Introduction to Biological Statistics

Imperial College London

Samraat Pawar (s.pawar@imperial.ac.uk),
with inputs from Steve Cook and David Orme

February 19, 2016

Contents

Chapter 0

Introduction

> Donald Knuth, 1995: *Science is what we understand well enough to explain to a computer. Art is everything else we do*

0.1 What is this document about?

This document contains the content of the Imperial College London Department of Life Sciences BSc Year 1 (Y1) and 2 (Y2) statistics modules, including workshops and practicals. Different subsets of the content of this document will be covered in Y1 and Y2. You will be given instructions about which sections will be covered in your respective (Y1 or Y2) modules. You will learn statistics with R in a hands-on on way in this module. A few "pure" lectures on statistical concepts will also be delivered, but you will find all that they cover in these notes as well.

This document is accompanied by data and code on which you can practice your skills in your own time and during the Practical sessions. These materials are available (and will be updated regularly) at bitbucket (it's like GitHub): `https://bitbucket.org/mhasoba/iccompbiostat`. They will *not* be available on blackboard. Why am I doing it this way? Because these notes and the associated data and R code are constantly evolving, and blackboard is just not built to handle that sort of updating. That is, I am treating this course as any computing project that needs to be regularly updated and improved. Changes to the notes and content will also be made based upon your feedback and how I see y'all progressing.

Note that you can download the code, data and everything from the bitbucket repository at one go, by going to `https://bitbucket.org/mhasoba/iccompbiostat/downloads` and then clicking on the "Download repository" link. You can then unzip the downloaded .zip and grab the files you need.

These notes and the scripts that you create form a valuable reference for using and interpreting statistics. You will be expected to be able to use your experience and these notes to complete analyses for statistical components of other courses, in the second and third year in particular.

It is also important for you to note that this document does not tell you every single thing you need to know to perform the exercises in it. In programming, you learn far more from trying to solve problems than from reading about how others have solved them.

Finally, it is important that you work through the problems in each chapter, particularly as some of the questions ask you to find out about commands and functions not introduced in the chapter's

text itself, but which will be relied on in later chapters. By the time you have attended the lectures and workshops, and completed the exercises in this document, you should able to:

- Design experiments that produce data which can be analysed using R
- Be able to export spread-sheet data into a form accessible by R, import data from such files into R, handle data in R vectors and data-frames, and have some appreciation of R as a general purpose programming language.
- Create properly labelled graphs using R (scatterplots, histograms, box-and-whisker plots, etc.)
- Use R to perform common statistical tests, particularly t, F, and χ^2 tests
- Develop ability to build, criticise and interpret linear models, especially linear regression and ANOVA.
- Interpret R output (particularly p values) correctly, and use them appropriately in practical write-ups and presentations

0.2 Why R?

R is a freely available statistical software with strong programming capabilities. R is becoming very popular in biology, mainly due to two factors: i) many packages are available to perform all sorts of statistical and mathematical analysis, and ii) it can produce beautiful graphics.

There are many "canned", "black box" statistics (minitab, SPSS, etc) programs in the world that appear quite warm and friendly. Why not just use them? Here are some very good reasons:

- It provides basically every statistical test you'll ever need and is constantly being improved.
- It is scriptable — it can provide a perfectly repeatable record of your analysis.
- It can produce publication-quality graphics.
- It is available for all common computer operating systems.
- It is free – if you want a copy on your laptop, help yourself at the CRAN website.

Indeed, R is very widely used by professional scientists both around the world and across Departments at Impe rial College.

Being able to program in R means you can develop and automate your own data handling, statistical analysis and generation of figures, a set of skills you are likely to need in many, if not most careers paths!

0.3 Conventions used in this document

You will find all R commandline/console arguments, code snippets and output in colored boxes like this:

```
> ls()
```

Here $>$ is the R prompt, and will type (or copy-paste, but not recommended!) the commands/code that you see from this document into the R command line. But please exclude the $>$ as this is just

the R command prompt! I have included the > prompts in the code shown in this document so that you are forced to see what each line is doing. Indeed, avoid copying-and-pasting chunks of R code you do not understand: blindly shovelling data into a black box and assuming the output is correct and meaningful will eventually lead to frustrations, and if you are unlucky, embarrassments or even catastrophes!

I have aimed to make the R programming content of this document computer platform (Mac, PC or Linux) independent because many of you will probably also later be working with R on your personal laptops or desktops. Indeed, platform-indepndence of your statistical analysis code is one of the main reasons why you are using R!

Finally, note that:

⋆ In all subsequent chapters, lines marked with a star like this are things for you to do.

0.4 A note on spreadsheets – beware the dark forces

You will NOT be using spreadsheets (e.g., Excel) in the stats modules. There are times when you will feel the pull of the dark side (ahem!), and imagine a better (more "comfortable") world where you are mouse-clicking your way happily though Excel-based data manipulations and analyses. NO! You will be doing yourself a disservice. On the long-ish run you will be much better off visualizing and manipulating data in R. This is something you will learn, my young *padawan*!

0.5 Keep your workflow organized!

In the following chapters, you will practice many examples where you are required to write large blocks of R code or functions (what's this? — you'll see soon!). Please get into the habit of writing these into text files with an `⋆.R` extension (more on this in Chapter 1). Furthermore, please keep all your code files organized in one or more directories (e.g., named `Code`!). Similarly, some of these scripts will take data files as inputs, and output some results in the form of text or graphics. Please keep these inputs and outputs organized as well, in separate directories (e.g., named `Data` and `Results`) respectively. Me and your demonstrators will help you get set up and abide by this "workflow"!

0.6 Assessment of the stats modules

Assessment will be through a computer-based test. You will not be expected to demonstrate deep R skills in the test, but will be expected to be able to apply the concepts you have learnt to address statistical questions and to interpret R input and output. The Y1 assessment will cover Chapters 1–6. The Y2 assessment will cover all chapters (1–10).

0.7 Readings

Look up the Readings directory on the bitbucket repository (link given above).

- Bolker, B. M.: Ecological Models and Data in R (eBook and Hardcover available).

- Beckerman, A. P. & Petchey, O. L. (2012) Getting started with R: an introduction for biologists. Oxford, Oxford University Press.
  Good, short, general introduction

- Crawley, R. (2013) The R book. 2nd edition. Chichester, Wiley.
  Excellent but enormous reference book, code and data available from `www.bio.ic.ac.uk/research/mjcraw/therbook/index.htm`

- Use the internet! Type "R tutorial", and scores will pop up. Choose one that seems the most intuitive to you.

Chapter 1

Introduction to R

1.1 Outline of the the R module

The content and structure of the chapters in the R module are geared towards the following objectives:

- Give you an introduction to R syntax and programming conventions, assuming you have never set your eyes on R or any other programming language before — we will breeze through this as you have already been introduced to R in the Stats Week!
- Teach you principles of clean and efficient programming in R, including delightful things like vectorization and debugging.
- Teach you how to generate publication quality graphics in R — publication quality is thesis quality!
- Teach you how to develop reproducible data analyses "work flows" so you (or anybody else) can run and re-run your analyses, graphics outputs and all, in R.

You will use R a lot during the rest of your courses and your thesis dissertations, and likely even your career — our aim is to lay down the foundations for you to become very comfortable with it.

1.2 What is `R`?

R is a freely available statistical software with strong programming capabilities. R is becoming very popular in biology, mainly due to two factors: i) many packages are available to perform all sorts of statistical and mathematical analysis, and ii) it can produce beautiful graphics.

There are many "canned", "black box" statistics (minitab, SPSS, etc) programs in the world that appear quite warm and friendly. Why not just use them? Here are some very good reasons:

- It provides basically every statistical test you'll ever need and is constantly being improved.
- It is scriptable — it can provide a perfectly repeatable record of your analysis.
- It can produce publication-quality graphics.
- It is available for all common computer operating systems.
- It is free – if you want a copy on your laptop, help yourself at the CRAN website.

Indeed, R is very widely used by professional scientists both around the world and across Departments at Imperial College.

Being able to program in R means you can develop and automate your own data handling, statistical analysis and generation of figures, a set of skills you are likely to need in many, if not most careers paths!

1.3 Would you ever need anything other than R?

Being able to program R means you can develop and automate your statistical analyses and the generation of figures into a reproducible work flow. However, if your work also includes extensive numerical simulations, manipulation of very large matrices, bioinformatics, or includes database access and manipulations, you will be much better off if you *also* know another programming language that is more versatile and computationally efficient (like `python`, `perl` or `C`).

But for many/most of you, R will do the job!

1.4 Installing R

Linux/Ubuntu: run the following in terminal

```
sudo apt-get install r-base r-base-dev
```

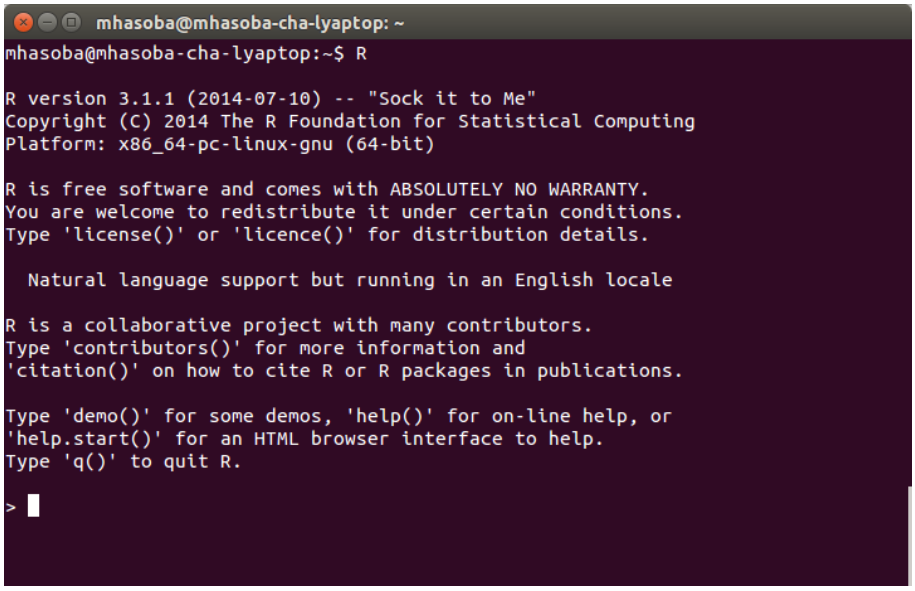
Mac OS X: download and install from `http://cran.r-project.org/bin/macosx/`

Windows: download and install from `http://cran.r-project.org/bin/windows/base/`

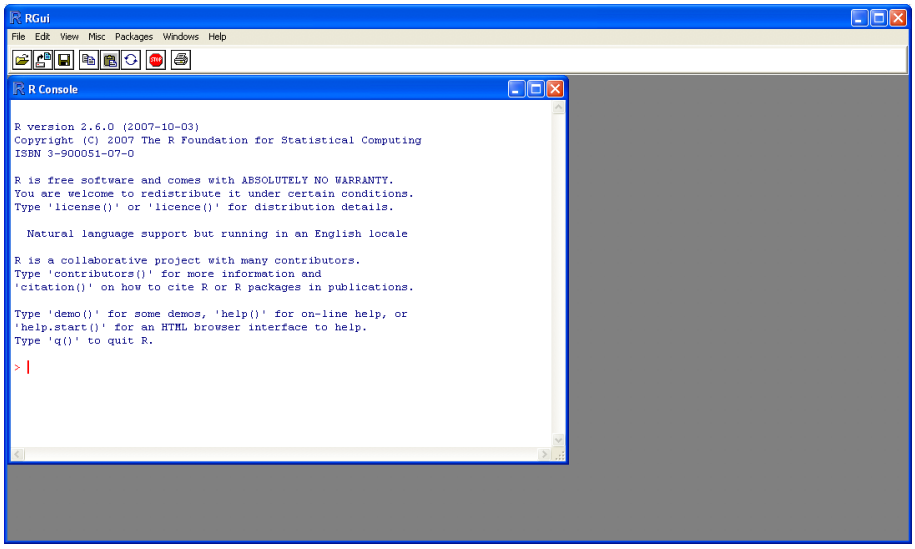
1.5 Getting started

We will start with the bare-bones R interface – you may switch to RStudio or something else at your own discretion (see section below).

Launch R (From Applications menu on Window or Mac, from terminal in Linux/Ubuntu) — it should look something like this (on Linux/Ubuntu or Mac terminal):



Or like this (Windows "console", similar in Mac):



You can also use an IDE (Interactive Development Environment) that can offer delights like syntax highlighting (google it!), such as RStudio, `geany`, `vim`, etc.

1.6 RStudio

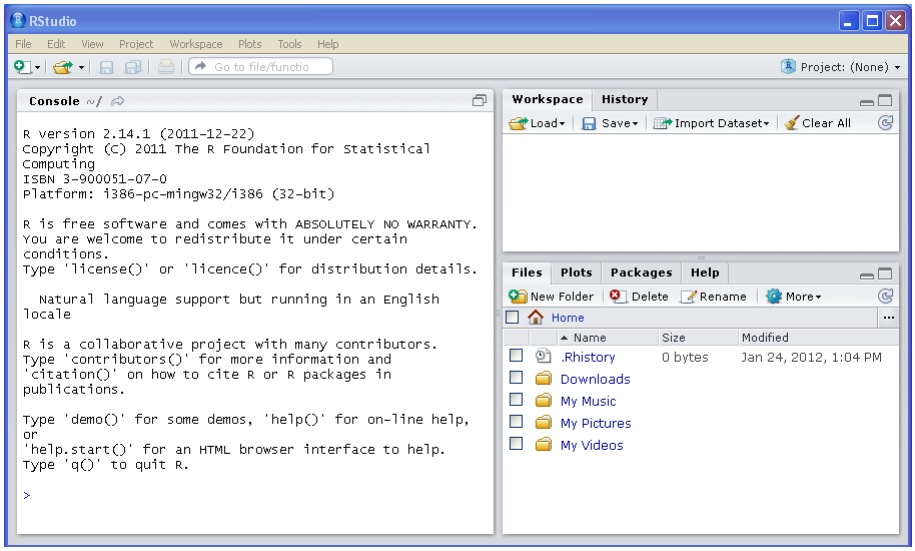
One of the reasons that R has become so popular is that there are of some very nice, freely available graphic user interfaces (GUI's) for it. In particular the program RStudio is excellent, and available on your college Desktops.

A big advantage of something like RStudio is that you will get "syntax highlighting" wherein R language elements such as variables, commands, and brackets are differently colored. This is very handy and will make your R programming far more convenient and error-free. If you are using your own desktop/laptop, yo can download it freely from `http://rstudio.org/` and install (versions are available for all platforms).

These practicals will not assume you are using RStudio (everything will be command line based), but I would encourage you to use it eventually, if not immediately.

Of course, if you already are doing so, you might want to continue using your multi-code editor such as geany or emacs!

Go to your Windows menu and launch RStudio (assuming it is available on your computer).



One one side is the same old R terminal. You are still in the R environment, but with some useful frills and additional thrills. Here you can also feed your mouse habit, but I strongly suggest using the terminal at the bottom, and resist, to the extent possible, the temptation to do everything using the shiny buttons you see (that is the dark side pulling, my young *padawan*).

1.7 Some Basics

Gets get started with some R basics. You will be working by entering R commands interactively at the R user prompt (>). Up and down arrow keys scroll through your command history.

1.7.1 Useful R commands

`ls()`	list all the variables in the work space
`rm('a', 'b')`	remove variable(s) `a` and `b`
`rm(list=ls())`	remove all variable(s)
`getwd()`	get current working directory
`setwd('Path')`	set working directory to `Path`
`q()`	quit R
`?Command`	show the documentation of `Command`
`??Keyword`	search the all packages/functions with `Keyword`, "fuzzy search"

1.7.2 R Warm-up

Now, try out the following in the R console:

```
> a <- 4  # store 4 as variable a
> a
[1] 4
> a*a  # product
[1] 16
> a_squared <- a*a
> sqrt(a_squared) # square root
[1] 4
> v <- c(0, 1, 2, 3, 4) # c: "concatenate"
```

`c()` (concatenate) is one of the most commonly used functions — it will appear again and again! (try `?c`).

Note that any text after a "#" is ignored by R — handy for commenting. *In general, please comment your code and scripts, for everybody's sake.*. You will be amazed by how difficult it is to read and understand what a certain R script does (or any other script, for that matter) without judicious comments — even scripts written by yourself.

```
> v # Display the vector variable you created
[1] 0 1 2 3 4
> is.vector(v) # check if it's a vector
[1] TRUE
> mean(v) # mean
```

A vector is like a single column or row in a spreadsheet.

```
[1] 2
> var(v) # variance
[1] 2.5
> median(v) # median
[1] 2
> sum(v) # sum all elements
[1] 10
> prod(v + 1) # multiply
[1] 120
> length(v) # length of vector
[1] 5
```

1.7.3 Variable names and Tabbing

In R, you can name variables in the following way to keep track of related variables:

```
> wing.width.cm <- 1.2 #Using dot notation
> wing.length.cm <- c(4.7, 5.2, 4.8)
```

This can be handy; type:

```
> wing.
```

And then hit the `tab` key. This is nice, but good style and readability is more important than just convenient variable names. Variable names should be as obvious as possible, not over-long!

1.7.4 E Notation

If you are not used to different representations of long numbers, the E notation might be confusing. R will use this notation in outputs of statistical tests to display very large or small numbers. Try this:

```
> 1E4
[1] 10000
> 1e4
[1] 10000
> 5e-2
[1] 0.05
> 1E4 ^ 2
[1] 1e+08
> 1 / 3 / 1e8
[1] 3.333333e-09
```

1.7.5 Operators

The usual operators are available in R:

+	Addition
−	Subtraction
*	Multiplication
/	Division
^	Power
%%	Modulo
%/%	Integer division
==	Equals
!=	Differs
>	Greater
>=	Greater or equal
&	Logical and
|	Logical or
!	Logical not

1.7.6 When things go wrong

Syntax errors are those where you've just made a typing mistake. Here are some common problems:

- missing close bracket leads to continuation line.

```
> x <- (1 + (2 * 3)
+
```

  Hit Ctrl C (see below) or keep typing!

- Too many parentheses: `2 + (2*3))`
- wrong/mismatched brackets (see next subsection)

- Do not mix double quotes and single quotes
- When things seem to take too long, try `Ctrl + C`

1.7.7 Types of parentheses

R has a somewhat confusing array of parentheses that you need to get used to:

- `f(3,4)` – call the function f, with the arguments 3 & 4.
- `a + (b*c)` – use to enforce order over which statements or calculations areexecuted.
- { expr1; expr2; ...exprn } – group a set of expressions into one compound expression. Value returned is value of last expression; used in looping/conditionals.
- `x[4]` – get the 4th element of the vector x.
- `l[[3]]` – get the 3rd element of some list l, and return it. (compare with l[3] which returns a list with just the 3rd element inside) (more on lists in next section)

1.8 Data types

R comes with data-types. Mastering these will help you write better, more efficient programs and also handle diverse between datasets. Now get back into R (if you quit R using `q()`).

