$Male, points(x, y, col="blue", pch=2))
> legend(1.2,1.4, legend=c("Female", "Male"), pch=1:2, col=c("red", "blue"))
> dev.off()
```

The pdf() function opens a pdf file in the working directory. From that point in the code, any graphics that are created are sent to that pdf file, until the dev.off() function is called (which turns the "device off"). The file is then closed and you can view it in a pdf viewer such as Acrobat. Experiment with making different types of file formats using the above code. Also play with different values for the arguments until you are familiar with producing R graphics files.

You can produce graphs that are ready to submit with your journal article, by finding out the specifications for each journal. For example, Journal of Ecology dimensions and specifications for a 1-column figure would be:

```
> pdf("Fig.pdf", width=2.8,height=2.8, pointsize=10, bg="white")
```

This is a specific example of the first command above. It creates the PDF file, but it is empty. You would then write your plotting commands, and end with dev.off() to close it.

13.3 Barplots

Barplots are a common plot in scientific papers. They are easy to create in R using the barplot() function. We illustrate it below using simulated data:

```
> set.seed(123)
> dat <- data.frame(z=sample(10:100, 30, replace=TRUE), cats=rep(LETTERS[1:6], each=5))
```

Suppose we want to make a barplot of the means, with error bars representing the standard error. We can calculate the means using the tapply() function. tapply() works by splitting the data (the first argument) according to an index variable (second argument), and then apply a function to each of the splits (the third argument gives the function to perform). So, we want to take the numbers in the column 'z', split by the values in 'cats', and apply the mean function to each split.

```
> mns <- with(dat, tapply(z, cats, mean))
> mns

   A    B    C    D    E    F
69.8 54.8 60.0 54.2 80.0 51.2
```

We can do the same for the standard error of the mean. The formula for the standard error is: $SE_\mu = \frac{s}{\sqrt{N}}$, where s is the sample standard deviation and N is the sample size. R does not have this function built in, but since it is so simple, we can write it ourselves and include it in a call to tapply().

```
> ses <- with(dat, tapply(z, cats, function (x) sd(x)/sqrt(length(x))))
> ses

        A         B         C         D         E         F
11.897058 12.302032 12.759310 16.617461  6.379655  9.409570
```

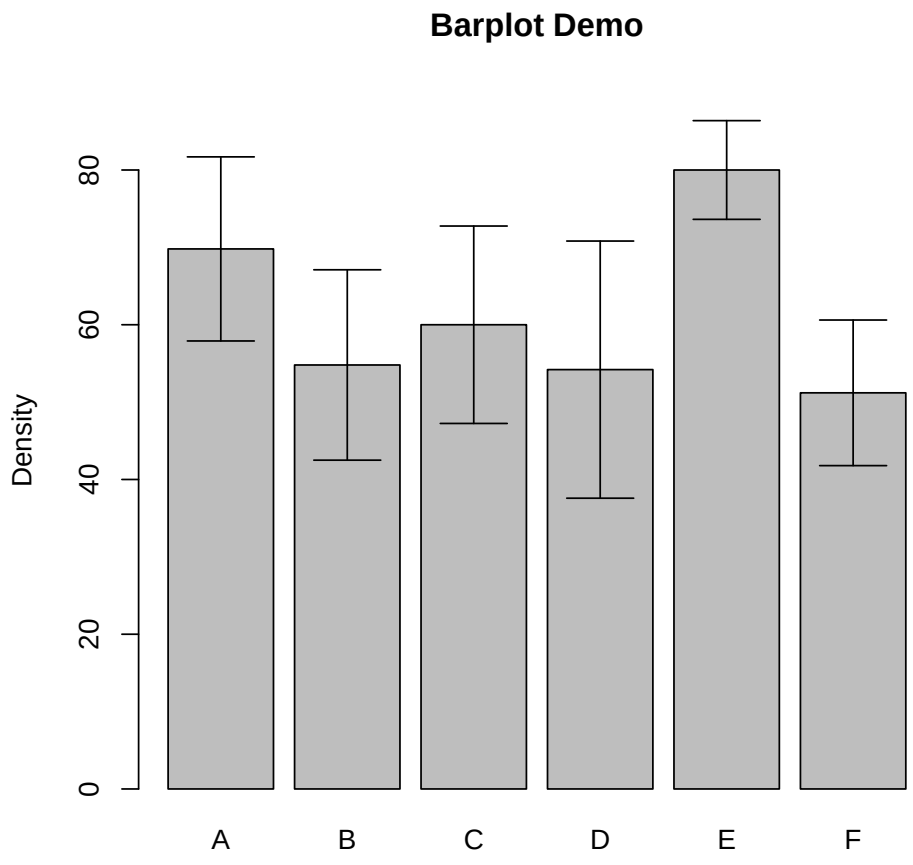
This makes a vector, called 'ses' that contains the SEs. Note that within our call to tapply(), we did not assign an object to the function to calculate SEs. Hence it has no name (it is an *anonymous* function). We just type the definition straight into tapply(). We have introduced three other functions in the definition of SEs: sd() calculates the standard deviation. sqrt() calculates the square root, and length() returns the length of the data vector (5 for each group in this case). Next we need to calculate the upper and lower limits of the error bars, storing them as vectors called 'lower' and 'upper'. Then rbind() binds the two vectors together as rows. ('cbind' would have joined them as columns).