1.8.1 Vectors

Vectors are a fundamental object for R. Scalars (single numbers) are treated as vector of length 1. *A vector is like a single column or row in a spreadsheet.* Try this:

```
> a <- 5
> is.vector(a)
[1] TRUE
> v1 <- c(0.02, 0.5, 1)
> v2 <- c("a", "bc", "def", "ghij")
> v3 <- c(TRUE, TRUE, FALSE)
```

1.8.2 Matrices and arrays

R has many functions to manipulate matrices (two-dimensional vectors) and arrays (multi-dimensional vectors).

```
> mat1 <- matrix(1:25, 5, 5)
> mat1
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11   16   21
[2,]    2    7   12   17   22
[3,]    3    8   13   18   23
[4,]    4    9   14   19   24
[5,]    5   10   15   20   25
> mat1 <- matrix(1:25, 5, 5, byrow=TRUE)
> mat1
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
[3,]   11   12   13   14   15
[4,]   16   17   18   19   20
[5,]   21   22   23   24   25
> dim(mat1) #get the size of the matrix
[1] 5 5
```

Make an array consisting of two 5×5 matrices containing the integers 1–50:

```
> arr1 <- array(1:50, c(5, 5, 2))
> arr1
, , 1

     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11   16   21
[2,]    2    7   12   17   22
[3,]    3    8   13   18   23
[4,]    4    9   14   19   24
[5,]    5   10   15   20   25

, , 2

     [,1] [,2] [,3] [,4] [,5]
[1,]   26   31   36   41   46
[2,]   27   32   37   42   47
[3,]   28   33   38   43   48
[4,]   29   34   39   44   49
[5,]   30   35   40   45   50
```

1.8.3 Data frames

This is a very important data type in R. It is great for storing data in which each column can contain a different data type (e.g., numbers, strings, boolean), just like a standard spreadsheet. Indeed, the dataframe data type was built to emulate some of the convenient properties of spreadsheets. Many statistical and plotting functions and packages in R naturally use data frames. Let's build and manipulate a dataframe:

```
> Col1 <- 1:10
> Col1
 [1]  1  2  3  4  5  6  7  8  9 10
> Col2 <- LETTERS[1:10]
> Col2
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"
> Col3 <- runif(10) # 10 random num. from uniform
> Col3
 [1] 0.29109 0.91495 0.64962 0.95503 0.26589 0.02482 0.59718
 [8] 0.99134 0.98786 0.86168
> MyDF <- data.frame(Col1, Col2, Col3)
> MyDF
   Col1 Col2      Col3
1     1    A 0.2910981
2     2    B 0.9149558
3     3    C 0.6496248
4     4    D 0.9550331
5     5    E 0.2658936
6     6    F 0.0248217
7     7    G 0.5971868
8     8    H 0.9913407
9     9    I 0.9878679
```

```
10   10    J 0.8616854
> names(MyDF) <- c("A.name", "another", "another.one")
> MyDF
     A.name another another.one
1        1       A   0.2910981
2        2       B   0.9149558
3        3       C   0.6496248
4        4       D   0.9550331
5        5       E   0.2658936
6        6       F   0.0248217
7        7       G   0.5971868
8        8       H   0.9913407
9        9       I   0.9878679
10      10       J   0.8616854
> MyDF$A.name
 [1]  1  2  3  4  5  6  7  8  9 10
> MyDF[,1]
 [1]  1  2  3  4  5  6  7  8  9 10
> MyDF[c("A.name","another")]
     A.name another
1        1       A
2        2       B
3        3       C
4        4       D
5        5       E
6        6       F
7        7       G
8        8       H
9        9       I
10      10       J
> class(MyDF)
[1] "data.frame"
> str(MyDF) # a very useful command!
'data.frame':   10 obs. of  3 variables:
 $ A.name     : int  1 2 3 4 5 6 7 8 9 10
 $ another    : Factor w/ 10 levels "A","B","C","D",..
 $ another.one: num  0.291 0.915 0.65 0.955 0.266 ...
```

1.8.4 Lists

A list is used to collect a group of objects of different sizes and types. It is simply an ordered collection of objects (that can be other variables). The outputs of many statistical functions in R are lists (e.g. linear model fitting using `lm()`), to return all relevant information in one object. So you need to know how to unpack and manipulate lists.

```
> List1 <- list(names=c("Fred","Bob"), ages=c(42, 77, 13, 91))
> List1
$names
[1] "Fred" "Bob"

$ages
[1] 42 77 13 91

> List1[[1]] # access using number of the list item instead of name
[1] "Fred" "Bob"
> List1[[2]]
[1] 42 77 13 91
> List1[["ages"]]
[1] 42 77 13 91
> List1$ages
[1] 42 77 13 91
```

You can build lists of lists too!

1.9 Variable Types, Type Conversion and Special Values

There are different kinds of data variable types such as integer, float (including real numbers), and string (e.g., text words). Beware of the difference between NA (Not Available) and NaN (Not a Number).

In the following examples, the `as.*` commands all convert a variable from one type to another:

```
> as.integer(3.1)
[1] 3
> as.numeric(4)
[1] 4
> as.roman(155)
[1] CLV
> as.character(155) # same as converting to string
[1] "155"
> as.logical(5)
[1] TRUE
> as.logical(0)
[1] FALSE
> b <- NA
> is.na(b)
[1] TRUE
> b <- 0./0.
> b
[1] NaN
> is.nan(b)
[1] TRUE
> b <- 5/0
> b
[1] Inf
> is.nan(b)
[1] FALSE
> is.infinite(b)
[1] TRUE
> is.finite(b)
[1] FALSE
> is.finite(0/0)
[1] FALSE
```

1.10 Creating and Manipulating Data structures

1.10.1 Creating Sequences

The `:` operator creates vectors of sequential integers:

```
> years <- 1990:2009
> years
[1] 1990 1991 1992 1993 1994 1995 1996 1997 1998 1999
[11] 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009

> years <- 2009:1990 # or in reverse order
> years
[1] 2009 2008 2007 2006 2005 2004 2003 2002 2001 2000
[11] 1999 1998 1997 1996 1995 1994 1993 1992 1991 1990
```

For sequences of fractional numbers, you have to use `seq()` :

```
> seq(1, 10, 0.5)
 [1]  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5  7.0  7.5  8.0
[16]  8.5  9.0  9.5 10.0
```

You can also `seq(from=1,to=10, by=0.5)` OR `seq(from=1, by=0.5, to=10)` with the same effect (try it) — this explicit, "argument matching" approach is partly why R is so popular.

1.10.2 Acessing parts of data stuctures – Indices and Indexing

Every element (entry) of a vector in R has an order: the first value, second, third, etc. To illustrate this, let's create a simple vector:

```
> MyVar <- c( 'a' , 'b' , 'c' , 'd' , 'e' )
```

Then, square brackets extract values based on their numerical position in the vector:

```
> MyVar[1] # Show element in first position
[1] "a"
> MyVar[4]
[1] "d" # Show element in fourth position
```

The values in square brackets are called "indices" — they give the index (position) of the required value. We can also select sets of values in different orders, or repeat values:

```
> MyVar[c(3,2,1)] # reverse order
[1] "c" "b" "a"
MyVar[c(1,1,5,5)] # repeat indices
[1] "a" "a" "e" "e"
```

You can also manipulate vectors by indexing:

```
> v <- c(0, 1, 2, 3, 4) # Re-create the vector variable v
> v[3] # access one element
[1] 2
> v[1:3] # access sequential elements
[1] 0 1 2
> v[-3] # remove elements
[1] 0 1 3 4
> v[c(1, 4)] # access non-sequential
[1] 0 3
```

For matrices, you need to use both row and column indices:

```
> mat1 <- matrix(1:25, 5, 5, byrow=TRUE) #create a matrix
> mat1
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
[3,]   11   12   13   14   15
[4,]   16   17   18   19   20
[5,]   21   22   23   24   25
> mat1[1,2]
```

```
[1] 2
> mat1[1,2:4]
[1] 2 3 4
> mat1[1:2,2:4]
     [,1] [,2] [,3]
[1,]    2    3    4
[2,]    7    8    9
```

1.10.3 Recycling

When vectors are of different lengths, R will recycle the shorter one to make a vector of the same length:

```
a <- c(1,5) + 2
x <- c(1,2); y <- c(5,3,9,2)
x + y
x + c(y,1)  ## somewhat strange!
```

Recycling is convenient, but dangerous!

1.10.4 Basic vector-matrix operations

```
> v2 <- v
> v2 <- v2*2 # whole-vector operation
> v2
[1] 0 2 4 6 8
> v * v2 # product element-wise
[1]  0  2  8 18 32
> t(v) # transpose the vector
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    1    2    3    4
> v %*% t(v) # matrix/vector product
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    0    0    0    0
[2,]    0    1    2    3    4
[3,]    0    2    4    6    8
[4,]    0    3    6    9   12
[5,]    0    4    8   12   16
> v3 <- 1:7 # assign using sequence
> v3
[1] 1 2 3 4 5 6 7
> v4 <- c(v2, v3) # concatenate vectors
> v4
 [1] 0 2 4 6 8 1 2 3 4 5 6 7
> q() # quit
```

1.10.5 Strings and Pasting

It is important to know how to handle strings in R two main reasons:

- To deal with text data, such at experimental treatment names
- To generate appropriate text labels and titles for figures

Let's try creating and manipulating strings:

```
> species.name <- "Quercus robur" #double quotes
> species.name
[1] "Quercus robur"
> species.name <- 'Fraxinus excelsior' #single quotes
> species.name
[1] "Fraxinus excelsior"
> paste("Quercus", "robur")
[1] "Quercus robur"
> paste("Quercus", "robur",sep = "") #Get rid of space
"Quercusrobur"
> paste("Quercus", "robur",sep = ", ") #insert comma to separate
```

As you can see above, both double and single quotes work, but I suggest that you use double quotes — this will allow you to define strings that contain a single quotes, which is often necessary.

And as is the case with so many R functions, pasting works on vectors:

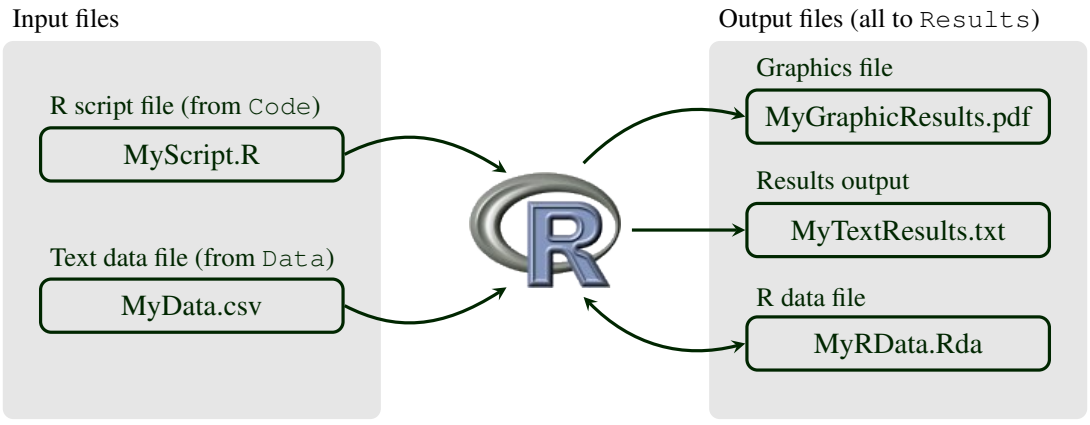
```
> paste('Year is:', 1990:2000)
 [1] "Year is: 1990" "Year is: 1991" "Year is: 1992" "Year is: 1993"
 [5] "Year is: 1994" "Year is: 1995" "Year is: 1996" "Year is: 1997"
 [9] "Year is: 1998" "Year is: 1999" "Year is: 2000"
```

Note that this last example creates a vector of 11 strings.

1.11 Your analysis workflow

In using R for an analysis, you will likely use and create several files. It is sensible to create a folder (directory) to keep all code files together. You can then set R to work from this directory, so that files are easy to find and run — this will be your "working directory" (more on this below). Also, you don't want to mix code files with data and results files. So to keep everything you will create separate directories for these as well.

Thus, your typical R analysis workflow will be:



Some details on each kind of file:

R script files These are plain text files containing all the R code needed for an analysis. These should always be created with a simple text editor like Notepad (Windows), TextEdit (MacOS) or Geany (Linux) and saved with the extension `*.R`. We will use RStudio in this class (more on this below). You should *never* use Word to save or edit these files as R can only read code from plain text files.

Text data files These are files of data in plain text format containing one or more columns of data (numbers, strings, or both). Although there are several format options, we will tyoucally be using csv files, where the entries are separated by commas. These are easy to create and export from Excel (if that's what you use...).[1]

Results output files These are a plain text files contain your results, such the the summary of output of a regression or ANOVA analysis. Typically, you will putput your results in a table format where the columns are separated by commas (csv) or tabs (tab-delimited)

Graphics files R can export graphics in a wide range of formats. This can be done automatically from R code and we will look at this later but you can also select a graphics window and click 'File ▷ Save as...'

Rdata files You can save any data loaded or created in R, including model outputs and other things, into a single`Rdata` file. These are not plain text and can only be read by R, but can hold all the data from an analysis in a single handy location. I never use these, but you can, if you want.

So let's build your workflow.

Do the following:

- ⋆ Create a directory called `MyICRModules` in an appropriate location in your `H:` drive — some place you can remember!
- ⋆ Create subdirectories within `MyICRModules` called `Code`, `Data`, and `Results`

You can create directories using `dir.create()` within R

1.11.1 The R Workspace and Working Directory

R has a "workspace" – a current working environment that includes any user-defined objects (vectors, matrices, data frames, lists, functions). At the end of an R session, the user can save an image of the current workspace that is automatically reloaded the next time R is started. Your workspace is saved in your "Working Directory", which has to be set manually.

So before we go any further, let's get sort out where your R "Working Directory" should be and how you should set it. R has a default location where it assumes your working directory is.

in Windows, it is `C:/Windows/system32` or similar.

in Macs, it is `/User/User Name` or similar.

In UNIX, it is where you are when you launch R.

To see where your current working directory is, in R, type:

```
> getwd()
```

Now, set the working directory to be `MyICRModules/Code`.

[1] If you are using a computer from elsewhere in the EU, Excel may use a comma ($\pi = 3,1416$) instead of a decimal point ($\pi = 3.1416$). In this case, *csv* files may use a semi-colon to separate columns and you can use the alternative function *read.csv2()* to read them into R.

This tells you what the current working directory ("wd") is. You can change it to the new directory you created. For example, if you created `MyICRModules` directly in your `H:\`, the you would use:

```
> setwd("H:/MyICRModules/Code")
> dir() #check what's in the current working directory
```

How would this work when you send your project to somebody else? *AVOID putting a* `setwd` *command at the start of your R script*, as setting the working directory always requires an absolute path, which will differ across computers, platforms, and users. Let the end user sort out how to set the working directory. The more important thing is that your script, internally, to import data and export results, should *not* "absolute paths". More on this below — see section on relative paths.

On your own computer, you can also change R's default to a particular working directory where you would like to start.

In Linux, you can do this by editing the `Rprofile.site` site with `sudo gedit /etc/R/Rprofile.site`. In that file, you would add your start-up parameters between the lines `.First <- function() cat("\n Welcome to R!\n\n")` and `.Last <- function() cat("\n Goodbye!\n\n")` — between thes lines, insert `setwd("/home/YourName/YourDirectoryPath")`

In Windows and Macs, you can find the `Rprofile.site` file by seraching for it. When I last checked, it used to be at `C:\Program Files\R\etc\Rprofile.site`

If you are using RStudio, you can change the default working directory by thorugh the RStudio "Options" dialog.

1.12 Importing and Exporting Data

We are now ready to see how to import and export data in R, typically the first step of your analysis. The best option is to have your data in a comma separated value text file or in a tab separated file. Then, you can use the function `read.csv` (or `read.table`) to import your data. Now, lets get some data into your `Data` directory.

- ⋆ Go to the repository you downloaded from bitbucket and unzipped, and navigate to the `Data` directory.
- ⋆ Copy the file `trees.csv` into your own `Data` directory.
- ⋆ Now, try the following:

```
> MyData <- read.csv("../Data/trees.csv")
> ls() #Check that MyData has appeared
> head(MyData) # Have a quick look at the data frame
> str(MyData) # Have a quick look at the column types
> MyData <- read.csv("../Data/trees.csv", header = TRUE) # with headers
> MyData <- read.table("../Data/trees.csv", sep = ',', header = TRUE) #another way
> head(MyData)
> MyData <- read.csv("../Data/trees.csv", skip = 5) # skip first 5 lines
```

Note that the resulting `MyData` in your workspace is a R dataframe. Also, note the UNIX-like paths using forward slashes (Windows uses back slashes).

1.12.1 Use relative paths!

The ../ in `read.csv("../Data/trees.csv")` above signifies a "relative" path. That is, you are asking R to load data that lies in a different directory (folder) relative your current location (in this case, you are in your `Code` directory). In other, more dorky words, `../Data/trees.txt` points to a file named `trees.txt` located in the "parent" of the current directory.

What is an absolute path?— one that specifies the whole path on your computer, say from `C:/` "upwards".

Using relative paths in in your R scripts and code will make your code computer independent and your life better! The relative path way should always be the way you load data in your analyses scripts — it will guarantee that your analysis works on every computer, not just your college computer.

1.12.2 Writing out to and saving files

You can also save your data frames using `write.table` or `write.csv`:

```
> write.csv(MyData, "../Results/MyData.csv")
> dir("../Results/") # Check if it worked
> write.table(MyData[1,], file = "../Results/MyData.csv",append=TRUE) # append
> write.csv(MyData, "../Results/MyData.csv", row.names=TRUE) # write row names
> write.table(MyData, "../Results/MyData.csv", col.names=FALSE) # ignore col names
```

1.13 Writing R code

Typing in commands interactively in the R console is good for starters, but you will want to switch to putting your sequence of commands into a script file, and then ask R to run those commands.

- ⋆ Open a new text file, call it `basic_io.R`, and save it to your `Code` directory.
- ⋆ Write the above input-output commands in it:

```
#A simple R script to illustrate R input-output.
# Run line by line and check inputs outputs to understand what is
# happening

MyData <- read.csv("../Data/trees.csv", header = TRUE) # import with headers

write.csv(MyData, "../Results/MyData.csv") #write it out as a new file

write.table(MyData[1,], file = "../Results/MyData.csv",append=TRUE) # Append to it

write.csv(MyData, "../Results/MyData.csv", row.names=TRUE) # write row names

write.table(MyData, "../Results/MyData.csv", col.names=FALSE) # ignore column names
```

You will get a warning in the `write.table` command above with `append = TRUE`, because you are adding the column headers every time you append to the existing file. To prvent R from appending column names, add the directive `col.names = FALSE`.

- ⋆ Place the cursor on the first line of code in the script file and run it by pressing the keyboard shortcut (PC: ctrl+R, Mac: command+enter, Linux: ctrl+enter if you are using `geany`).

⋆ Check after every line that you are getting the expected result.

1.13.1 Running R code

But even writing to a script file and running the code line-by-line or block-by-block is not your ultimate goal. What you would really like to do (believe me, you do!) is to just run your full analysis and outputs all the results. The way to run `⋆.R` script/code from the command line is to `source` it. This causes R to accept code input from a named file and run it.

⋆ Try sourcing `basic_io.R`:

```
> source("basic_io.R") # Assuming you are in Code directory!
```

⋆ If you get errors, fix them!

Note that you will need to add the directory path to the file name (`basic_io.R` in the above example), if the script file is not in your working directory. For example, you will need `source("../Code/control.R"` if your working directory is `Data`.

Also, `source` has a `chdir` argument whose default value is FALSE. When set to TRUE, it will change the working directory to the directory of the file being sourced.

1.14 Writing R Functions

R lets you write your own functions, just like any other programming language. The syntax is quite simple, with each function accepting arguments and returning a value:

```
MyFunction <- function(Arg1, Arg2){

  ## statements involving Arg1, Arg2

  return (ReturnValue)

}
```

Note the curly brackets – these are necessary for R to know where the specification of the function starts and ends.

Or you can try something more useful, for example a simple scripot to caculate the standard error of a mean, given some data (vector) `x`.

⋆ Open a code or text editor (not Mirosoft Word™!), and type the code below in a file.
⋆ Call the file `seMean.R`, and save it in your `Code` directory.

```
# get standard error of the mean from a set of values (x)
seMean <- function(x){

    x <- na.omit(x)      # get rid of missing values

    se <- sqrt(var(x)/length(x))     # calculate the standard error

    # tell the function to report the standard error
    return(se)
```

```
}
print(seMean(rnorm(1000, m = 0, sd = 1)))
```

Here `x` is the only argument you need to give this function. In the script above, you are running the function `seMean` with the argument `x` the result of running `rnorm(1000, m = 0, sd = 1)` (what 'data' does this generate?).

- ⋆ Now run `seMean.R` by sourcing it as you learnt above:

```
> source("seMean.R")
```

Now let's write another script containing a useful function:

- ⋆ Open a code or text editorand type the code below in a file.
- ⋆ Call the file `TreeHeight.R`, and save it in your `Code` directory

```
# This function calculates heights of trees
# from the angle of elevation and the distance
# from the base using the trigonometric formula
# height = distance * tan(radians)
#
# Arguments:
# degrees      The angle of elevation
# distance     The distance from base
#
# Output:
# The height of the tree, same units as "distance"

TreeHeight <- function(degrees, distance)
{
  radians <- degrees * pi / 180
  height <- distance * tan(radians)
  print(paste("Tree height is:", height))
  return (height)
}

TreeHeight(37, 40)
```

- ⋆ Run `TreeHeight.R` block by block and check what each line is doing.
- ⋆ Now run it using `source`.
- ⋆ If you get errors, fix them!

1.15 Control statements

In R, you can write `if`, `then`, `else` statements, and `for` and `while` loops like any programming language. However, loops are slow in R, so use them sparingly.

- ⋆ Type the following in a script file called `control.R` (save it in your `Code` directory)
- ⋆ Run `control.R` function block by block (not line by line!) and check what each line is doing.
- ⋆ Now run it using `source`.
- ⋆ If you get errors, fix them! (OK, I am going to stop saying this henceforth)

```
## If statement
if (a == TRUE){
```

```
  print ("a is TRUE")
}
else{
  print ("a is FALSE")
}

## On the same line
if (z <= 0.25) print ("Less than a quarter")

## Basic for loop using a sequence
for (i in 1:100){
  j <- i * i
  print(paste(i, " squared is", j ))
}

## for loop using a vector
v1 <- c("a","bc","def")
for (element in v1){
  print(element)
}

## Basic while loop
i <- 0
while (i<100){
  print(i^2)
}
```

1.16 Useful R Functions

There are a number of very useful functions available by default (in the "base packages"). Here are some particularly useful ones:

1.16.1 Mathematical

`log(x)`	Natural logarithm
`log10(x)`	Logarithm in base 10
`exp(x)`	e^x
`abs(x)`	Absolute value
`floor(x)`	Largest integer $< x$
`ceiling(x)`	Smallest integer $> x$
`pi`	π
`sqrt(x)`	$\sqrt{x}$
`sin(x)`	Sinus function

1.16.2 Strings

`strsplit(x,';')`	Split the string at ';'
`nchar(x)`	Number of characters
`toupper(x)`	Set to upper case
`tolower(x)`	Set to lower case
`paste(x1,x2,sep=';')`	Join the strings using ';'

1.16.3 Statistical

`mean(x)`	Compute mean (of a vector or matrix)
`sd(x)`	Standard deviation
`var(x)`	Variance
`median(x)`	Median
`quantile(x,0.05)`	Compute the 0.05 quantile
`range(x)`	Range of the data
`min(x)`	Minimum
`max(x)`	Maximum
`sum(x)`	Sum all elements

1.16.4 Random number distributions

`rnorm(10, m=0, sd=1)`	Draw 10 normal random numbers with mean 0 and s.d. 1
`dnorm(x, m=0, sd=1)`	Density function
`qnorm(x, m=0, sd=1)`	Cumulative density function
`runif(20, min=0, max=2)`	Twenty random numbers from uniform [0,2]
`rpois(20, lambda=10)`	Twenty random numbers from Poisson(λ)

1.17 Packages

The main strength of R is that users can easily build packages and share them through `cran.r-project.org`. There are packages to do most statistical and mathematical analysis you might conceive, so check them out before reinventing the wheel! Visit `cran.r-project.org` and go to packages to see a list and a brief description.

In UNIX, you can use the terminal to install R packages:

```
$ sudo apt-get install r-cran-ggplot2 r-cran-plyr r-cran-reshape2
```

In Windows and Macs, you can install a package within R by using the `install.packages()` command. For example, try:

```
> install.packages(c("ggplot2","plyr","reshape2"))
```

In UNIX, you will have to launch a `sudo R` session first to get this to work.

You can also use the RStudio GUI to install packages using your mouse and menu.

1.18 Practical

1. Modify the script `TreeHeight.R` so that it does the following:
   - Loads `trees.csv` and calculates tree heights for all trees in the data. Note that the distances have been measured in meters. (Hint: use relative paths))

- Creates a csv output file called `TreeHts.csv` in `Results` that contains the calculated tree heights along with the original data in the following format (only first two rows and headers shown):

```
"Species","Distance.m","Angle.degrees","Tree.Height.m"
"Populus tremula",31.6658337740228,41.2826361937914,25.462680727681
"Quercus robur",45.984992608428,44.5359166583512,46.094124200205
```

2. Imagine you wanted to make the `TreeHeight.R` script more general so that it could be used for other datasets, not just `trees.csv`.

   - Write another R script called `get_TreeHeight.R` that takes a csv file name from the command line (e.g., `get_TreeHeight.R Trees.csv`) and outputs the result to a file just like `TreeHeight.R` above, but this time includes the input file name in the output file name as `InputFileName_treeheights.csv`. Note that you will have to strip the `.csv` or whatever the extension is from the filename, and also `../` etc., if you are using relative paths.
   - Write a Unix shell script called `run_get_TreeHeight.sh` that tests `get_TreeHeight.R` — you can include `trees.csv` as your example file.
   - Command-line parameters are accessible within the R running environment via `commandArgs()` — so help(commandArgs) might be your starting point.

1.19 Readings

- Check the readings under the R directory in the bitbucket master repository
- Use the internet! Google "R tutorial", and plenty will pop up. Choose one that seems the most intuitive to you.
- The Use R! series (the yellow books) by Springer are really good. In particular, consider: "A Beginner's Guide to R", "R by Example", "Numerical Ecology With R", "ggplot2" (coming up in Chapter 2), "A Primer of Ecology with R", "Nonlinear Regression with R", "Analysis of Phylogenetics and Evolution with R".
- For more focus on dynamical models: Soetaert & Herman. 2009 "A practical guide to ecological modelling: using R as a simulation platform".
- There are excellent websites besides cran. In particular, check out `www.statmethods.net` and `http://en.wikibooks.org/wiki/R_Programming`.
- For those who are coming with Matlab experience: `http://www.math.umaine.edu/~hiebeler/comp/matlabR.html`
- Bolker, B. M.: Ecological Models and Data in R (eBook and Hardcover available).
- Beckerman, A. P. & Petchey, O. L. (2012) Getting started with R: an introduction for biologists. Oxford, Oxford University Press.
  Very basic, good if you are really stuck at the outset.

- Crawley, R. (2013) The R book. 2nd edition. Chichester, Wiley.
  Excellent but enormous reference book, code and data available from `www.bio.ic.ac.uk/research/mjcraw/therbook/index.htm`

Chapter 2

Plotting and graphics in R

R can produce beautiful graphics, without the time-consuming and fiddly methods that you might have used in Excel or equivalent. You should also make it a habit to quickly plot the data for exploratory analysis. So we are going to learn some basic plotting first. Then, you will learn how the package `ggplot2`, can allow the rapid creation of truly elegant publication-grade graphics.

By the time you have worked through this chapter you should be able to:

- Use R to quickly import data and inspect it
- Do basic, exploratory scatter, bar, histogram, etc plots to examine subsets of your data.
- Create truly elegant, publication-grade graphics using the "grammar of graphics", with `ggplot2`.

2.1 Basic plotting and graphical data exploration

In many cases you just want to quickly plot the data for exploratory analysis of your data.

2.1.1 Basic plotting commands

Here is a menu of basic R plotting commands (use `?commandname to learn more about it`):

`plot(x,y)`	Scatterplot
`plot(y~x)`	Scatterplot with `y` as a response variable
`hist(mydata)`	Histogram
`barplot(mydata)`	Bar plot
`points(y1~x1)`	Add another series of points
`boxplot(y~x)`	Boxplot

2.1.2 R graphics devices

In all that follows, you may often end up plotting multiple plots on the same graphics window without intending to do so, because R by default keeps plotting in the most recent plotting window that was opened. You can close a particular graphics window or "device" by using `dev.off()`,

and all open devices/windows with `graphics.off()`. By default, `dev.off()` will close the most recent figure device that was opened. Note that there are invisible devices as well! That is if you are printing to pdf (coming up below), the device or graphics window will not be visible on your computer screen.

So let's try some simple plotting for data exploration. As a case study, we will use a dataset on Consumer-Resource (e.g., Predator-Prey) body mass ratios taken from the Ecological Archives of the ESA (Barnes *et al.* 2008, Ecology 89:881).

- ⋆ Go to the repository you downloaded from bitbucket and unzipped, and navigate to the `Data` directory.
- ⋆ Copy the file `EcolArchives-E089-51-D1.csv` into your own `Data` directory.
- ⋆ Now, do the following:

```
> MyDF <- read.csv("../Data/EcolArchives-E089-51-D1.csv")
> dim(MyDF) #check the size of the data frame you loaded
[1] 34931    15
```

Let's look at what the data contain (type `MyDF$` and hit the TAB key twice):

```
> MyDF$
MyDF$Record.number                MyDF$Predator.mass
MyDF$In.refID                     MyDF$Prey
MyDF$IndividualID                 MyDF$Prey.common.name
MyDF$Predator                     MyDF$Prey.taxon
MyDF$Predator.common.name         MyDF$Prey.mass
MyDF$Predator.taxon               MyDF$Prey.mass.unit
MyDF$Predator.lifestage           MyDF$Location
MyDF$Type.of.feeding.interaction
```

You can also use the `str()` and `head()` commands that you learned about previously.

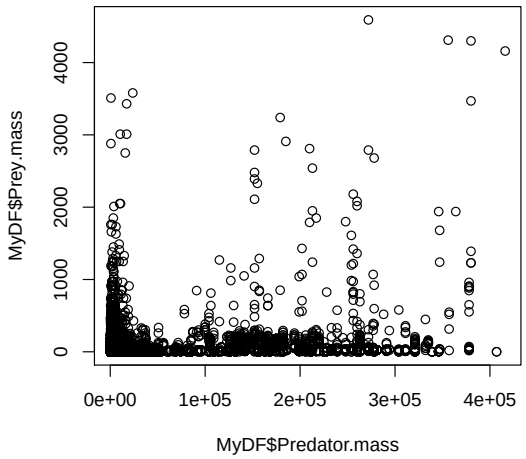
As you can see, these data contain predator-prey body size information. This is an interesting dataset because it is huge, and covers a wide range of body sizes of aquatic species involved in consumer-resource interactions — from unicells to whales. Analyzing this dataset should tell us a lot about what sizes of prey predators like to eat.

2.1.3 Scatter Plot

So let's start by plotting Predator mass vs. Prey mass:

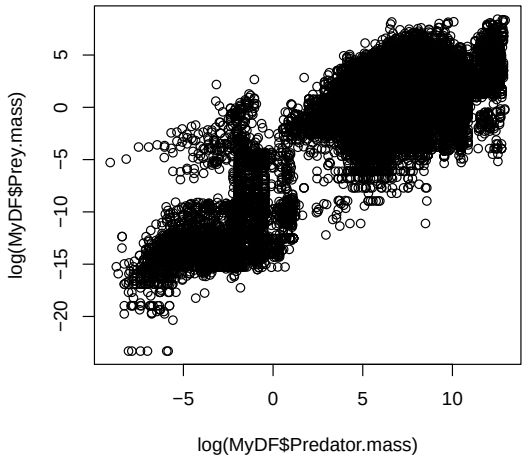
```
> plot(MyDF$Predator.mass,MyDF$Prey.mass)
```

Figure 2.1: A consumer-resource (predator-prey) interaction waiting to happen.



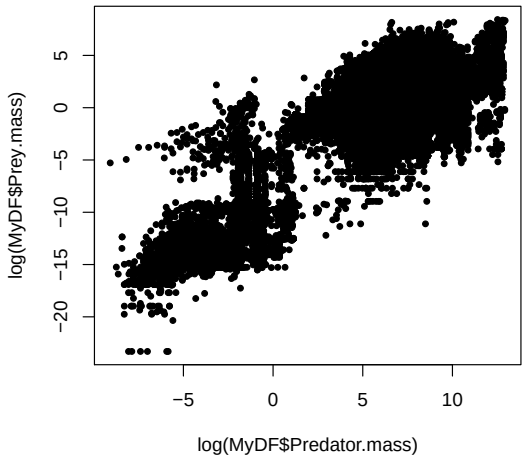
That doesn't look very nice! Let's try taking logarithms (why?).

```
> plot(log(MyDF$Predator.mass),log(MyDF$Prey.mass))
```

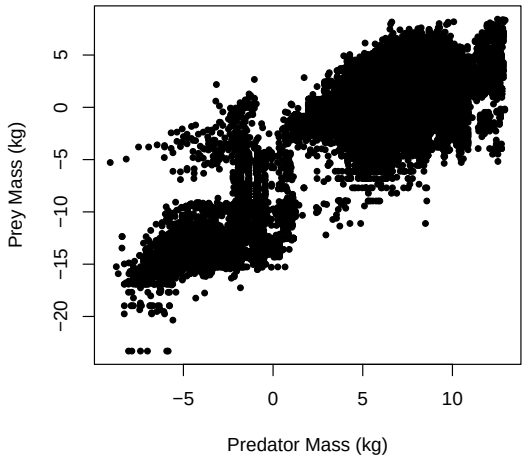


We can change almost any aspect of the resulting graph; let's change the symbols by specifying the plot characters using pch:

```
> plot(log(MyDF$Predator.mass),log(MyDF$Prey.mass),pch=20) # Change marker
```




```
> plot(log(MyDF$Predator.mass),log(MyDF$Prey.mass),pch=20,
    xlab = "Predator Mass (kg)", ylab = "Prey Mass (kg)") # Add labels
```

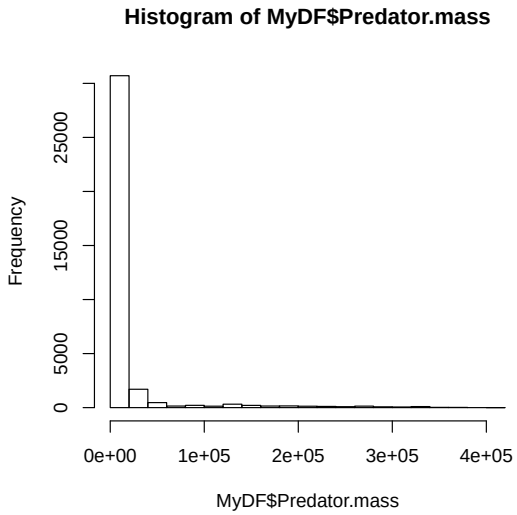


2.1.4 Histograms

Why did we have to take a logarithm to see the relationship between predator and prey size? Plotting histograms of the two classes (predator, prey) should be insightful, as we can then see the "marginal" distributions of the two variables.

Let's first plot a histogram of predator body masses:

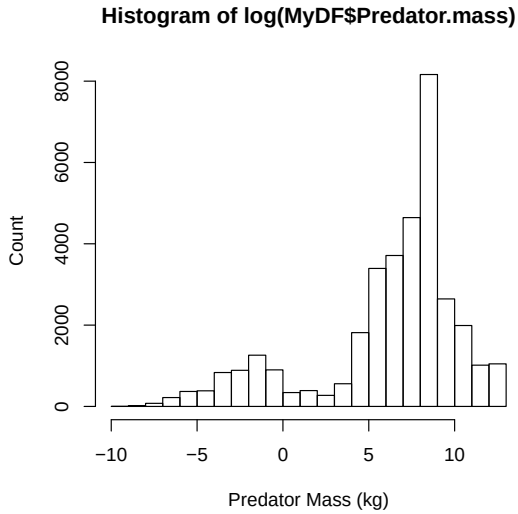
```
> hist(MyDF$Predator.mass)
```



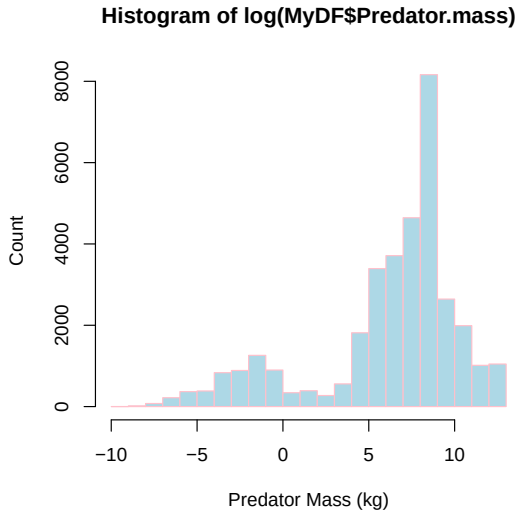
Clearly, the data are heavily skewed, with small body sized organisms dominating (that's a universal pattern on planet earth). Let's now take a logarithm and see if we can get a better idea of what the distribution of predator sizes looks like:

```
> hist(log(MyDF$Predator.mass),
```

```
    xlab = "Predator Mass (kg)", ylab = "Count") # include labels
```




```
> hist(log(MyDF$Predator.mass),xlab="Predator Mass (kg)",ylab="Count",
    col = "lightblue", border = "pink") # Change bar and borders colors
```



So, taking a log really makes clearer what the distribution of body predator sizes looks like. *Try the same with prey body masses.*

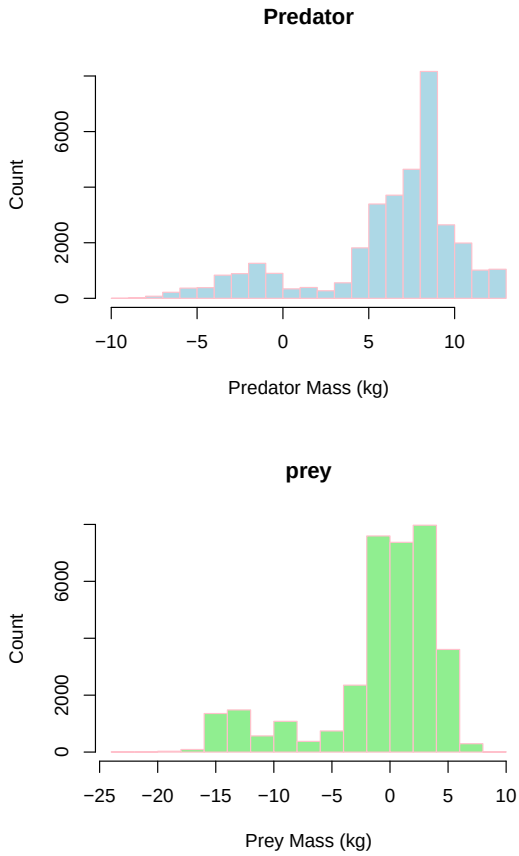
2.1.5 Exercise

We can do a lot of beautification and fine-tuning of your R plots! As an exercise, try adjusting the histogram bin widths to make them same for the predator and prey, and making the x and y labels larger and in boldface. To get started, look at the help documentation of `hist`.

2.1.6 Subplots

We can also plot both predator and prey body masses in different sub-plots using `par` so that we can compare them visually.

```
> par(mfcol=c(2,1)) #initialize multi-paneled plot
> par(mfg = c(1,1)) # specify which sub-plot to use first
> hist(log(MyDF$Predator.mass),
    xlab = "Predator Mass (kg)", ylab = "Count",
    col = "lightblue", border = "pink",
    main = 'Predator') # Add title
> par(mfg = c(2,1)) # Second sub-plot
> hist(log(MyDF$Prey.mass),
    xlab="Prey Mass (kg)",ylab="Count",
    col = "lightgreen", border = "pink",
    main = 'prey')
```



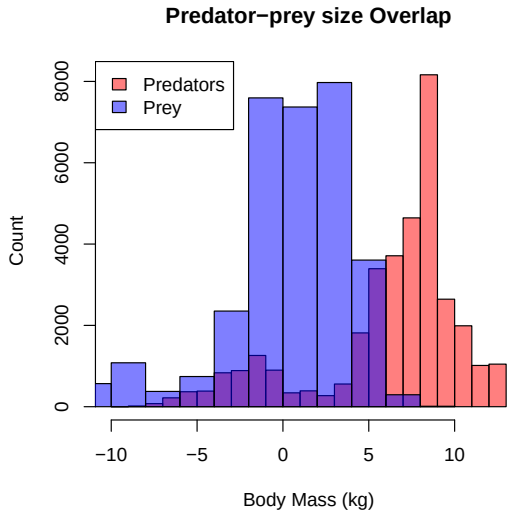
Another option for making multi-panel plots is the `layout` function.