```
> lower <- mns - ses
> upper <- mns + ses
> rbind(lower, upper)

             A        B        C        D        E        F
lower 57.90294 42.49797 47.24069 37.58254 73.62034 41.79043
upper 81.69706 67.10203 72.75931 70.81746 86.37966 60.60957
```

Finally we are ready to make the plot:

```
> my.plot <- barplot(mns, main="Barplot Demo", ylim=c(0,90), ylab="Density")
> arrows(my.plot, mns, my.plot, lower, angle=90, length=.2)
> arrows(my.plot, mns, my.plot, upper, angle=90, length=.2)
```

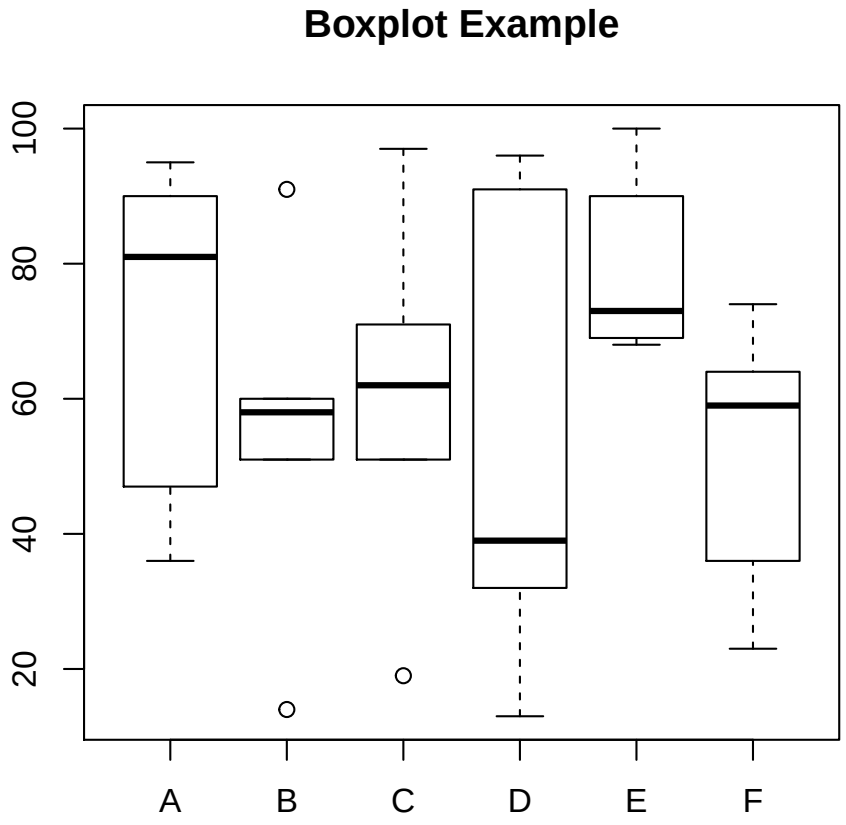


We assigned the barplot() call to an object named my.plot, so that we could store the values it calculates. This is because barplot() is different from other plot commands, as it returns a value, rather than just drawing the plot. The value gives the coordinates of the middle of the bars on the x axis. This enables us to use these coordinates to place the error bars in the centre of the bars. We draw the error bars using arrows(), thinking of them as arrows with "flat" heads (so angle=90). The first four arguments to arrows() are the starting x coordinates, starting y coordinates, ending x coordinates, and ending y coordinates. To get the error bar, we specify angle=90 (try experimenting with other values for angle!). Finally, the width of the error bar should be less than the width of the bars in the barplot, which we specified using length=.2.

13.4 Boxplots

Boxplots, or box-and-whisker plots are a useful plot type that allows comparison of different groups, like a bar graph. However, it provides more information about the distribution of the data within groups. Examine the following code and graph:

```
> boxplot(z~cats, data=dat, main="Boxplot Example")
```



The central dark line on each box is the group median, while the hinges of each box represent the interquartile range. The whiskers represent 1.5 times the length of the box. Outliers are plotted as open circles. See ?boxplot.

13.5 Further graphics

Multiple plots in one Figure - it is possible to draw several plots next to each other, using either par(mfrow=c(2,2)), which tells the number of rows and columns of equal-sized plots, or using layout() to specify multiple plots of different sizes. Alternatively, you can add more data to an existing plot, by using commands such as points(), lines(), or using plot() with the argument "add=TRUE".

Symbols - you can look at all the plotting symbols by making a graph as follows: (You may need to make a larger graphics window, or try writing to PDF using the commands above)

```
> n<-1:25
> x<- rep(1,25)
> plot(x, n, pch=n, axes=F, xlim=c(1,2), main="Symbol pch numbers")
> x2<-x+0.2
> text(x2, n, labels=as.character(n), cex=0.8)
```

Colours - you can view all the R colours on the Color Chart website. Secondly, the package RColorBrewer, has has an interactive interface that helps you choose colours for your graphics: visit http://colorbrewer2.org/

You can get further inspiration and help with graphics from sources such as:

- The package 'Rgraphics' contains data and code from the book "R Graphics" by Paul Murrell. View online.
- Inspiration and code in the R Graph Gallery.

- Graphics Task View on CRAN's"Task Views".

14 Simple statistics

Here we show how to use the above techniques to conduct some simple statistical analyses. In particular, we examine the t-test, the contingency table, and the linear regression. We concentrate on these because they will be familiar to most people and they are particularly easy to do in R, not because you will necessarily use them to analyse your own data.

14.1 The t-test

The two-sample t-test is a common test for the equality of group means when there are two groups. Consider the following data (from Sokal and Rohlf):

```
> dat3 <- data.frame(I=c(7.2, 7.1, 9.1, 7.2, 7.3, 7.2, 7.5),
 II=c(8.8, 7.5, 7.7, 7.6, 7.4, 6.7, 7.2))
> dat3

    I  II
1 7.2 8.8
2 7.1 7.5
3 9.1 7.7
4 7.2 7.6
5 7.3 7.4
6 7.2 6.7
7 7.5 7.2
```

We can perform the t-test as follows:

```
> with(dat3, t.test(I,II))

        Welch Two Sample t-test

data:  I and II
t = -0.1186, df = 11.871, p-value = 0.9076
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.8312357  0.7455214
sample estimates:
mean of x mean of y
 7.514286  7.557143
```

By default, R gives us the Welch 2-sample t-test, which is a version of the t-test that corrects for different variances between groups by fudging the degrees of freedom. If we want the t-test that assumes equal variances, we can use the var.equal argument:

```
> with(dat3, t.test(I,II, var.equal=TRUE))

        Two Sample t-test

data:  I and II
t = -0.1186, df = 12, p-value = 0.9076
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.8302870  0.7445727
sample estimates:
mean of x mean of y
 7.514286  7.557143
```

In this case, there is little difference to the Welch t-test. t.test also has a formula interface, which we can use if the data is in the right format:

```
> dat3a <- stack(dat3)
> dat3a

   values ind
1     7.2   I
2     7.1   I
3     9.1   I
4     7.2   I
5     7.3   I
6     7.2   I
7     7.5   I
8     8.8  II
9     7.5  II
10    7.7  II
11    7.6  II
12    7.4  II
13    6.7  II
14    7.2  II

> my.test <- t.test(values~ind, data=dat3a, var.equal=TRUE)
```

Here we have used the stack() function to shape our data into one column with an indicator variable, as opposed to the two column format used previously.

14.1.1 Contingency tables

Contingency tables are widely used to look for associations between categorical variables when the data are counts. We take an example from Sokal and Rohlf. Contingency tables are eminently suited to representation by matrices, so we build the matrix:

```
> mat <- matrix(c(13,44,25,29), byrow=TRUE, nrow=2,
 dimnames=list( treat=c("Antiserum", "NoAntiserum"),
                state=c("Dead", "Alive")))
> mat

             state
treat         Dead Alive
  Antiserum     13    44
  NoAntiserum   25    29
```

This is an example of a 2 x 2 table, but tables of any size are possible in R. Let's break up the command to understand it: we have made a matrix out of the 4 numbers, with two rows (this also means there will be two columns, since we only have 4 numbers). We have also given it "dimnames". These are the names for our dimensions - the row names are first, and then the column names.

To test for an association between variables state and treat, we can do:

```
> tst <- chisq.test(mat)
> tst

        Pearson's Chi-squared test with Yates' continuity correction

data:  mat
X-squared = 5.7923, df = 1, p-value = 0.0161

> names(tst)

[1] "statistic" "parameter" "p.value"   "method"    "data.name" "observed"
[7] "expected"  "residuals" "stdres"

> expected <- tst$expected
> expected
```

```
             state
treat              Dead    Alive
  Antiserum   19.51351 37.48649
  NoAntiserum 18.48649 35.51351
```

```
> resids <- tst$residuals
> resids
```

```
             state
treat               Dead     Alive
  Antiserum   -1.474510  1.063844
  NoAntiserum  1.514915 -1.092996
```

Here we have performed the test and assigned it to the object tst. Assigning analyses to objects is usually the best way to conduct data analysis in R. This is because just printing the object does not necessarily provide all the information about the analysis that you may want. In this case, we used the names() function to examine the objects listed in "tst". Then we extracted the expected values and the Pearson residuals using the \$ notation.

14.1.2 Ordinary Linear Regression

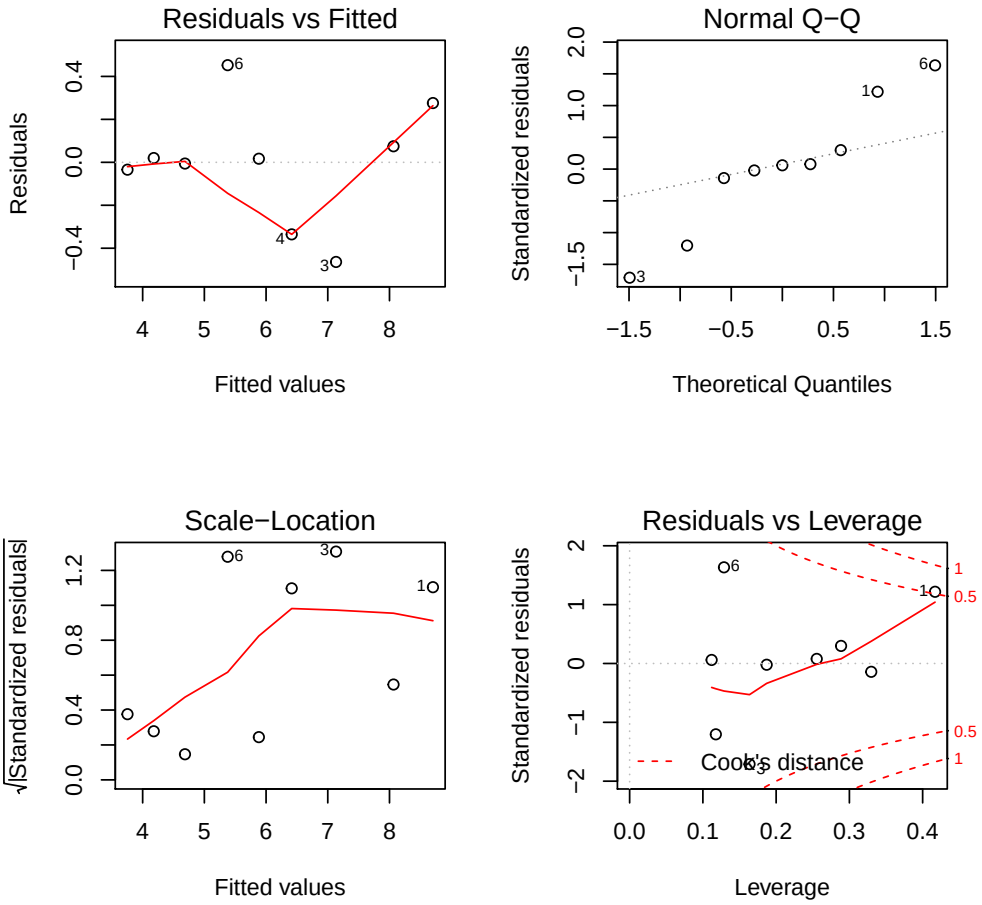
Performing a linear regression is a more complicated procedure, usually because we would also like to plot the results (with confidence bands) and check for departures from the assumptions. Again we use data from Sokal and Rohlf:

```
> dat4 <- data.frame(wt.loss=c(8.98,8.14,6.67,6.08,5.90,5.83,4.68,4.20,3.72),
 humid=c(0, 12.0, 29.5, 43.0, 53.0, 62.5, 75.5, 85.0, 93.0))
> dat4
```

```
  wt.loss humid
1    8.98   0.0
2    8.14  12.0
3    6.67  29.5
4    6.08  43.0
5    5.90  53.0
6    5.83  62.5
7    4.68  75.5
8    4.20  85.0
9    3.72  93.0
```

We fit the regression model using the lm() function, and examine the diagnostic plots:

```
> fit <- lm(wt.loss ~ humid, data=dat4)
> par(mfrow=c(2,2))
> plot(fit)
```

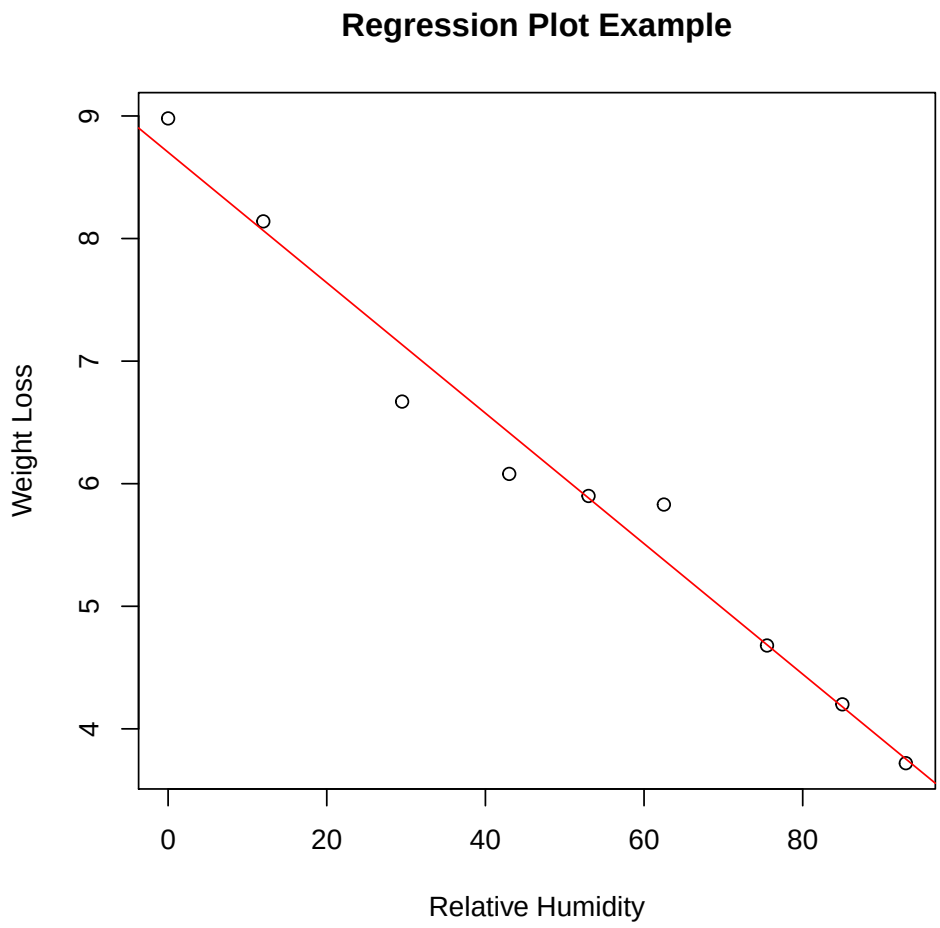


Here we used the par() function to set the number of plots in each figure. par() is a very useful function. It has a lot of arguments which can change the look of your plots. See ?par. Setting `par(mfrow=c(2,2))` causes R to plot the next four plots on the same graphics device, in a 2 x 2 layout.