2.1.7 Overlaying plots

Better still, we would like to see if the predator mass and prey mass distributions are similar by overlaying them.

```
> hist(log(MyDF$Predator.mass), # Predator histogram
```

```
    xlab="Body Mass (kg)", ylab="Count",
    col = rgb(1, 0, 0, 0.5), # Note 'rgb', fourth value is transparency
    main = "Predator-prey size Overlap")
> hist(log(MyDF$Prey.mass), col = rgb(0, 0, 1, 0.5), add = T) # Plot prey
> legend('topleft',c('Predators','Prey'),   # Add legend
    fill=c(rgb(1, 0, 0, 0.5), rgb(0, 0, 1, 0.5))) # Define legend colors
```

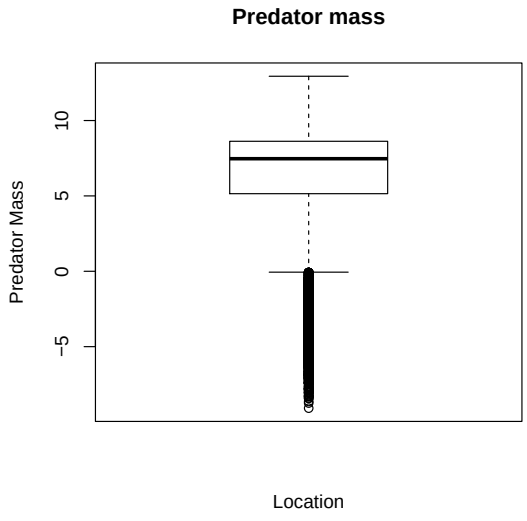


It would be nicer to have both the plots with the same bin sizes – try to do it

2.1.8 Boxplots

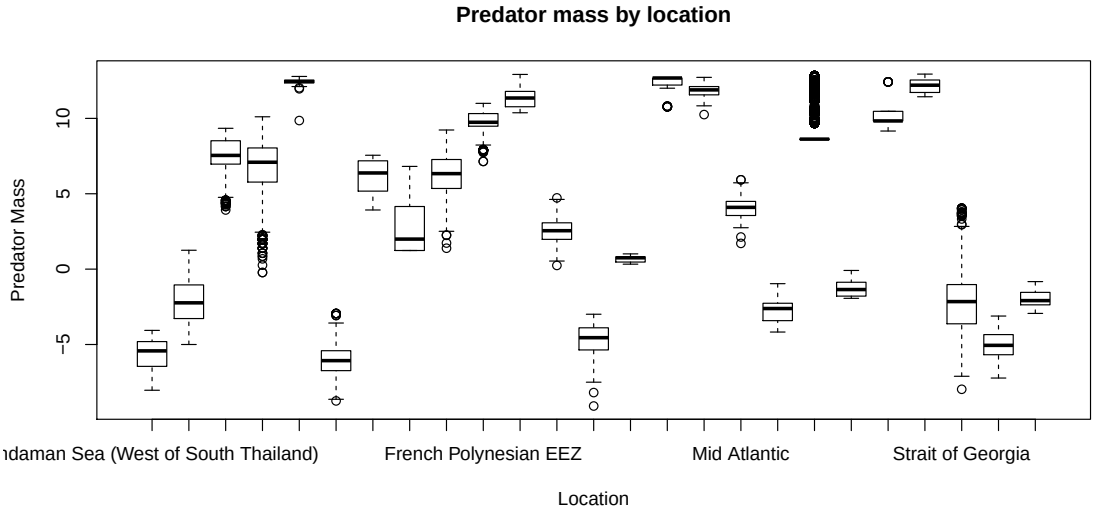
Now, let's try plotting boxplots instead of histograms. These are useful for getting a visual summary of the distribution of your data.

```
> boxplot(log(MyDF$Predator.mass),
    xlab = "Location", ylab = "Predator Mass",
    main = "Predator mass")
```



Now let's see how many locations the data are from:

```
> boxplot(log(MyDF$Predator.mass) ~ MyDF$Location, # Why the tilde?
    xlab = "Location", ylab = "Predator Mass",
    main = "Predator mass by location")
```



Note the tilde (~). This is to tell R to subdivide or categorize your analysis and plot by the "Factor" location. More on this later.

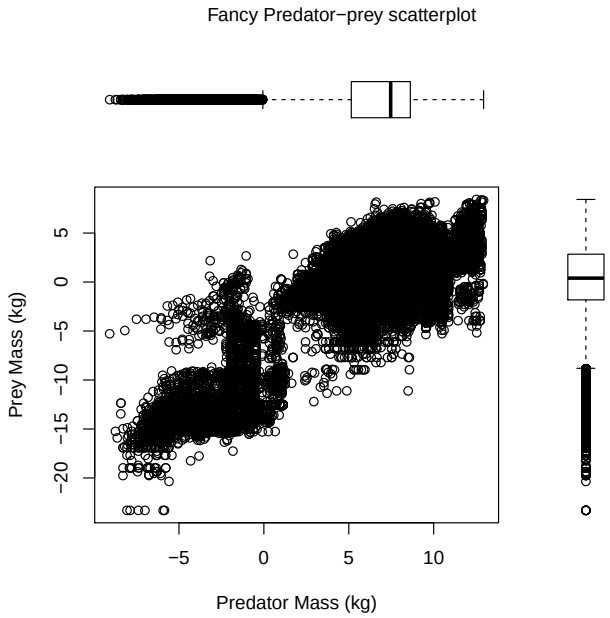
That's a lot of locations! You will need an appropriately wide plot to see all the boxplots adequately. Now let's try boxplots by feeding interaction type:

```
> boxplot(log(MyDF$Predator.mass) ~ MyDF$Type.of.feeding.interaction,
    xlab = "Location", ylab = "Predator Mass",
    main = "Predator mass by feeding interaction type")
```

2.1.9 Combining plot types

It would be nice to see both the predator and prey marginal distributions as well as the scatterplot for an exploratory analysis. We can do this by adding boxplots of the marginal variables to the scatterplot.

```
> par(fig=c(0,0.8,0,0.8), new=TRUE) # specify figure size as proportion
> plot(log(MyDF$Predator.mass),log(MyDF$Prey.mass),
    xlab = "Predator Mass (kg)", ylab = "Prey Mass (kg)") # Add labels
> par(fig=c(0,0.8,0.55,1), new=TRUE)
> boxplot(log(MyDF$Predator.mass), horizontal=TRUE, axes=FALSE)
par(fig=c(0.65,1,0,0.8),new=TRUE)
> boxplot(log(MyDF$Prey.mass), axes=FALSE)
> mtext("Fancy Predator-prey scatterplot", side=3, outer=TRUE, line=-3)
```



To understand this plotting method, think of the full graph area as going from (0,0) in the lower left corner to (1,1) in the upper right corner. The format of the `fig=` parameter is a numerical vector of the form `c(x1, x2, y1, y2)`. The first `fig=` sets up the scatterplot going from 0 to 0.8 on the x axis and 0 to 0.8 on the y axis. The top boxplot goes from 0 to 0.8 on the x axis and 0.55 to 1 on the y axis. The right hand boxplot goes from 0.65 to 1 on the x axis and 0 to 0.8 on the y axis. You can experiment with these proportions to change the spacings between plots.

2.1.10 Lattice plots

You can also make lattice graphs to avoid the somewhat laborious `par()` approach above of getting multi-panel plots. For this, you will need to "load" a "library" that isnl't included by default when you run R:

```
> library(lattice)
```

A lattice plot of the above data for predator mass could look like Fig. 2.1.10 (as a density plot). This was generated using (and printing to a pdf with particular dimensions):

```
> densityplot(~log(Predator.mass) | Type.of.feeding.interaction,
data=MyDF)
```

Look up `http://www.statmethods.net/advgraphs/trellis.html` and the `lattice` package help.

2.1.11 Saving your graphics

And you can also save the figure in a vector graphics format like a pdf. It is important to learn to do this, because you want to be able to save your plots in good resolution, and want to avoid the

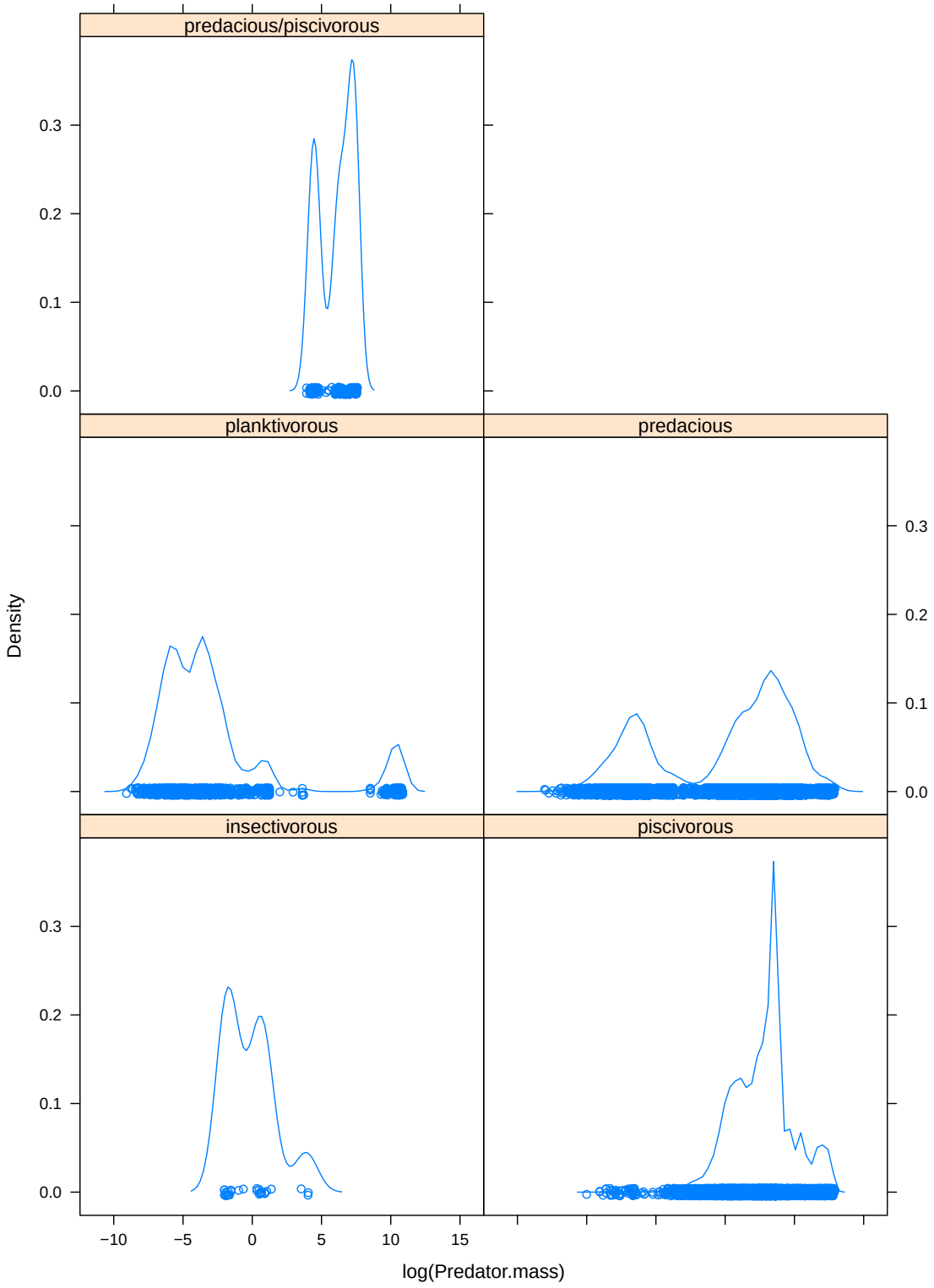


Figure 2.2: A `lattice` representation of the predator size data

manual steps of clicking on the figure, doing "save as", etc. So let's save the figure as a PDF:

```
> pdf("../Results/Pred_Prey_Overlay.pdf", # Open blank pdf page
    11.7, 8.3) # These numbers are page dimensions in inches
> hist(log(MyDF$Predator.mass), # Plot predator histogram (note 'rgb')
    xlab="Body Mass (kg)", ylab="Count",
    col = rgb(1, 0, 0, 0.5),
    main = "Predator-Prey Size Overlap")
> hist(log(MyDF$Prey.mass), # Plot prey weights
    col = rgb(0, 0, 1, 0.5),
    add = T)  # Add to same plot = TRUE
> legend('topleft',c('Predators','Prey'), # Add legend
    fill=c(rgb(1, 0, 0, 0.5), rgb(0, 0, 1, 0.5)))
> dev.off()
```

Note that you are saving to the `Results` directory now. This should always be your workflow: store and retrieve data from a `Data` directory, keep your code and work from a `Code` directory, and save outputs to a `Results` directory.

You can also try other graphic output formats. For example, `png()` (a raster format) instead of `pdf()`. As always, look at the help documentation of each of these commands!

2.2 Practical

In this practical, you will write script that draws and saves three lattice graphs by feeding interaction type: one of predator mass , one of prey mass and one of the size ratio of prey mass over predator mass. Note that you would want to use logarithms of masses (or mass-ratios) for all three plots. In addition, the script will calculate the mean and median predator mass, prey mass and predator-prey size-ratios to a csv file. The workflow would be:

- ⋆ Write a script file called `PP_Lattice.R` and save it in the `Code` directory — sourcing or running this script should result in three files called `Pred_Lattice.pdf`, `Prey_Lattice.pdf`, and `SizeRatio_Lattice.pdf` being saved in the `Results` directory (the names are self-explanatory, I hope).
- ⋆ In addition, the script should calculate the mean and median log predator mass, prey mass, and predator-prey size ratio, *by feeding type*, and save it as a single csv output table called `PP_Results.csv` to the `Results` directory. The table should have appropriate headers (e.g., Feeding type, mean, median). (Hint: you will have to initialize a new dataframe in the script to first store the calculations)
- ⋆ The script should be self-sufficient and not need any external inputs — it should import the above predator-prey dataset from the appropriate directory, and save the graphic plots to the appropriate directory (Hint: use relative paths!).
- ⋆ There are multiple ways to do this practical. The plotting and saving component is simple enough. For calculating the statistics by feeding type, you can either use the "loopy" way — first obtaining a list of feeding types (look up the `unique` or `levels` functions) and then loop over them, using `subset` to extract the dataset by feeding type at each iteration, or the the "smart", R-savvy way, by using `tapply` and avoiding looping altogether (you used this in the stats week).

2.3 Publication-quality figures in R

R can produce beautiful graphics, but it takes a lot of work to obtain the desired result. This is because the starting point is pretty much a "bare" plot, and adding features commonly required for publication-grade figures (legends, statistics, regressions, etc.) can be quite involved.

Moreover, it is very difficult to switch from one representation of the data to another (i.e., from boxplots to scatterplots), or to plot several datasets together. The R package ggplot2 overcomes these issues, and produces truly high-quality, publication-ready graphics suitable for papers, theses and reports.

One thing to note though is that currently, ggplot2 cannot be used to create 3D graphs or mosaic plots. In any case, most of you won't be needing 3D plots! If you do, there are many ways to do 3D plots using other plotting packages in R. In particular, look up the scatterplot3d and plot3D packages.

ggplot2 differs from other approaches as it attempts to provide a "grammar" for graphics in which each layer is the equivalent of a verb, subject etc. and a plot is the equivalent of a sentence. All graphs start with a layer showing the data, other layers and commands are added to modify the plot. Specifically, according to this grammar, a statistical graphic is a "mapping" from data to aesthetic attributes (colour, shape, size; set using aes) of geometric objects (points, lines, bars; set using geom).

For more on the ideas underlying ggplot, see the book "ggplot2: Elegant Graphics for Data Analysis", by H. Wickham (in your Reading directory). Also, the website ggplot2.org a great resource.

ggplot2 should already be available on the college computers. If you are using your own computer, look up the section on installing packages in the previous chapter.

ggplot can be used in two ways: with qplot (for quick plotting) and ggplot for full-blown, customizable plotting.

Finally, note that ggplot2 only accepts data in data frames.

2.3.1 Basic plotting with qplot

qplot can be used to quickly produce graphics for exploratory data analysis, and as a base for more complex graphics. It uses syntax that is closer to the standard R plotting commands.

We will use the same predator-prey body size dataset again – you will soon see how much nice the same types of plots you made above look when done with ggplot!.

Scatterplots

Let's start plotting the Predator.mass vs Prey.mass:

```
> require(ggplot2)  ## Load the package
Loading required package: ggplot2
> qplot(Prey.mass, Predator.mass, data = MyDF)
```

As before, let's take logarithms and plot:

```
> qplot(log(Prey.mass), log(Predator.mass), data = MyDF)
```

Now, color the points according to the type of feeding interaction:

```
> qplot(log(Prey.mass), log(Predator.mass), data = MyDF,
    colour = Type.of.feeding.interaction)
```

The same as above, but changing the shape:

```
> qplot(log(Prey.mass), log(Predator.mass), data = MyDF,
    shape = Type.of.feeding.interaction)
```

Aesthetic mappings

These examples demonstrate a key difference between `qplot` and the standard `plot` command: When you want to assign colours, sizes or shapes to the points on your plot, using the `plot` command, itâĂŹs your responsibility to convert (i.e., "map") a categorical variable in your data (e.g., type of feeding interaction in the above case) onto colors (or shapes) that `plot` knows how to use (e.g., by specifying "red", "blue", "green", etc). ggplot does this mapping for you automatically, and also provides a legend! This makes it really easy to quickly include additional data (e.g., if a new feeding interaction type was added to the data) on the plot.

Instead of using ggplot's automatic mapping, if you want to manually set a color or a shape, you have to use `I()` (meaning "Identity"). To see this in practise, try the following:

```
> qplot(log(Prey.mass), log(Predator.mass),
    data = MyDF, colour = "red")
```

You chose red, but ggplot used mapping to convert it to a particular shade of red. To set it manually to the real red, do this:

```
> qplot(log(Prey.mass), log(Predator.mass),
    data = MyDF, colour = I("red"))
```

Similarly, for point size, compare these two:

```
> qplot(log(Prey.mass), log(Predator.mass),
    data = MyDF, size = 3) #with ggplot size mapping
> qplot(log(Prey.mass), log(Predator.mass),
    data = MyDF, size = I(3)) #no mapping
```

But for shape, ggplot doesn't have a continuous mapping because shapes are a discrete variable. To see this, compare these two:

```
> qplot(log(Prey.mass), log(Predator.mass),
    data = MyDF, shape = 3) #will give error
> qplot(log(Prey.mass), log(Predator.mass),
    data = MyDF, shape= I(3))
```

Setting transparency

Because there are so many points, we can make them semi-transparent using `alpha` so that the overlaps can be seen:

```
> qplot(log(Prey.mass), log(Predator.mass), data = MyDF,
    colour = Type.of.feeding.interaction, alpha = I(.5))
```

Here try using `alpha = .5` instead of `alpha = I(.5)` and see what happens.

Adding smoothers and regression lines

Now add a smoother to the points:

```
> qplot(log(Prey.mass), log(Predator.mass), data = MyDF,
    geom = c("point", "smooth"))
```

If we want to have a linear regression, we need to specify the method as being `lm`:

```
> qplot(log(Prey.mass), log(Predator.mass), data = MyDF,
    geom = c("point", "smooth"), method = "lm")
```

We can add a smoother for each type of interaction:

```
> qplot(log(Prey.mass), log(Predator.mass), data = MyDF,
    geom = c("point", "smooth"), method = "lm",
    colour = Type.of.feeding.interaction)
```

To extend the lines to the full range, use `fullrange = TRUE`:

```
> qplot(log(Prey.mass), log(Predator.mass), data = MyDF,
    geom = c("point", "smooth"), method = "lm",
    colour = Type.of.feeding.interaction,
    fullrange = TRUE)
```

Now we want to see how the ratio between prey and predator mass changes according to the type of interaction:

```
> qplot(Type.of.feeding.interaction,
       log(Prey.mass/Predator.mass), data = MyDF)
```

Because there are so many points, we can "jitter" them to get a better idea of the spread:

```
> qplot(Type.of.feeding.interaction,
       log(Prey.mass/Predator.mass), data = MyDF,
       geom = "jitter")
```

Boxplots

Or we can draw a boxplot of the data (note the `geom` argument, which stands for `geom`etry):

```
> qplot(Type.of.feeding.interaction,
        log(Prey.mass/Predator.mass), data = MyDF,
        geom = "boxplot")
```

Histograms and density plots

Now let's draw an histogram of predator-prey mass ratios:

```
> qplot(log(Prey.mass/Predator.mass), data = MyDF,
    geom =  "histogram")
```

Color the histogram according to the interaction type:

```
> qplot(log(Prey.mass/Predator.mass), data = MyDF,
    geom =  "histogram",
    fill = Type.of.feeding.interaction)
```

You may want to define binwidth (in units of x axis):

```
> qplot(log(Prey.mass/Predator.mass), data = MyDF,
    geom =  "histogram",
    fill = Type.of.feeding.interaction,
    binwidth = 1)
```

To make it easier to read, we can plot the smoothed density of the data:

```
> qplot(log(Prey.mass/Predator.mass), data = MyDF,
    geom =  "density", fill = Type.of.feeding.interaction)
```

And you can make the densities transparent so that the overlaps are visible:

```
> qplot(log(Prey.mass/Predator.mass), data = MyDF,
    geom =  "density", fill = Type.of.feeding.interaction, alpha =
    I(0.5))
```

Or using `colour` instead of `fill` draws only the edge of the curve:

```
> qplot(log(Prey.mass/Predator.mass), data = MyDF,
    geom =  "density", colour = Type.of.feeding.interaction)
```

Similarly, `geom = "bar"` produces a barplot, `geom = "line"` a series of points joined by a line, etc.

Multi-faceted plots

An alternative way of displaying data belonging to different classes is using "faceting". We did this using `lattice()` previously, but ggplot does a much nicer job:

```
> qplot(log(Prey.mass/Predator.mass),
    facets = Type.of.feeding.interaction ~.,
    data = MyDF, geom =  "density")
```

The `~.` (the space is not important) notation tells ggplot whether to do the faceting by row or by column. So if you want a by-column configuration, switch ~ and `.`, and also swap the position of the `.~`:

```
> qplot(log(Prey.mass/Predator.mass),
    facets =  .~ Type.of.feeding.interaction,
    data = MyDF, geom =  "density")
```

You can also facet by a combination of categories (this is going to be a big plot!):

```
> qplot(log(Prey.mass/Predator.mass),
    facets = .~ Type.of.feeding.interaction + Location,
    data = MyDF, geom =  "density")
```

And you can also change the order of the combination:

```
> qplot(log(Prey.mass/Predator.mass),
    facets = .~ Location + Type.of.feeding.interaction,
    data = MyDF, geom =  "density")
```

Logarithmic axes

A more elegant way of drawing logarithmic quantities is to set the axes to be logarithmic:

```
> qplot(Prey.mass, Predator.mass, data = MyDF, log="xy")
```

Plot annotations

Let's add a title and labels:

```
> qplot(Prey.mass, Predator.mass, data = MyDF, log="xy",
    main = "Relation between predator and prey mass",
    xlab = "log(Prey mass) (g)",
    ylab = "log(Predator mass) (g)")
```

Adding `+ theme_bw()` makes it suitable for black and white printing.

```
> qplot(Prey.mass, Predator.mass, data = MyDF, log="xy",
```

```
    main = "Relation between predator and prey mass",
    xlab = "Prey mass (g)",
    ylab = "Predator mass (g)") + theme_bw()
```

Saving your plots

Finally, let's save a pdf file of the figure (same approach as we used before):

```
> pdf("../Results/MyFirst-ggplot2-Figure.pdf")
> print(qplot(Prey.mass, Predator.mass, data = MyDF,log="xy",
    main = "Relation between predator and prey mass",
    xlab = "log(Prey mass) (g)",
    ylab = "log(Predator mass) (g)") + theme_bw())
> dev.off()
```

Using `print` ensures that the whole command is kept together and that you can use the command in a script.

2.3.2 Some more important ggplot options

Other important options to keep in mind:

`xlim`	limits for x axis: `xlim = c(0,12)`
`ylim`	limits for y axis
`log`	log transform variable `log = "x"`, `log = "y"`, `log = "xy"`
`main`	title of the plot `main = "My Graph"`
`xlab`	x-axis label
`ylab`	y-axis label
`asp`	aspect ratio `asp = 2`, `asp = 0.5`
`margins`	whether or not margins will be displayed

2.3.3 Various `geom`

`geom` Specifies the geometric objects that define the graph type. The geom option is expressed as a character vector with one or more entries. geom values include "point", "smooth", "boxplot", "line", "histogram", "density", "bar", and "jitter". Try the following:

```
# load the package
require(ggplot2)

# load the data
MyDF <- as.data.frame(
  read.csv("../Data/EcolArchives-E089-51-D1.csv"))

# barplot
qplot(Predator.lifestage,
      data = MyDF, geom = "bar")

# boxplot
qplot(Predator.lifestage, log(Prey.mass),
      data = MyDF, geom = "boxplot")

# density
```

```
qplot(log(Predator.mass),
      data = MyDF, geom = "density")

# histogram
qplot(log(Predator.mass),
      data = MyDF, geom = "histogram")

# scatterplot
qplot(log(Predator.mass), log(Prey.mass),
      data = MyDF, geom = "point")

# smooth
qplot(log(Predator.mass), log(Prey.mass),
      data = MyDF, geom = "smooth")

qplot(log(Predator.mass), log(Prey.mass),
      data = MyDF, geom = "smooth", method = "lm")
```

2.3.4 Advanced plotting: `ggplot`

The command `qplot` allows you to use only a single dataset and a single set of "aesthetics" (x, y, etc.). To make full use of `ggplot2`, we need to use the command `ggplot`, which allows you to use "layering". Layering is the mechanism by which additional data elements are added to a plot. Each layer can come from a different dataset and have a different aesthetic mapping, allowing us to create plots that could not be generated using `qplot()`, which permits only a single dataset and a single set of aesthetic mappings.

For a `ggplot` plotting command, we need at least:

- The data to be plotted, in a data frame;
- Aesthetics mappings, specifying which variables we want to plot, and how;
- The `geom`, defining the geometry for representing the data;
- (Optionally) some `stat` that transforms the data or performs statistics using the data.

To start a graph, we nust specify the data and the aesthetics:

```
> p <- ggplot(MyDF, aes(x = log(Predator.mass),
                        y = log(Prey.mass),
                        colour = Type.of.feeding.interaction ))
```

Here we have created a graphics object `p` to which we can add layers and other plot elements.

Now try to plot the graph:

```
> p
Error: No layers in plot
```

That is because we are yet to specify a geometry — only then can we see the graph:

```
> p + geom_point()
```

We can use the "plus" sign to concatenate different commands:

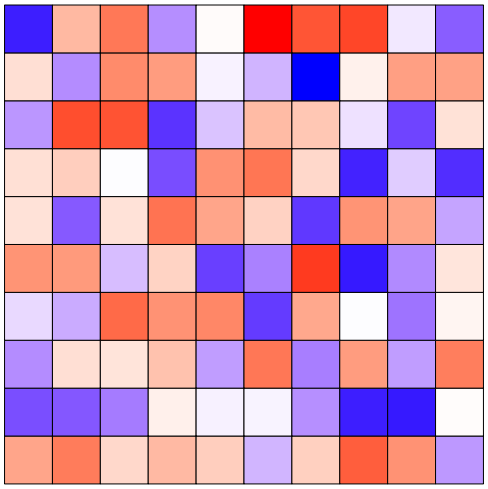


Figure 2.3: Random matrix with values sampled from uniform distribution.

```
> p <- ggplot(MyDF, aes(x = log(Predator.mass),
                        y = log(Prey.mass),
                        colour = Type.of.feeding.interaction ))
> q <- p + geom_point(size=I(2), shape=I(10)) + theme_bw()
> q
```

Let's remove the legend:

```
> q + theme(legend.position = "none")
```

We will not look at some case studies to see some useful ways in which you can use ggplot.

2.3.5 Case study 1: plotting a matrix

Here we will plot a matrix of random values taken from a normal distribution $\mathscr{U}[0,1]$. Our goal is to produce the plot in Figure 2.3. Because we want to plot a matrix, and `ggplot2` accepts only dataframes, we use the package `reshape2` that can "melt" a matrix into a dataframe:

```
require(ggplot2)
require(reshape2)

GenerateMatrix <- function(N){
    M <- matrix(runif(N * N), N, N)
    return(M)
}

> M <- GenerateMatrix(10)

> M[1:3, 1:3]
           [,1]      [,2]      [,3]
[1,] 0.2700254 0.8686728 0.7365857
[2,] 0.1744879 0.8488169 0.4165879
[3,] 0.3980783 0.7727821 0.4271121
```

```
> Melt <- melt(M)

> Melt[1:4,]
  Var1 Var2     value
1    1    1 0.0698925
2    2    1 0.6333296
3    3    1 0.8990120
4    4    1 0.8425578

> ggplot(Melt, aes(Var1, Var2, fill = value)) + geom_tile()

# adding a black line dividing cells
> p <- ggplot(Melt, aes(Var1, Var2, fill = value))
> p <- p + geom_tile(colour = "black")

# removing the legend
> q <- p + theme(legend.position = "none")

# removing all the rest
> q <- p + theme(legend.position = "none",
     panel.background = element_blank(),
     axis.ticks = element_blank(),
     panel.grid.major = element_blank(),
     panel.grid.minor = element_blank(),
     axis.text.x = element_blank(),
     axis.title.x = element_blank(),
     axis.text.y = element_blank(),
     axis.title.y = element_blank())

# exploring the colors
> q + scale_fill_continuous(low = "yellow",
                            high = "darkgreen")
> q + scale_fill_gradient2()
> q + scale_fill_gradientn(colours = grey.colors(10))
> q + scale_fill_gradientn(colours = rainbow(10))
> q + scale_fill_gradientn(colours =
                c("red", "white", "blue"))
```

2.3.6 Case study 2: plotting two dataframes

According to Girko's circular law, the eigenvalues of a matrix M of size $N \times N$ are approximately contained in a circle in the complex plane with radius $\sqrt{N}$. We are going to draw a simulation displaying this result (Figure 2.4).

```
require(ggplot2)

# function that returns an ellipse
build_ellipse <- function(hradius, vradius){
  npoints = 250
  a <- seq(0, 2 * pi, length = npoints + 1)
  x <- hradius * cos(a)
  y <- vradius * sin(a)
  return(data.frame(x = x, y = y))
}

# Size of the matrix
N <- 250
# Build the matrix
M <- matrix(rnorm(N * N), N, N)
# Find the eigenvalues
eigvals <- eigen(M)$values
# Build a dataframe
eigDF <- data.frame("Real" = Re(eigvals),
                    "Imaginary" = Im(eigvals))
```

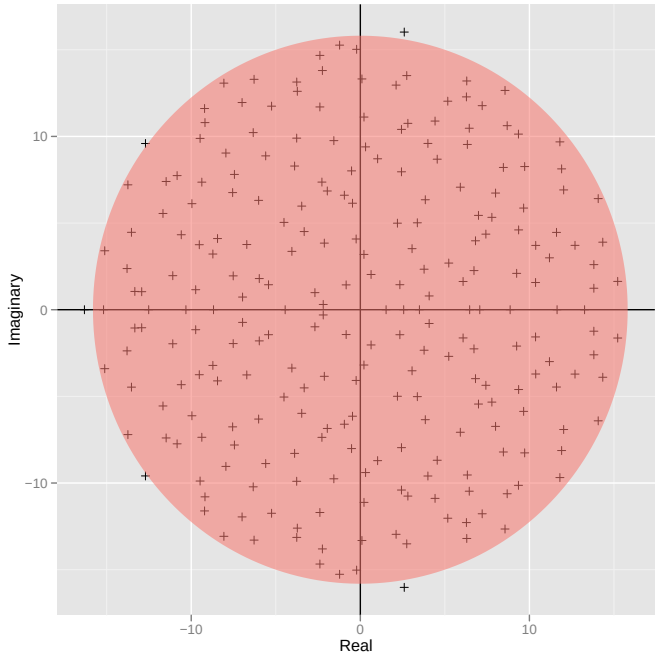


Figure 2.4: Girko's circular law.

```
# The radius of the circle is sqrt(N)
my_radius <- sqrt(N)
# Ellipse dataframe
ellDF <- build_ellipse(my_radius, my_radius)
# rename the columns
names(ellDF) <- c("Real", "Imaginary")

# Now the plotting:
# plot the eigenvalues
p <- ggplot(eigDF, aes(x = Real, y = Imaginary))
p <- p +
  geom_point(shape = I(3)) +
  opts(legend.position = "none")


# now add the vertical and horizontal line
p <- p + geom_hline(aes(intercept = 0))
p <- p + geom_vline(aes(intercept = 0))

# finally, add the ellipse
p <- p + geom_polygon(data = ellDF,
                      aes(x = Real,
                          y = Imaginary,
                          alpha = 1/20,
                          fill = "red"))
pdf("Girko.pdf")
print(p)
dev.off()
```

2.3.7 Case study 3: annotating plots

In the plot in Figure 2.5, we use the geometry "text" to annotate the plot.

```
require(ggplot2)
```

```
filename <- "Results.txt"
a <- read.table(filename, header = TRUE)
# here's how the data looks like
print(a[1:3,])
print(a[90:95,])

# append a col of zeros
a$ymin <- rep(0, dim(a)[1])

# print the first linerange
p <- ggplot(a)
p <- p + geom_linerange(data = a, aes(
                          x = x,
                          ymin = ymin,
                          ymax = y1,
                          size = (0.5)
                          ),
                        colour = "#E69F00",
                        alpha = 1/2, show_guide = FALSE)

# print the second linerange
p <- p + geom_linerange(data = a, aes(
                          x = x,
                          ymin = ymin,
                          ymax = y2,
                          size = (0.5)
                          ),
                        colour = "#56B4E9",
                        alpha = 1/2, show_guide = FALSE)

# print the third linerange
p <- p + geom_linerange(data = a, aes(
                          x = x,
                          ymin = ymin,
                          ymax = y3,
                          size = (0.5)
                          ),
                        colour = "#D55E00",
                        alpha = 1/2, show_guide = FALSE)

# annotate the plot with labels
p <- p + geom_text(data = a,
                   aes(x = x, y = -500, label = Label))

# now set the axis labels,
# remove the legend, prepare for bw printing
p <- p + scale_x_continuous("My x axis",
                            breaks = seq(3, 5, by = 0.05)
                            ) +
  scale_y_continuous("My y axis") + theme_bw() +
  opts(legend.position = "none")


# Finally, print in a pdf
pdf("MyBars.pdf", width = 12, height = 6)
print(p)
dev.off()
```

2.3.8 Case study 4: mathematical display

In Figure 2.6, you can see the mathematical annotation of the axis and on the plot.

```
require(ggplot2)
```

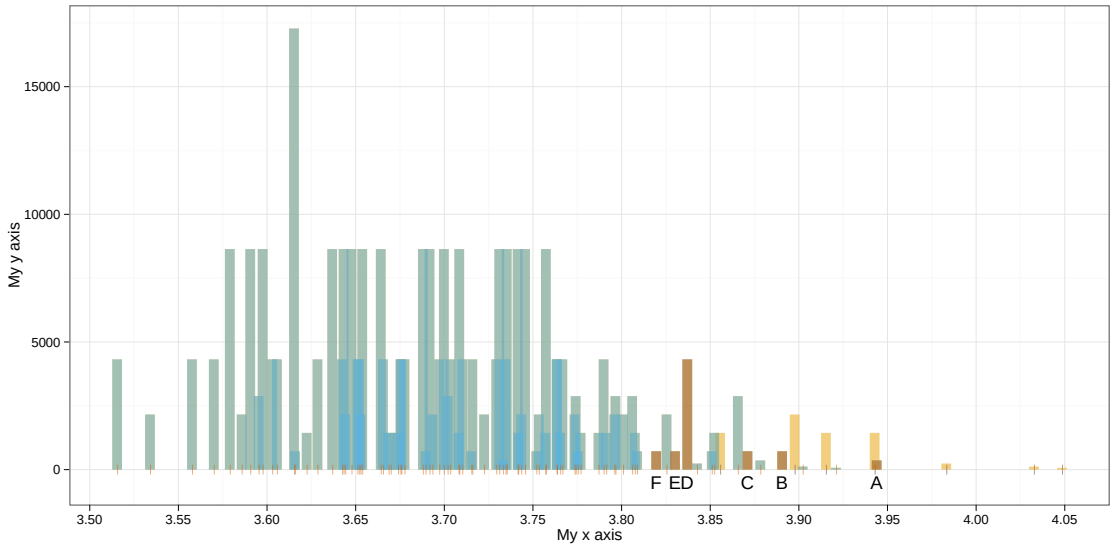


Figure 2.5: Overlay of three lineranges and a text geometry.

```
# create an "ideal" linear regression data!
x <- seq(0, 100, by = 0.1)
y <- -4. + 0.25 * x +
  rnorm(length(x), mean = 0., sd = 2.5)

# now a dataframe
my_data <- data.frame(x = x, y = y)

# perform a linear regression
my_lm <- summary(lm(y ~ x, data = my_data))

# plot the data
p <-  ggplot(my_data, aes(x = x, y = y,
                          colour = abs(my_lm$residual))
             ) +
  geom_point() +
  scale_colour_gradient(low = "black", high = "red") +
  opts(legend.position = "none") +
  scale_x_continuous(
    expression(alpha^2 * pi / beta * sqrt(Theta)))

# add the regression line
p <- p + geom_abline(
  intercept = my_lm$coefficients[1][1],
  slope = my_lm$coefficients[2][1],
  colour = "red")
# throw some math on the plot
p <- p + geom_text(aes(x = 60, y = 0,
                       label = "sqrt(alpha) * 2* pi"),
                       parse = TRUE, size = 6,
                       colour = "blue")

# print in a pdf
pdf("MyLinReg.pdf")
print(p)
dev.off()
```

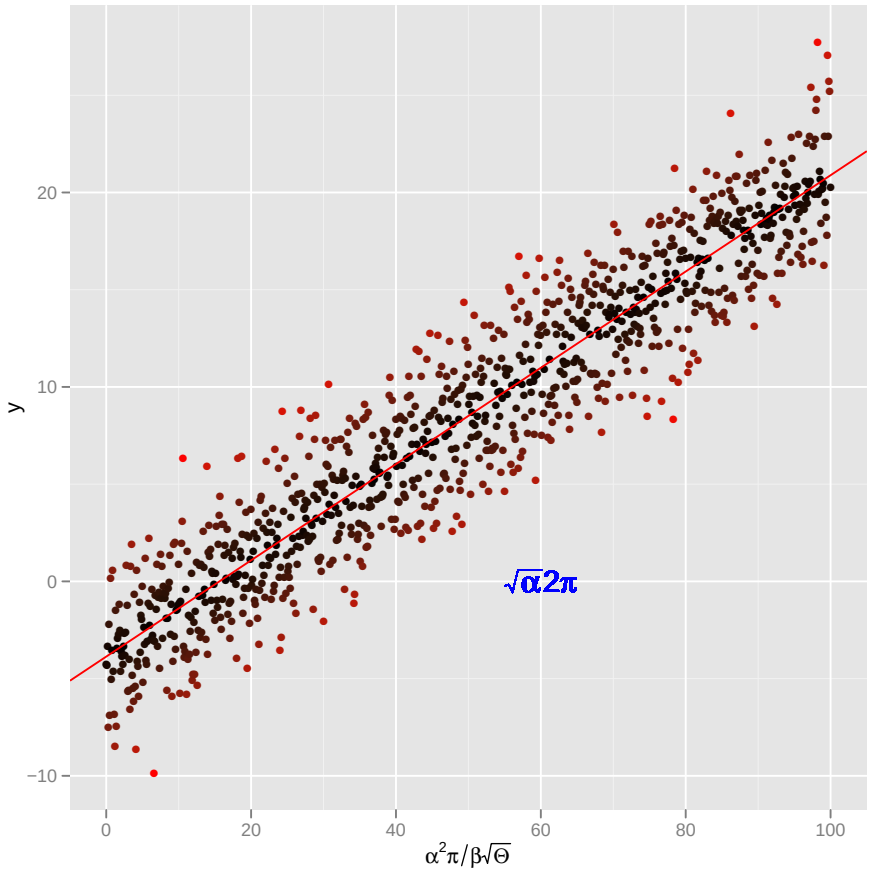
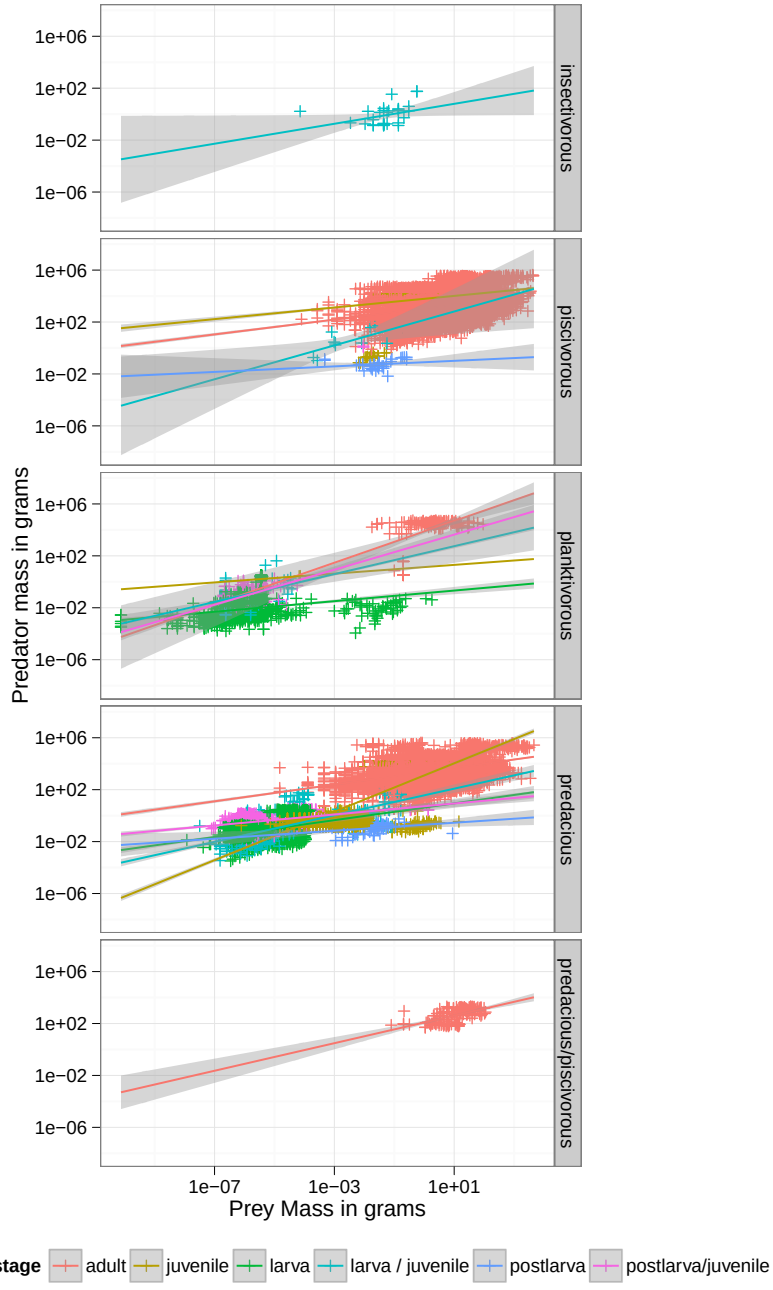


Figure 2.6: Linear regression with colors expressing residuals and mathematical annotations.