We then plot the diagnostics of the fitted object (called "fit"). Notice that we are using the plot() function, but it behaves very differently when used on lm() objects, compared to its use for scatterplots. The first diagnostic plot is the fitted values *versus* the residuals. We expect there to be no relationship between them. The sample size is small, but it looks as if the assumption of no relationship appears to hold. The next plot is the Normal Quantile-Quantile plot (QQ plot). If the residuals are truly distributed according to the Normal distribution, the values should all fall along the straight line. There is some evidence for skew. The other two plots are less important in determining model fit. The third plot is useful in detecting relationships between the residuals and the fitted values in the presence of skew. The fourth plot is an Influence plot. It plots the residuals versus their leverage, which can be used to assess the impact of outliers and other influential points.

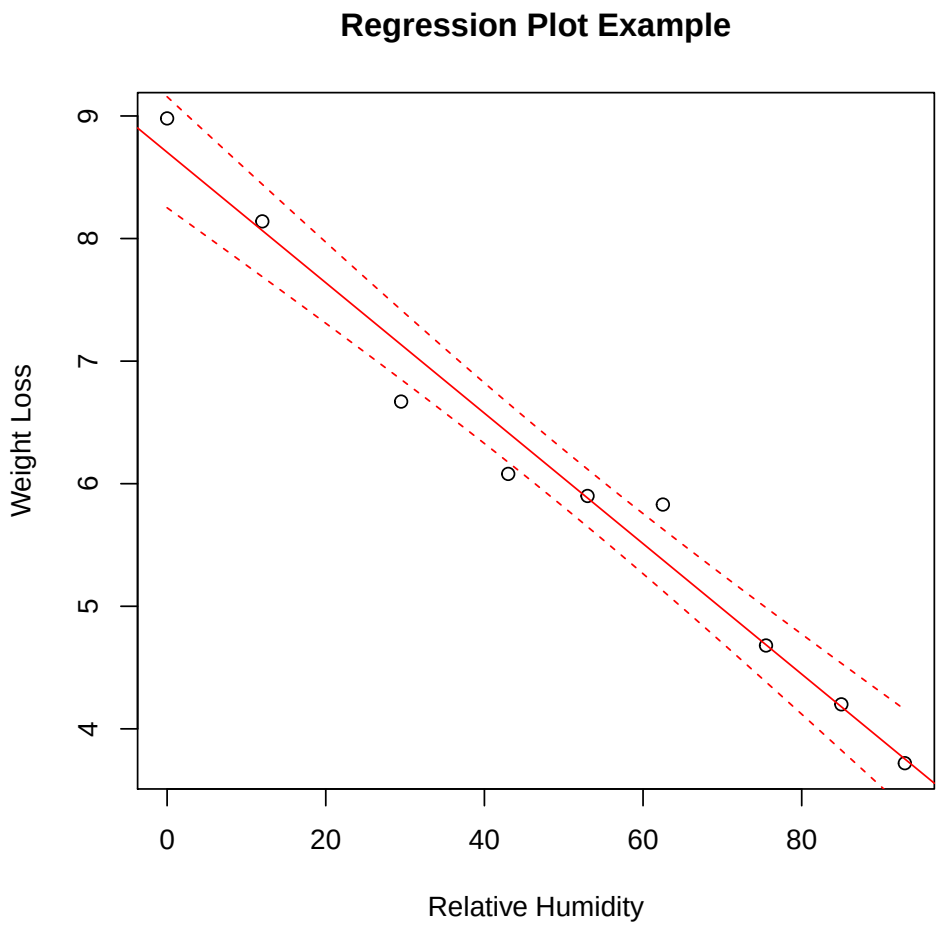
Now we have enough information to plot the data and the regression line:

```
> with(dat4, plot(wt.loss~humid, main="Regression Plot Example",
 xlab="Relative Humidity", ylab="Weight Loss"))
> abline(fit, col="red")
```



We need to do a bit more work to plot the 95% confidence bands. In particular, we need to construct a new dummy data set and use the predict() function:

```
> newdat <- data.frame(humid=seq(min(dat4$humid), max(dat4$humid), length.out=150))
> preds <- predict(fit, newdat, interval="confidence")
> newdat <- cbind(newdat, preds)
> with(dat4, plot(wt.loss~humid, main="Regression Plot Example", xlab="Relative Humidity",
 ylab="Weight Loss"))
> abline(fit, col="red")
> with(newdat, {
 lines(humid,lwr, lty=2, col="red")
 lines(humid,upr, lty=2, col="red")
 })
```



Here we have introduced several new functions:

- seq() produces a *sequence* of numbers from a minimum to a maximum. We specified 150 equally spaced numbers, over the range of values of humid. We used min() and max() to calculate these bounds.
- predict() can be used on many different types of object, and produce different results (similar to plot() which draws different plots depending on the object). In this case, predict() will predict the values for 'weight loss' based on our new values of humid given in the new data frame called newdat. It also provides upper and lower confidence limits for the values (95% confidence intervals by default).
- abline() simply draws the regression line calculated by fit onto the current graphics device.
- lines() is used to draw the 95% confidence bands onto the plot. We set `lty=2`, which produces dashed lines.

This example shows how closely the statistical analysis and graphical visualisation work together, by using the analysis to provide information for the plot. Finally, we can use the summary() function to examine the regression model in detail:

```
> summary(fit)

Call:
lm(formula = wt.loss ~ humid, data = dat4)

Residuals:
     Min       1Q   Median       3Q      Max
-0.46397 -0.03437  0.01675  0.07464  0.45236

Coefficients:
```

```
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  8.704027   0.191565   45.44 6.54e-10
humid       -0.053222   0.003256  -16.35 7.82e-07

Residual standard error: 0.2967 on 7 degrees of freedom
Multiple R-squared:  0.9745,        Adjusted R-squared:  0.9708
F-statistic: 267.2 on 1 and 7 DF,  p-value: 7.816e-07
```

15 Quitting R

Type:

```
> q()
```

R will ask if you want to save the workspace. As described above (under 'The Workspace'), this means saving the objects in your workspace into a file on your hard disk, in you working directory.

16 A Demonstration

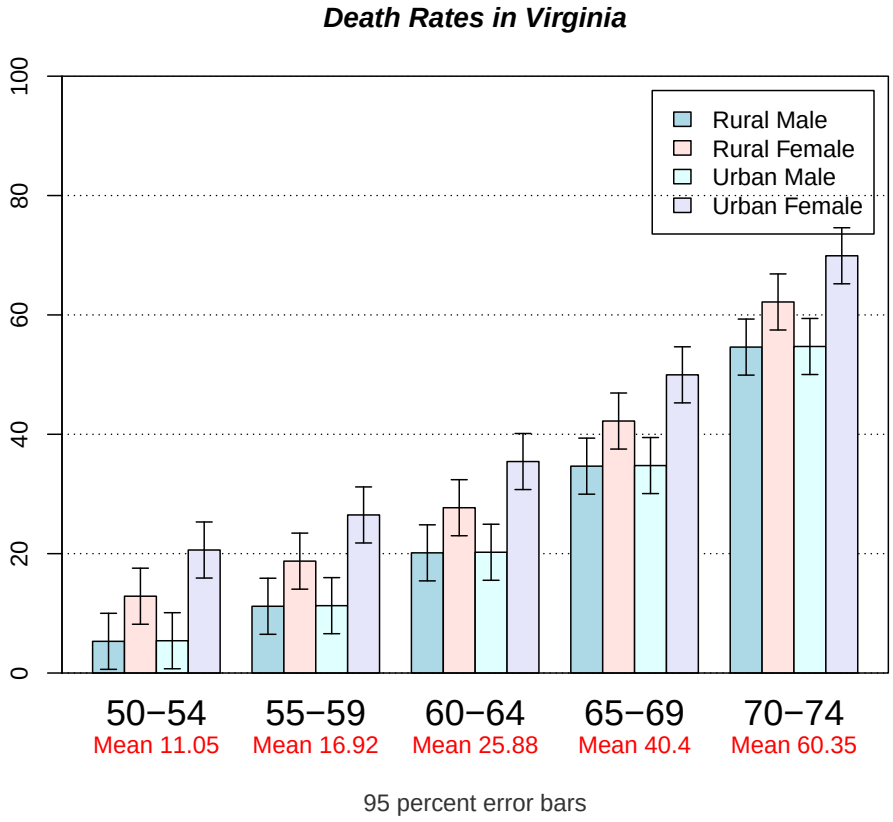
To give you an idea of what a typical R session may look like, we will look at the death rates in Virginia, USA, classified by age and sex/location. This example is just to give you a taste of what is possible (and even easy) with R. The data set is from the gplots package. Lines beginning with a hash symbol (#) are comments and are ignored by R, but they should give you an idea of what is going on in the analysis. The first part of the analysis is to fit a simple two factor ANOVA without the interaction term. The second part of the analysis produces and extracts predicted means and confidence intervals for the predictions. The third part draws the graph.

```
# PART 1: first install the necessary packages, and then load them
# install.packages(c("gplots","reshape","xtable"), repos="http://cran.csiro.au/")
library(gplots)
library(reshape)
library(xtable)
# reshape the data appropriately
mlt <- melt(VADeaths)
names(mlt) <- c("Age", "Category", "DeathRate")
# fit the anova model and get output
fit <- aov(DeathRate ~ Age + Category, data=mlt)
xtable(anova(fit))
```

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Age	4	6288.50	1572.12	135.35	0.0000
Category	3	797.32	265.77	22.88	0.0000
Residuals	12	139.38	11.61		

```
# PART 2: construct a new data frame for the predicted values
newdat <- expand.grid(Age=levels(mlt$Age), Category=levels(mlt$Category))
# get the predicted values and 95% CI
preds <- predict(fit, newdat, interval="confidence")
preds <- cbind(newdat, preds)
# reshape the data and extract predictions, lower and upper CI.
mlt <- melt(preds)
ci.l<- as.matrix(cast(mlt, Category~Age, subset=variable=="lwr"))
ci.u <- as.matrix(cast(mlt, Category~Age, subset=variable=="upr"))
prd3 <- as.matrix(cast(mlt, Category~Age, subset=variable=="fit"))
# PART 3: Draw the graph
mybarcol <- "gray20"
mp <- barplot2(prd3, beside = TRUE,
        col = c("lightblue", "mistyrose",
```

```
            "lightcyan", "lavender"),
        legend = colnames(VADeaths), ylim = c(0, 100),
        main = "Death Rates in Virginia", font.main = 4,
        sub = "95 percent error bars", col.sub = mybarcol,
        cex.names = 1.5, plot.ci = TRUE, ci.l = ci.l, ci.u = ci.u,
        plot.grid = TRUE)
mtext(side = 1, at = colMeans(mp), line = 2,
      text = paste("Mean", formatC(colMeans(prd3))), col = "red")
box()
```



17 R in your publications

So you've run your analyses and produced figures, and need to cite R in your thesis or paper? You can say "statistical analyses were performed in R 3.11.0 (R Core Development Team, 2014)" and also reference any specific packages you have used. To find the correct current citations, use the function citation() for citing R itself (i.e. with empty brackets), and citation("packagename") to access this information for particular packages.

Thanks and good luck! Learning R is a really valuable skill and we hope this has been a helpful starting point.