2.4 Practical

In this practical, you will write script that draws and saves a pdf file of the figure below, and writes the accompanying regression results to a formatted table in csv. Note that the plots show that the analysis must be subsetted by the `Predator.lifestage` field of the dataset. The guidelines are:

- Write a script file called `PP_Regress.R` and save it in the `Code` directory — sourcing or running this script should result in one pdf file containing the following figure being saved in the `Results` directory: (Hint: Use the `print()` command to write to the pdf)



Write a script that generates this figure

- In addition, the script should calculate the regression results corresponding to the lines fitted in the figure and save it to a csv delimited table called (`PP_Regress_Results.csv`), in the `Results` directory. (Hint: you will have to initialize a new dataframe in the script to first store the calculations and then `write.csv()` or `write.table()` it.)
  All that you are being asked for here is results of an analysis of Linear regression on subsets of the data corresponding to available Feeding Type $\times$ Predator life Stage combination — not a multivariate linear model with these two as separate covariates!
- The regression results should include the following with appropriate headers (e.g., slope, intercept, etc, in each Feeding type $\times$ life stage category): regression slope, regression intercept, R^2, F-statistic value, and p-value of the overall regression (Hint: Review the Stats week!).
- The script should be self-sufficient and not need any external inputs — it should import the above predator-prey dataset from the appropriate directory, and save the graphic plots to the appropriate directory (Hint: use relative paths). I should be able to `source` it without errors.
- I suggest using the `plyr` function instead of looping, and using the `ggplot` command instead of `qplot`...

Extra Credit: Do the same as above, but the analysis this time should be separate by the dataset's `Location` field. Call it `PP_Regress_loc.R`

2.5 Readings

Check out the Visualization directory on your master repository

- Rolandi et al. "A Brief Guide to Designing Effective Figures for the Scientific Paper", doi:10.1002/adma.201102518
- The classic Tufte `www.edwardtufte.com/tufte/books_vdqi` (btw, check out what Tufte thinks of PowerPoint!)
  Available in the Central Library, I have also added extracts and a related book in pdf on the master repository
- Lauren `et al.` "Graphs, Tables, and Figures in Scientific Publications: The Good, the Bad, and How Not to Be the Latter", doi:10.1016/j.jhsa.2011.12.041
- Effective scientific illustrations: `www.labtimes.org/labtimes/issues/lt2008/lt05/lt_2008_05_52_53.pdf`
- `https://web.archive.org/web/20120310121708/http://addictedtor.free.fr/graphiques/thumbs.php`
- Make `xkcd` style graphs in R: `http://xkcd.r-forge.r-project.org/`

Chapter 3

Experimental design and Data exploration

Ideally, you would like to design experiments (manipulations and/or observations) that are appropriate for the question you want to answer. However, you still need to explore you data to determine what kind of statistical tests would be appropriate because: (a) Your experiments or observations may not go as planned, and (b) You might have somebody else's data to analyse (very likely in your UG projects). Building on the previous chapter, in this chapter we you learn how to use R to explore your data and determine appropriate statistical tests. By the time you have worked through this chapter, you should be able to

- Provided sufficient information is available, be able to judge whether the sampling design used to generate a particular dataset was appropriate
- Determine if your sample sizes are adequate, especially for a specific statistical test qualitatively (Y1) or quantitatively using power analysis (Y2).
- Calculate basic statistical measures on your data to determine its properties

We are going to start off in Y1 with the simplest of scenarios for statistical testing — that you want to determine whether a sample, or a pair of samples meet some expectation (hypothesis) or not.

First, some conceptual preliminaries.

3.1 Some statistical parlance

The following terms are important for you to get familiar with:

(Statistical) Population A statistical population is a *complete set* of items that share at least one *attribute* of interest. This attribute of interest is the target of your statistical analysis. For example, if we are interested in studying the weight of year-old cod in the Oceans, the population consists of *all* year-old cod, but more specifically, the weight measurements of all the individuals of the cod population is what we want to analyse.

(Statistical) Distribution A statistical distribution is a mathematical description (expressed as a mathematical equation) of the properties of a population of interest. Theoreticians have come up with a bunch of distributions (e.g., Gaussian or Normal, Poisson, Binomial, etc.) that are appropriate for different kinds of data. Figuring out which distribution best describes a population of interest is one of the first steps in a statistical analysis. In reality, of course,

even collecting and measuring all the individuals of a population may not be sufficient to characterize its statistical properties — imagine the situation where the cod population has declined to a few hundred populations (not an impossibility in the future!).

(Data or Population) Sample A data *sample* is a set of measurements of the attribute of interest collected from a *statistical population* by a defined procedure (*sampling methodology*). In the cod example above, this would be the weight of every individual of a *subset* of the year-old Cod population.

(Statistical) Parameter A statistical parameter is a measure of some attribute of the *theoretical* statistical distribution that is support to represent your population. An example would be the average weight of yearling cod. In practice, this is not measurable because the population is much too large or incompletely inaccessible/invisible — imagine measuring the weight of every year-old cod individual in the oceans!

Statistic A statistic (singular) is an *estimate* of a statistical parameter of the population of interest, obtained by calculating the *measure* for a *sample* (e.g., the average or mean weight of individuals in a sample of one-year old cod). This is also your *descriptive statistic*. Therefore, a *Statistic* is to a *Statistical Parameter* what a *Sample* is to the *Statistical Population*. For example, the average of a sample of cod weights is a statistic that *estimates* the theoretical "real" average of the weights of the entire one-year Cod population, which is its statistical parameter.

Hypothesis A Hypothesis is a (hopefully) informed *postulate* about an attribute of your population of interest. For example, you may hypothesize that one-year old cod population's mean weight has declined across the last two decades (it has!). You will want to confront your main hypothesis with a "null" Hypothesis, to minimize the risk of making a "type I" error. A type I error is the probability of accepting an alternative (or main) hypothesis (and rejecting the null hypothesis) that is not really valid (e.g., the yearling cods have actually not declined in weight). This is a big NO NO from a scientific and philosophical standpoint. The rate of the type I error is denoted by the Greek letter α, and equals the *significance level* of a statistical test.

3.2 Descriptive Statistics

The fundamental statistics that describe a sample (or a population) are, firstly, the mean, or average value of a sample, typically denoted by a $\bar{x}$:

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n} = \frac{x_1 + x_2 + \cdots + x_n}{n} \tag{3.1}$$

That is, it is the sum of all the values in a sample divided by the number, n, of items in the sample. Thus it is a measure of the *central-tendency* of the sample and population.

Second, the standard deviation (s) is:

$$s = \sqrt{\frac{\sum_{i=1}^{n} (\bar{x} - x_i)^2}{n-1}} = \frac{(\bar{x} - x_1)^2 + (\bar{x} - x_2)^2 + \cdots + (\bar{x} - x_n)^2}{n-1} \tag{3.2}$$

That is, the the square root of the sum of squares ("SS") of the differences between each item in the sample and the mean, divided by the *degrees of freedom, "df"* remaining in the data set ($n-1$). df is the sample size, n, minus the number of statistical parameters estimated from the data set. This is to reduce the *bias* in your *estimate* of the statistic, as you are calculating it from the sample, and not the whole theoretical population.

Thus, the formula for s above has $n-1$ in its denominator because, to to work out the standard deviation, you must have already estimated the mean ($\bar{x}$) from the same data set. This removes 1 degree of freedom. Also, note that the sample variance, s^2 is the square of standard deviation. Or, in other words, the standard deviation is the square-root of the variance!

The above two statistics (mean and sd) are particularly meaningful when the sample and population have a symmetric distribution (e.g., normal or gaussian). When the distribution is not symmetric (that is, it is *skewed*), another statistic, the *median* becomes important. This is the middle value in the ordered set of data, that is exactly 50% of the data live below and 50% lie above the median. In skewed distributions (like the predator and prey mass distributions from the example in Chapter 2) is a better measure of the *central-tendency* of the sample and population.

Other descriptive statistics you should keep in mind are the range (difference between the largest and smallest values), and the quartiles (values lying in the data divided into the intervals $[\frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1]$ or at 1% intervals (percentiles). Box-plots, which you have seen, represent a number of these statistics in one figure.

3.2.1 Descriptive statistic functions in R

`mean(x)`	Compute mean (of a vector or matrix)
`sd(x)`	Standard deviation
`var(x)`	Variance
`median(x)`	Median
`quantile(x,0.05)`	Compute the 0.05 quantile
`range(x)`	Range of the data
`min(x)`	Minimum
`max(x)`	Maximum
`sum(x)`	Sum all elements

3.3 Data types and distributions

You will typically encounter or sample the following main types of data:

Continuous numeric This is the `numeric` or `real` data type in R, and as far as you are concerned, these data typically will be made up of (mathematically) real numbers such as human height or weight. These may be unbounded (any value between negative infinity to positive infinity), or bounded (e.g., between or zero and some upper positive number) like human weight.

Discrete numeric This is the `integer` data type in R, and consist of (mathematically) integer (whole) numbers such as counts of individuals in a population, e.g., The number of bacteria in a ml of pond water.

Percentage (proportion) Percentage data is a particular kind of numeric data that is strictly bounded between 0 and 100. The fact that you can never get samples of percentages that exceed these bounds makes the data tricky to analyse.

Categorical These are typically stored as the `character` data type in R. Categorical data are discrete, typically expressed as a one of a fixed number of levels of a factor. For example, the the factor "Type.of.feeding.interaction" from Chapter 1's predator-prey dataset had five levels: "insectivorous", "piscivorous", "planktivorous", "predacious", and "predacious/piscivorous".

Binary (presence/absence) data A special type of categorical data are binary, where only two categories or states are possible: (1, 0) (or "present", "absent"), e.g., a disease symptom. These may be stored as `integer` or `character` in R.

While designing experiments or exploring data obtained by somebody else, you need to keep in mind that each type will typically be best represented a particular *statistical distribution*. For example, continuous numeric data are *often* normally distributed. On the other hand, count data are likely to be distributed according to the Poisson distribution.

If you are lucky, you will mostly have to deal with data that are continuous or discrete numeric, which are the most straightforward to analyse using Linear models (more on that in subsequent chapters). However, some of the most interesting and important problems in biology involve proportion (percentage), categorical and binary data (e.g., Presence or absence of a disease symptom).

For example, think about what type of data, and what type of distribution, a sample of the following is likely to be:

- Wavelength of light
- Temperature
- Egg clutch size
- Rate of a reaction
- Eye-colour
- Score in Scrabble
- UG Degree class
- Ground-cover of grass in a quadrat
- Winning side in chess

3.3.1 Sampling from distributions in R

You can generate samples form many distributions in R (and handy thing to know). In particular, the following are important:

`rnorm(10, m=0, sd=1)`	Draw 10 normal random numbers with mean 0 and s.d. 1
`dnorm(x, m=0, sd=1)`	Density function
`qnorm(x, m=0, sd=1)`	Cumulative density function
`runif(20, min=0, max=2)`	Twenty random numbers from uniform [0,2]
`rpois(20, lambda=10)`	Twenty random numbers from Poisson(λ)

3.4 Two basic rules of experimental design and sampling

In general, while designing experiments, and sampling from a *population*, there are two key (and simple) rules:

1. **The more you sample, the more your sample's distribution will look like the population distribution** (obviously!)
2. **The more you sample, the closer will your sample statistic be to the population's statistical parameter** (the central limit theorem)

Let's have a quick look at rule 1 using R (open R and `setwd` to `Code`):

```
# Draw 5 normal random nos w/ mean 0 and s.d. 1:
> MySample5 <- rnorm(5, m=0, sd=1)
> MySample10 <- rnorm(10, m=0, sd=1)
> MySample20 <- rnorm(20, m=0, sd=1)
> MySample40 <- rnorm(40, m=0, sd=1)
> MySample80 <- rnorm(80, m=0, sd=1)
> MySample160 <- rnorm(160, m=0, sd=1)
```

Now let's visualize these "samples":

```
> par(mfcol = c(2,3)) #initialize multi-paneled plot
> par(mfg = c(1,1)); hist(MySample5, col = rgb(1,1,0), main = 'n = 5')
> par(mfg = c(1,2)); hist(MySample10, col = rgb(1,1,0), main = 'n = 10')
> par(mfg = c(1,3)); hist(MySample20, col = rgb(1,1,0), main = 'n = 20')
> par(mfg = c(2,1)); hist(MySample40, col = rgb(1,1,0), main = 'n = 40')
> par(mfg = c(2,2)); hist(MySample80, col = rgb(1,1,0), main = 'n = 80')
> par(mfg = c(2,3)); hist(MySample160, col = rgb(1,1,0), main = 'n = 160')
```

The second rule above states that if I was to repeat even $n = 5$ sufficient number of times, I would get a good *estimate* of mean (= 0) and standard deviation (= 1) of the normal distribution we sampled from.

3.5 A data exploration case study

As a case study, we will use data from a paper looking at the relationship between genome size and body size across species of dragonflies and damselflies (Odonata):

> Ardila-Garcia, AM & Gregory, TR (2009) 'An exploration of genome size diversity in dragonflies and damselflies (Insecta: Odonata)' Journal of Zoology, 278, 163 - 173

You will work with the script file `Chapter3.R`, which performs exploratory analyses on the data in `GenomeSize.csv`. Let's go through the code block by block.

- ⋆ Get the script `Chapter3.R` from the Bitbucket repository and put it in your own `Code` directory.
- ⋆ Also get `GenomeSize.csv`
- ⋆ Open the script `Chapter3.R` in RStudio (or some other text editor).
- ⋆ Use the shift and arrow keys to select the code in block (2), including the comments. Now use the keyboard short cut (look back at Chapter 2 if you don't know how!) to run the highlighted block of code.

This first line (block (1)) reads in the data, as we learned in Chapter 2.

⋆ Now the code in block (2) line by line of code.

Have a good look at the data. There are three factors (categorical variables): Suborder, splitting the species into dragonflies (Anisoptera) and damselflies (Zygoptera); Family, splitting the species further into 9 taxonomic families; and Species, giving the latin binomial for each species in the table. The remaining columns are measurements of genome size (in picograms) and measurements of body size and morphology (in grams, mm and mm^2). There are two columns ending with an N that show the sample size from which the observations for each species are taken and a column ending SE showing standard errors.

One thing you should see in the output from `head` or `str` is that there are some observations marked as `NA` – this is the way R shows *missing data*. It is important to check how much missing data there is an dataset, so we'll use another function that includes this information. Many R functions refuse to use variables containing missing data — this is just R being careful and you can add `na.rm=TRUE` into most functions to avoid this problem.

⋆ Run the `summary` line from the script window (block 3).

Look at the output. There is a column for each variable: for factors, it provides a short table of the number of observations in each level and for continuous variables, it provides some simple summary statistics about the distribution (range, quartiles, mean and median), and the number of missing values

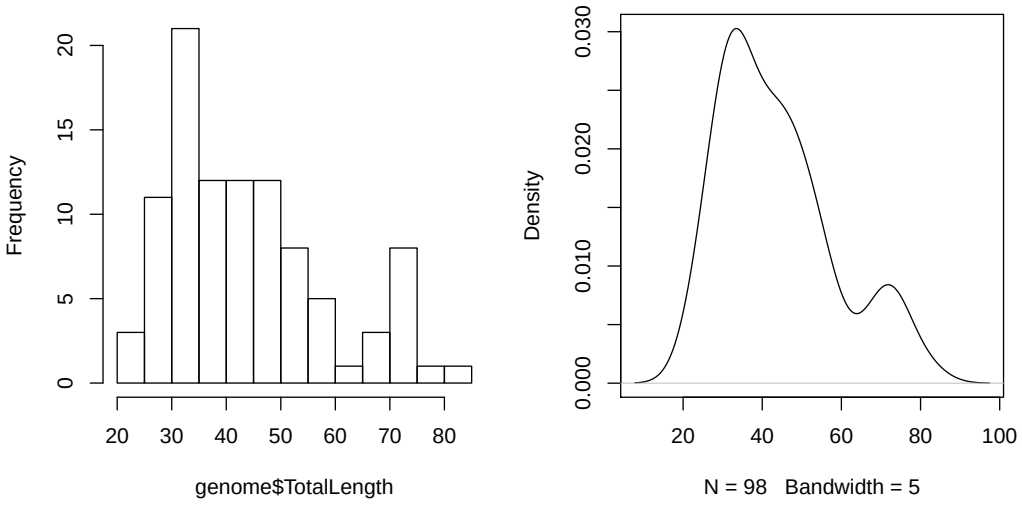
3.5.1 Visualise distributions of the variables

The `summary` function shows us the basic distribution (range, quartiles, mean and median) of a continuous variable, but this is easier to interpret if we visualise it. We'll look at two ways:

Histogram In the simplest form, this shows the number of observations of the variable falling into a set of bins spanning the range of the variable. The option `breaks` allows you to change the number of bins.

Density plot Rather than showing blocks of counts, the density plot shows a continuous smooth line. This is a *smoothed* estimate of the how frequently data is observed across the range of values and the *bandwidth* (`bw=0.1`) controls the degree of the smoothing.

⋆ Go to block (4) of the script and run each line separately, looking at the output.
⋆ In the editor, change the values of `breaks` and `bw` — for example `breaks=5` and `bw=0.05` — and re-run these lines to see how this affects the graph. Basically, with both types of graph you can look at the data too coarsely or too finely.
⋆ The graphs you've just created look at genome size. Add a copy of those two lines of code in the script and change them to look at the variable `TotalLength`. You will need to alter the `density` function to ignore missing values (`na.rm=TRUE`) and to play around with the bandwidth. You should get something like this:



3.5.2 Take a quick look at effects of certain factors

R has a special way of describing a model that defines the response variable and the explanatory variables ("factors"). This is called a 'formula' and is used to define linear models (more on these in a later chapters). The same structure is used in many plotting functions and will put the response variable on the *y* axis and the explanatory variable on the *x* axis. The structure is "response variable $\sim$ explanatory variables". We will look at multiple explanatory variables in a later practical but an example with one explantory variable (factor) is:

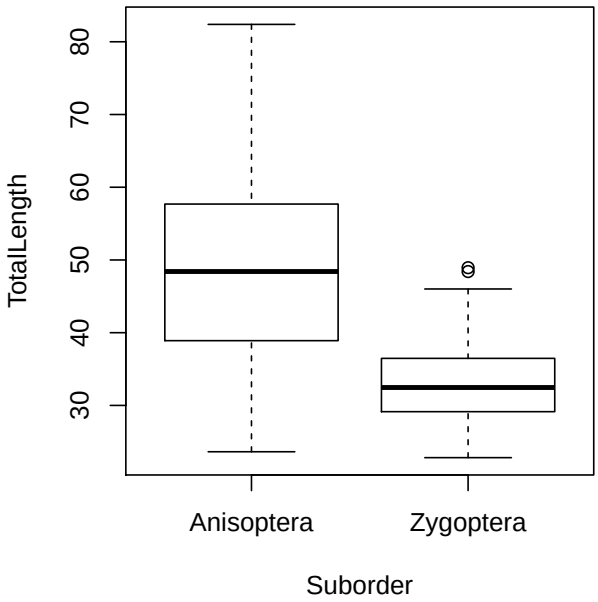
```
Genome Size ~ Suborder
```

This formula tells R to model genome size 'as a function of' ($\sim$) the suborders of Odonata. In a plot function, the result will be to plot genome size as a function of the suborders.

3.5.3 Compare distribution of the variable across levels of a factor

Although looking at the distribution of variables is a good first step, we often want to compare distributions. In this case, we might want to know how genome size varies between dragonflies and damselflies. The first way we will look at is using boxplots — these show the median and the 25% and 75% quantiles as a box, with whiskers extending to the minimum and maximum. More extreme outliers are plotted independently as points. The `plot` function in R automatically generates a boxplot when the explanatory variable is a factor.

- ⋆ Go to block 5 of the script and run the first line, looking at genome size between the two suborders.
- ⋆ Duplicate and alter this line to look at the same plot for total length. You should get a plot like this:

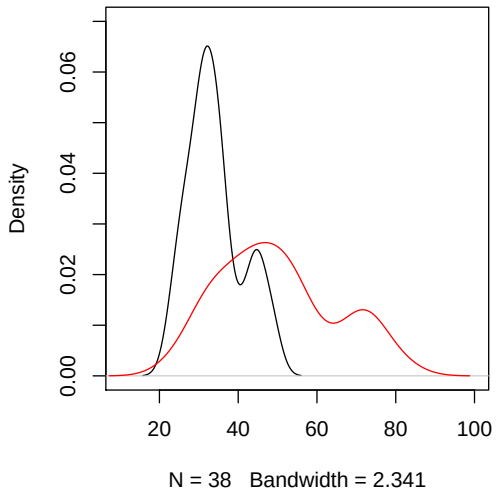


Although histograms are great for one variable, plotting two histograms on top of one another rarely works well because the overlapping bars are hard to interpret (recall the body size example from Chapter 2). Density plots don't have this problem but it takes a bit more code to create the plot.

- ⋆ block 6 of the script uses the `subset` function to create two new data frames separating the data for dragonflies and damselflies. Run the first two lines of this block. Remember that the arrow symbol (`<-`) is used to save the output of a function into a new object in R — if you use `ls()` in the console, you will see the two new data frames.
- ⋆ In the console, use `str` and `summary` to explore these two new dataframes.

Now that we've got the data separated we can go about plotting the two curves.

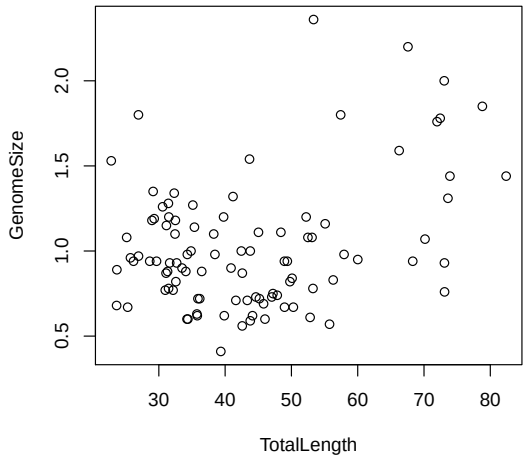
- ⋆ Run the next two lines of code in block 6. The first draws the plot for damselflies and the second adds a line for the dragonflies.
- ⋆ Duplicate these last two lines of code and edit them to generate a similar plot for total body length. You will need to edit the code to change the range of the x and y axes (`xlim` and `ylim`) to get both curves to fit neatly on to the graph. It should look like this:



3.5.4 Explore further by scatter-plotting two variables

Once we've looked at the distribution of variables, the next thing is to look at the relationships between continuous variables using scatterplots. The `plot` function in R automatically generates a scatterplot when the explanatory variable is continuous, so we can use the same syntax and structure as for the boxplot above.

- ⋆ Go to block (7) of the script and run the first plot command to plot body weight as a function of genome size.
- ⋆ Create a new line of code to plot forewing area as a function of genome size. It should look like this:



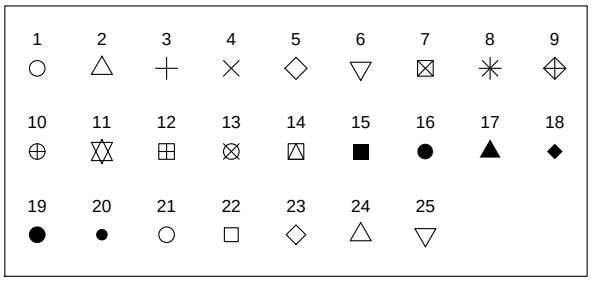
The scatterplot seems to show a weak relationship between genome size and morphology. But maybe dragonflies and damselflies show different relationships, and we can't distinguish between them! To explore this possibility, we need to plot the two orders using different colours or plot characters. In the next code block, we want customize the plots to show different types of points for each suborder. It is done by using *indexing* (see Chapter 1).

- ⋆ Run the first two lines of code in block 8. There are two levels of suborder and these two lines set up a colour and a plot symbol that will be used for each one.
- ⋆ Run the next line which shows the structure of the factor `Suborder`. You can see that there are two levels, with Anisoptera first and then Zygoptera. You can also see that these are stored as numeric values: 1 refers to the first level and 2 will refer to the second. We can use these as *indices* to pair the colours and plot symbols to each suborder. These are set in the `plot` function using the options `col=` and `pch=`, which stands for "plot character".
- ⋆ Run the next plot command to see the resulting plot — each point gets the appropriate colour and symbol for its group.

There are a lot of built in colours and plot symbols in R, so the next thing to experiment with is changing these to your own versions.

- ⋆ In the console, type in the function `colors()`. You'll see a long list of options to choose from, so pick two to replace red and blue in the script window.
- ⋆ The options for the plot symbols are shown below. Pick two to replace the current symbol choices.

⋆ Rerun the `plot` function and see what you get!



3.5.5 Saving the exploratory graphics

The file 'GenomeSize.pdf' in the practical folder was created using the next block of code using the approach you learned in Chapter 1. The function `pdf` opens a new empty pdf file which can then be used to plot graphs. You can set the width and the height of the page size in the pdf but note that this is set in *inches*. When you have finished creating a plot, the function `dev.off` closes the pdf file and makes it readable.

⋆ Open 'GenomeSize.pdf' in a PDF reader. It uses the original colours and plot symbols. Close the file and then delete it from the folder.
⋆ Now go back to the script in R and select and run all the code in block (9)
⋆ Go back to the `Results` folder. The pdf file should have been recreated — open it and it should now use your choice of colours and symbols.

3.5.6 Saving data

One last thing you can do is to save the data and variables in R format — the original data, two subsets of the data and the two sets of colours and symbols. We can recreate the subsets easily, so we'll just save the data and your colour sets.

⋆ Go to the script window and run the final line in block (10)
⋆ Still in the script window, choose 'File ▷ Save' to save your changes to the script file.
⋆ Quit from R `q()`.

Chapter 4

t tests and *F* tests

Aims of this chapter:

- Using *t* tests to look at differences between means
- Using *F* tests to compare the variance of two samples
- Using non-parametric tests for differences

In the last chapter, we looked at the genome size and morphology of species of dragonflies and damselflies (Odonates: Anisoptera and Zygoptera). Box and whisker plots and density plots both show that the two groups have rather different genome size and morphology. We can use *t* tests to test whether the means of the variables of the two groups are different and we can use the *F* test to check whether they have the same variance.

In this chapter, we will continue to practise building scripts and creating your own R code, so we will start from an empty script file. Use this script to store the code you run in this practical session and add notes to keep a record of what each bit is doing:

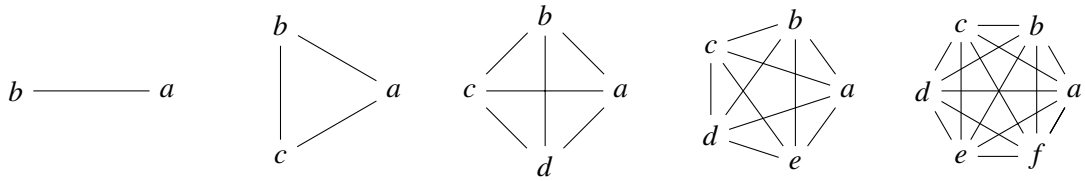
⋆ Open R and change to the `Code` directory.
⋆ If you have misplaced the data then it is on Blackboard / Bitbucket, so download it again.
⋆ Create a new blank script called `ttests.R` and save it to the working directory.
⋆ Put a comment at the top (using #) to describe the script and.
⋆ For the rest of this session, type your code into this script, adding comments and then run them in R using Ctrl+R. If you make mistakes, correct the script and run the code again. This way you will end up with a complete neat version of the commands for your analysis.
⋆ Add code to your script to load the genome size data into R, assigning the object name `genome` and use `str(genome)` to check it has loaded correctly.

4.1 Using *t* tests

The *t* test is used to compare the mean of a sample to another value: either some reference point (Is the mean different from 5?) or another mean (Does the mean of A differ from the mean of B?). If you have a factor with two levels then a *t* test is a good way to compare the means of those samples. If you have more than two levels, then you have a problem: as the number of levels (n) increases, the number of possible comparisons between pairs of levels (p) increases very rapidly[1]. Making these many comparisons using something like a t-test is a problem because we neglect

[1]The number of pairs is the binomial coefficient $C(n,2)$ — inevitably, R has a function for this: `choose(2:6,2)`

the covariance among measures and inflate the chance of falsely rejecting at least one hypothesis (Type I error – recall hypothesis testing in the previous chapter).

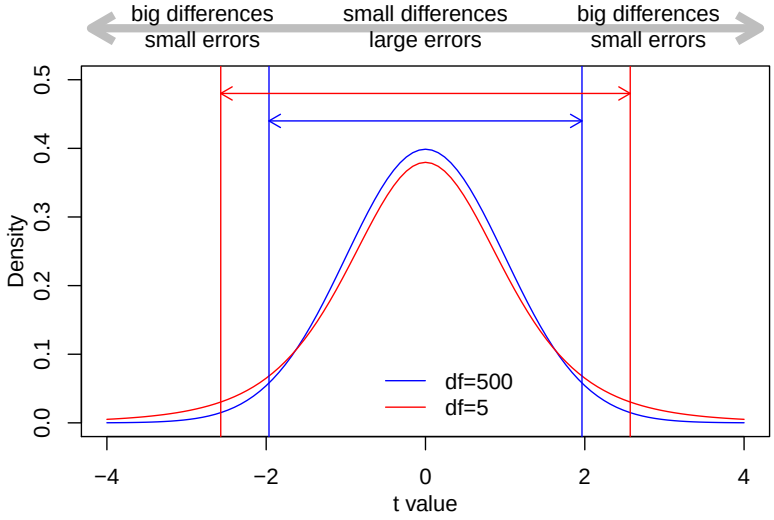


$n = 2, p = 1$ $n = 3, p = 3$ $n = 4, p = 6$ $n = 5, p = 10$ $n = 6, p = 15$

The basic idea behind the t test is to divide the difference between two values by a measure of how much uncertainty there is about the size of that difference. The measure of uncertainty used is the *standard error*.

$$t = \frac{\text{difference between values}}{\text{standard error}}$$

When there is no difference between the values t will be zero, but with big differences and/or small errors, will be larger. The t distribution below shows how commonly different values of t are found under the null hypothesis.



Some points about the plot above:

- The null hypothesis is that there is no difference between the values but because we only estimate the values from samples, differences will creep in by chance.
- Mostly these differences will be small — hence the peak in the middle — but sometimes the differences will be large and the errors will be small.
- 95% of the area under the curves are between these two sets of vertical lines. Values of t more extreme than this will only occur 1 in 20 times or with a probability (p) of 0.05.
- The means of small samples are more easily influenced by extreme values and so produce extreme t values more frequently. This is why the red curve above for smaller samples is more flattened out and why the 95% limits are more spread out.

4.2 One sample t tests

In the simplest example, a t test can be used to test whether the mean of a sample is different from a specific value. For example:

- Is the ozone in a set of air samples above the legal limit?
- Is the change in a set of patient' weights different from zero?
- Is the mean genome size for Odonata smaller than 1.25 pg, which is the average for insects [see here]?

Oh look! We can test that last one...

To calculate t, we need that observed difference and then the standard error of the difference between the mean of our sample and the known value. This is calculated using the *variance* and the *sample size* (n) of the sample (s).

$$se_s = \sqrt{\frac{\text{var}(s)}{n}}$$

This simple equation trades off variance — high variance in the data gives higher uncertainty about the location of the mean — and sample size – more data gives more certainty. So, *low variance* and *large datasets* have *small* standard errors; *high variance* and *small datasets* have *large* standard errors. Variance is calculated using sums of squares and so the square root is needed to give a standard error in the same units as the mean.

So, all we need are three values calculated from the data: mean, variance and the number of data points and we can calculate t. R can do this for us:

```
# calculate the three values from the data:
> mean.gs <- mean(genome$GenomeSize)
> print(mean.gs)
[1] 1.014
> var.gs <- var(genome$GenomeSize)
> print(var.gs)
[1] 0.1397
> n.gs <- length(genome$GenomeSize)
> print(n.gs)
[1] 100
# get the difference
> diff <- mean.gs - 1.25
> print(diff)
[1] -0.2357
# get the standard error
> se.gs <- sqrt(var.gs/n.gs)
> print(se.gs)
[1] 0.03738
# get the t value
> t.gs <- diff/se.gs
> print(t.gs)
[1] -6.306
```

⋆ Copy and paste the code above into your script in R and run it. Read through the code and make sure you understand the steps.

This is a big t value — values this extreme don't even appear on the graph above — so we would conclude that the mean genome size for Odonata is different from the average for insects.

We can do this more easily and get some more information using the function `t.test`. The null hypothesis can be set using the option (sometimes called a function *argument*) `mu` — the Greek letter μ is often used to refer to a mean:

```
> t.test(genome$GenomeSize, mu = 1.25)

    One Sample t-test

 data:  genome$GenomeSize
 t = -6.306, df = 99, p-value = 8.034e-09
 alternative hypothesis: true mean is not equal to 1.25
 95 percent confidence interval:
  0.9401 1.0885
 sample estimates:
 mean of x
     1.014
```

This confirms the values we calculated by hand and adds a p value. The output also gives the degrees of freedom. This is something we will come back to later, but the degrees of freedom are basically the number of data points minus the number of estimated parameters, which in this case is one mean.

The output also gives a confidence interval for the observed mean. The mean is the best estimate of the population mean given our sample of species of Odonata, but the actual mean for the order could be bigger or smaller. The confidence interval tells us the region in which we are 95% confident that this actual mean lies.

It is calculated using the t distribution. Remember that t is a difference divided by a standard error; if we multiply t by a standard error, we get back to a difference. If we pick a pair of t values that contain the middle 95% of the t distribution, as in the plot on page 2, then we can multiply that by the standard error from the data to get a range above and below the mean. If we sampled lots of sets of 100 species of Odonata, we expect 95% of the observed means to lie inside this range. The code below shows the calculation of the confidence interval for the test above.

```
# Find the edges of the middle 95% of a t distribution with 99 df
# (quantiles of the t distribution, so qt)
> tlim <- qt(c(0.025,0.975), df = 99)
> print(tlim)
[1] -1.984 1.984
# use the mean and standard error from above to get a confidence interval
> mean.gs + tlim * se.gs
[1] 0.9401 1.0885
```

⋆ Using the `t.test` code above as an template, test whether the body weight (in grams) of Odonata is different from the average[2] for arthropods of 0.045 grams.

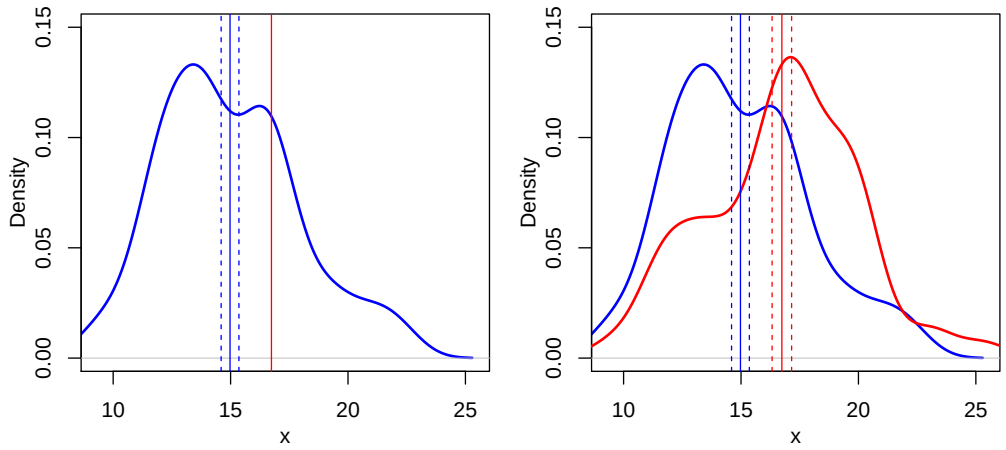
4.3 Two sample t tests

It is more common to use a t test to compare the means of two samples. This includes questions like:

[2]This slightly dodgy estimate comes from an estimated average volume for arthropods of of 45.21 mm^3 and assuming a density of 1 gm per cm^3. The volume is from: Orme, C. D. L., Quicke, D. L. J., Cook, J. M. and Purvis, A. (2002), Body size does not predict species richness among the metazoan phyla. Journal of Evolutionary Biology, 15: 235–247.

- Do two rivers have the same concentration of a pollutant?
- Do chemical A and chemical B cause different rates of mutation?
- Do damselflies and dragonflies have different genome sizes?

The main difference here is that with a one sample t test, we assume that one of the means is known exactly: the only error is in the single sample. With a two sample test, we are comparing two means estimated from samples and both contain error. The graph below illustrates this:



The vertical lines show the mean (solid lines) and one standard error to each side (dashed lines). The red mean is the same in both cases, but the second graph shows that this is also estimated from a sample with error: the difference in the means looks less convincing and we'd expect a smaller t value. The t tests in for these two graphs confirm this:

- The mean for blue is significantly different from 16.74 (mean=14.98, se=0.38, df=59, t=-4.65, p=0.00002).
- The means of blue and red are significantly different (blue: mean=14.98, se=0.38; red: mean=16.74, se=0.42; df=118, t=-3.13, p=0.002)

⋆ Have a close look at the previous two statements. This shows the kind of detail needed when reporting the results of t tests. The following is *not* acceptable: The means of blue and red are significantly different ($p = 0.002$).

So, with two samples, we shouldn't be so confident about the difference between the values — it should have a higher standard error. We can do this simply by combining the variance and sample size for the two samples (a and b) into the calculation:

$$se_{a-b} = \sqrt{\frac{\text{var}(a)}{n_a} + \frac{\text{var}(b)}{n_b}}$$

We'll use a t test to address that last question: are the genome sizes of Anisoptera and Zygoptera different? First, we'll do this by hand. We'll use a really handy function `tapply(X, INDEX, FUN)` to quickly find the values for the two groups: it takes some values (X), splits those values into groups based on a factor (INDEX) and runs each group through another function (FUN).

```
# calculate the three values from the data
> mean.gs <- tapply(X = genome$GenomeSize, INDEX = genome$Suborder, FUN = mean)
> print(mean.gs)
```

```
Anisoptera  Zygoptera
1.018               1.012
> var.gs <- tapply(X = genome$GenomeSize, INDEX = genome$Suborder, FUN = var)
> print(var.gs)
Anisoptera  Zygoptera
0.18458         0.06946
> n.gs <- tapply(X = genome$GenomeSize, INDEX = genome$Suborder, FUN = length)
> print(n.gs)
Anisoptera  Zygoptera
38                  62

# get the difference
> diff <- mean.gs[1] - mean.gs[2]
> print(diff)
Anisoptera
-0.006647

# get the standard error of the difference
> se.gs <- sqrt((var.gs[1]/n.gs[1]) + (var.gs[2]/n.gs[2]))
> print(se.gs)
Anisoptera
0.06932

# get the t value
> t.gs <- diff/se.gs
> print(t.gs)
Anisoptera

-0.09589
```

- ⋆ Type the code above into your script in R and run it. Again, read through the code and make sure you understand the steps.

The `t.test` function automates this all for us, and we can use a formula (remember previous Chapter) to get a test between the two suborders.

```
>  t.test(GenomeSize ~ Suborder, data = genome)

    Welch Two Sample t-test

 data:  GenomeSize by Suborder
 t = -0.0959, df = 98, p-value = 0.9238
 alternative hypothesis: true difference in means is not equal to 0
 95 percent confidence interval:
  -0.1442  0.1309
 sample estimates:
 mean in group Anisoptera  mean in group Zygoptera
                   1.012                    1.018
```

The output looks very similar to the one sample test, except that the output now gives two estimated means, rather than one and it reports the p value for the calculated t value.

- ⋆ Add this to your script and run it.
- ⋆ Copy and modify this in your script to test whether the body weight of the two suborders are different.

4.4 F tests for equal variance

F tests are used to compare the variances of two samples or populations. You will most prominently see them in analysis of variance (ANOVA), to test the hypothesis that the means of a given

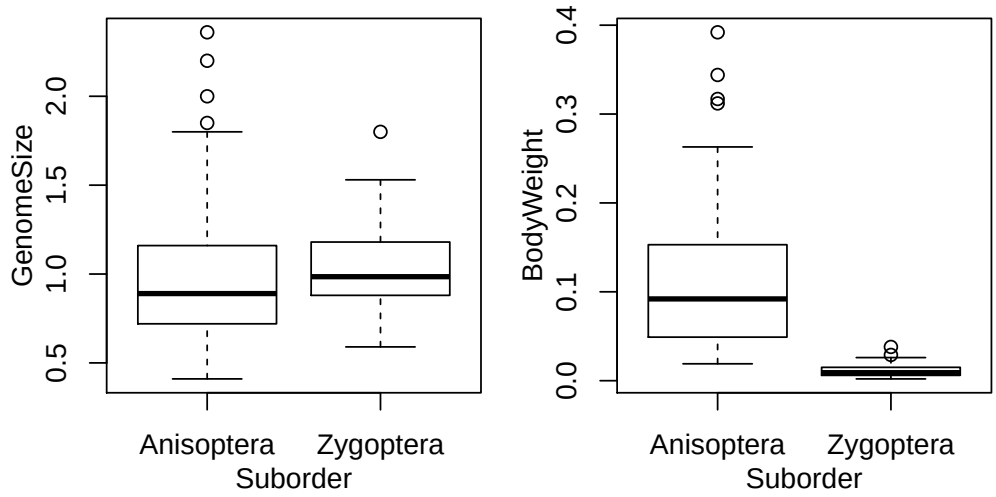
set of normally distributed populations all have the same variance.

Let's use our genomesize datast to have a look at F-tests as well. Ideally, the *t* test should be used with data that are: i) *relatively normally distributed*, so that means can be estimated sensibly; and ii) have *similar variances*. We'll deal with the similar variance question here using an F test for equal variances.

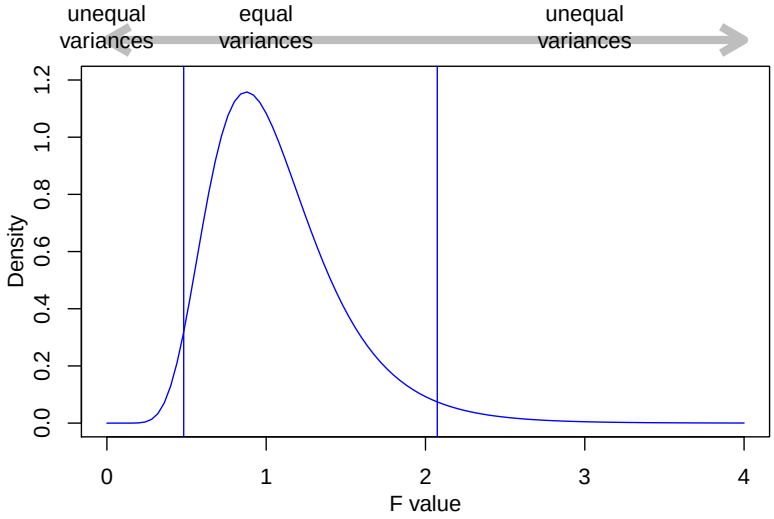
First, let's visualize the data. As I hope you've already noticed, this session has been neglecting one very important part of analysis — *plotting the data*. We are going to compare two plots, so it helps to have them side by side in the same window. We can use the function `par` to change a set of options called graphics parameters to get R to do this. The option to change is `mfrow`. This sets a graphics window to include *m*ultiple *f*igures and we need to tell R the number of rows and columns to divide the window into: `par(mfrow=c(1,2))`.

- ⋆ Copy `par(mfrow=c(1,2))` into your script, add a comment and run it.
- ⋆ Using your skills from Chapter 1, create a boxplot comparing the genome sizes of the two suborders.
- ⋆ Add another boxplot beside it comparing the body weight of the two suborders.

It should look like this:



The distribution of the test statistic F is simply the ratio of the variances for sample a and b: $\text{var}(a)/\text{var}(b)$. If the two variances are the same then $F = 1$; if $\text{var}(a) > \text{var}(b)$ then $F > 1$; and if $\text{var}(a) < \text{var}(b)$ then $F < 1$:



The two vertical blue lines again show the edges of the central 95% of the area of the curve: if the two samples are drawn at random from a population with the same variance then values of $F < 0.482$ or > 2.074 are observed fewer than 1 time in 20 ($p \leq 0.05$ again)[3]. The shape of the F distribution changes depending on the amount of data in each of the two samples but will always follow this broad pattern: centred near 1 and with a longer tail to the right. Note that the F-distribution arises as the ratio of two appropriately scaled *chi-square distributed variates*, because, as we saw above, variances should be chi-square distributed.

We can use R to calculate F for the variance in genome size in each of the two suborders. We calculated the variance for the t test above, so we can just do this:

```
> var.gs[1]/var.gs[2]
Anisoptera
     2.657
```

That's quite a big F value and we can use the function `var.test` to do all the calculations for us and give us the actual p value:

```
> var.test(GenomeSize ~ Suborder, data = genome)
    F test to compare two variances

 data:  GenomeSize by Suborder
 F = 2.657, num df = 61, denom df = 37, p-value = 0.001946
 alternative hypothesis: true ratio of variances is not equal to 1
 95 percent confidence interval:
  1.449 4.671
 sample estimates:
 ratio of variances
              2.657
```

It produces the same value that we calculated by hand and shows that, if the two samples are drawn from populations with the same variance, an F value this extreme will only be observed roughly 1 time in 500 ($1/0.00195 \approx 500$) .

⋆ Open a new empty script called `FTests.R`.

[3] Note that $1/0.482 \approx 2.074$ and $1/2.074 \approx 0.482$: in this case, it doesn't matter which way round you compare the two variances!

⋆ In this write your script to test whether the variances in the body weight of the two suborders from the `GenomSize` dataset are different.

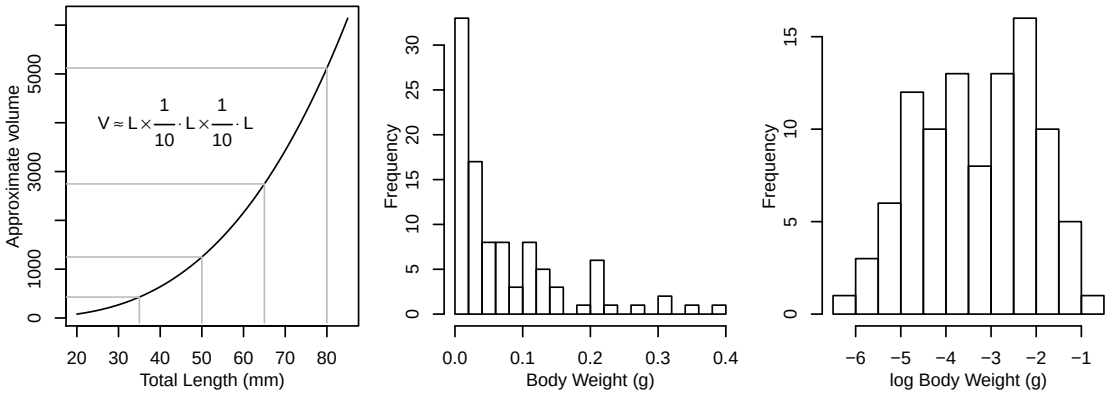
There are clearly problems with the variance in both examples. The next two sections present ways to address these kinds of problems.

4.5 *t* tests revisited

The first thing to say is that R is aware of the problem with the variance. If you look back at the output from the previous *t* tests, you will see that the degrees of freedom vary a lot. We have 100 observations and – after subtracting one for each mean we calculate — our degrees of freedom should be either 99 (one sample test) or 98 (two sample test). What we actually see are smaller numbers, with the smallest being `df = 60.503` for the two sample test of body weight.

The explanation is that R is applying a *penalty* to the degrees of freedom to account for differences in variance. With fewer degrees for freedom, more extreme *t* values are more likely and so it is harder to find significant results. This doesn't mean we can forget about checking the variances or plotting the data!

In this case, we can also apply a transformation to the data in order to make the variances more equal. Forgetting the wings and assuming Odonata are shaped like a box, the model in the graph below shows how volume changes with length: equal changes in length do not lead to equal changes in volume and longer species will have a disproportionately large volume. This is a classic feature of morphological data known as allometric scaling and we'll look at it again in a the next Chapter. In the meantime, a log transformation will turn body weight from a skewed distribution to a more normal distribution.



$\log_e$ body weight as follows:

```
> genome$logBodyWeight <- log(genome$BodyWeight)
```

⋆ Copy the line into your script and run it.
⋆ Now write three lines of code to get a boxplot of $\log_e$ body weight and then run a variance test and *t* test on the differences in $\log_e$ body weight between suborders.

This gives a much clearer result — the variances are almost identical and the differences between the suborders are much more cleanly tested.

4.6 Non-parametric tests

What happens if there isn't a convenient transformation for the variable that gives roughly constant variation and equal variance? In a parametric test, like the t and F test above, we use parameters (mean and variance) to describe the data, assume these describe the data well and then just use these parameters to run the test. If these assumptions don't seem very sound, the non-parametric tests provide a way of using the ranks of the data to test for differences. They aren't as powerful — they are less likely to reveal significant differences — but they are more robust. The most commonly used alternative is the Wilcoxon test, which uses the function `wilcox.test` in R.

- ⋆ Using `wilcox.test` as a replacement for `t.test`, repeat the one and two sample t test for genome size and body weight.
- ⋆ Compare the two results.
- ⋆ repeat the same with the Predator and prey body mass data from the previous chapter – check how different the results are when using t vs. Wilcoxon test.

Chapter 5

Linear Models: Regression

Aims of this chapter:

- More functions for plotting data and models.
- Calculating correlation coefficients.
- Fitting a regression model and significance testing.
- Using diagnostic plots to assess model suitability.

As with the previous chapter, we'll start with creating a new blank script for you to fill in during the practical. We'll also be using the genome size data again, so:

⋆ Open R and change to the `code` directory.
⋆ Create a new blank script called 'Regression.R' and add some introductory comments.
⋆ Add code to your script to load the genome size data into R and check it.

5.1 Exploring the data

In previous chapters we used `plot` to create a scatterplot between two variables. If you have a set of variables to explore, writing code for each plot is tiresome, so R provides a the function `pairs`, which creates a grid of scatterplots between each pair of variables. All it needs is a dataset.

⋆ Add `pairs(genome, col=genome$Suborder)` into your script and run the code.

The result is messy! There are far too many variables in `genome` for this to be useful. We need to cut down the data to fewer variables. In Chapter 4, we used indices to select colours; here, we can use indices to select columns from the data frame. This again uses square brackets (`x[ ]`), but a data frame has two dimensions, rows and columns, so you need to provide an index for each dimension, separated by commas. If an index is left blank, then all of that dimension (i.e. all rows or columns) are selected. Try the following to re-acquaint yourself to access data frame content using indices:

```
# create a small data frame:
> dat <- data.frame(A = c("a", "b", "c", "d", "e"), B = c(1, 2, 3, 4, 5))
> dat[1, ] # select row 1 (all columns selected)
  A B
1 a 1

> dat[, 2] # select column 2 (all rows selected)
[1] 1 2 3 4 5
```

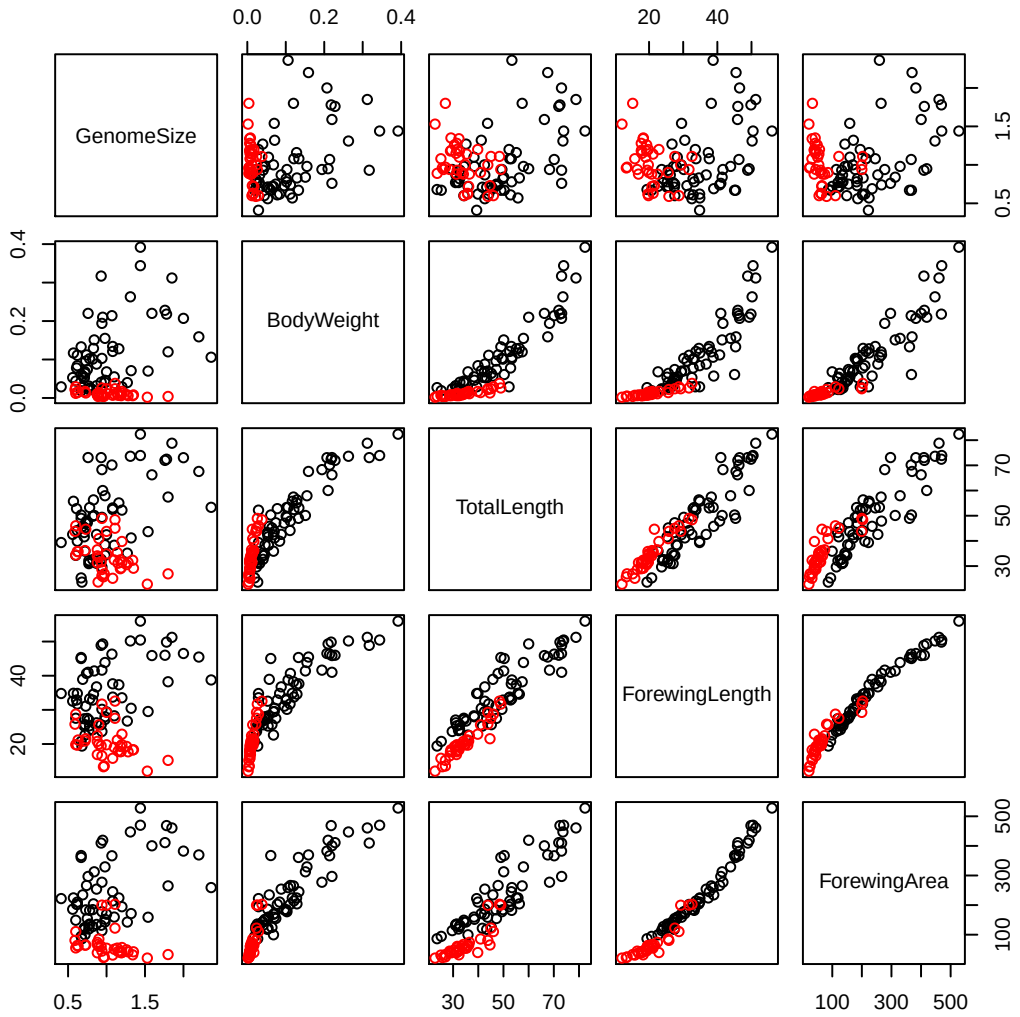
```
> dat[2, 1] # select row 2, column 1
[1] "b"
```

Now let's get started with the actual analysis. We will look at five key variables: genome size, body weight, total length, forewing length and forewing area. If you look at the output of `str(genome)`, you'll see that these are in columns 4, 7, 8, 12 and 14. We can record the indices of these columns and use this to select the data in the pairs plot.

```
morpho <- c(4, 7, 8, 12, 14)
pairs(genome[, morpho], col = genome$Suborder)
```

⋆ Add the code above to your script and run it

The `pairs` plot should give you something like the plot below:

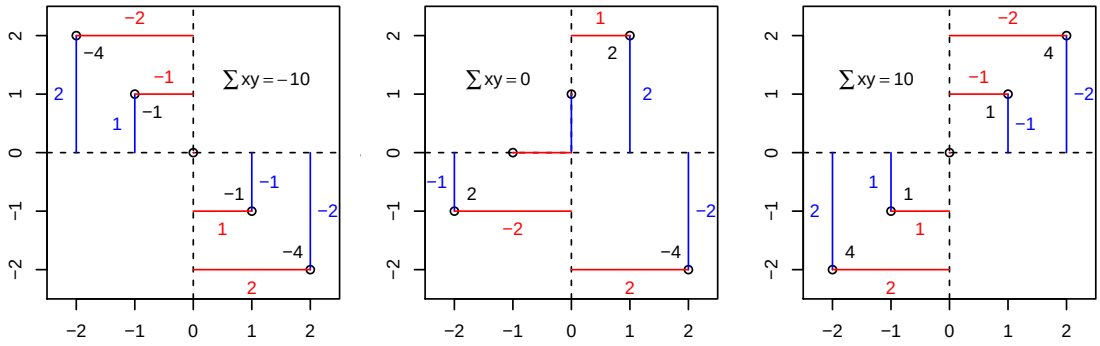


Each scatterplot is shown twice, with the variables swapping between the x and y axes. You can see immediately that the relationships between the four morphological measurements and genome size are fairly scattered but that the plots comparing morphology show much clearer relationships.

5.2 Correlations

One way of summarising how close strong the relationship between these variables are is to calculate a correlation coefficient. Pearson correlations look at the difference of each point from the mean of each variable (and since it uses means, it is a parametric statistic).

It is calculated using of the differences from the mean on each axis. The key calculation is — for each point – to get the product of the differences on each axis and add them up. If the points are mostly top left ($-x$, y) or bottom right (x, $-y$) then these products are mostly negative ($-xy$); if the points are mostly top right (x, y) or bottom left ($-x$, $-y$) then the products are mostly positive (xy).



The plots above show three clear cases where all the values of xy are negative or positive or where both are present and sum to zero. The Pearson correlation coefficient simply scales these sums of xy to be between -1 (perfectly negatively correlated) and 1 (perfectly positively correlated) via zero (no correlation).

We will use two functions to look at correlations. The first is `cor`, which can calculate correlations between pairs of variables, so is a good partner for `pairs` plots. The second is `cor.test`, which can only compare a single pair of variables, but uses a t test to assess whether the correlation is significant.

⋆ Try the following (and include it in your R script file)

```
> cor(genome[, morpho], use = "pairwise")
```

You should see a correlation matrix. Then,

```
> cor.test(genome$GenomeSize, genome$TotalLength, use = "pairwise")

Pearson's product-moment correlation

data: genome$GenomeSize and genome$TotalLength
t = 3.551, df = 96, p-value = 0.0005972

alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
  0.1526 0.5050
sample estimates:
    cor
 0.3407
```

The `use='pairwise'` in the above tells R to omit observations with missing data and use complete pairs of observations. The first function confirms our impressions from the graphs: the correlations between genome size and morphology are positive but comparatively weak and the correlations between morphological measurements are positive and very strong (i.e. close to 1). The correlation test tells us that genome size and body length are positively correlated (r=0.34, t = 3.5507, df = 96, p = 0.0006).

⋆ Again, remember this example when reporting correlations!

5.3 Transformations and allometric scaling

There is one problem with the correlations above: *correlations assume a straight line relationship*. Some of the scatterplots above are fairly straight but there are some strongly curved relationships. This is due to the allometric scaling mentioned in Chapter 4: two of the variables are in linear units (total and forewing length), one is in squared units (forewing area) and one in cubic units (body weight, which is approximately volume).

The relationships between these variables can be described using a power law: $y = ax^b$. Fortunately, if we log transform this equation, we get $\log(y) = \log(a) + b\log(x)$. This is the equation of a straight line ($y = a + bx$), so we should be able to make these plots straighter by logging both axes. We saw in Chapter 4 that we can create a new logged variable in the data frame like this:

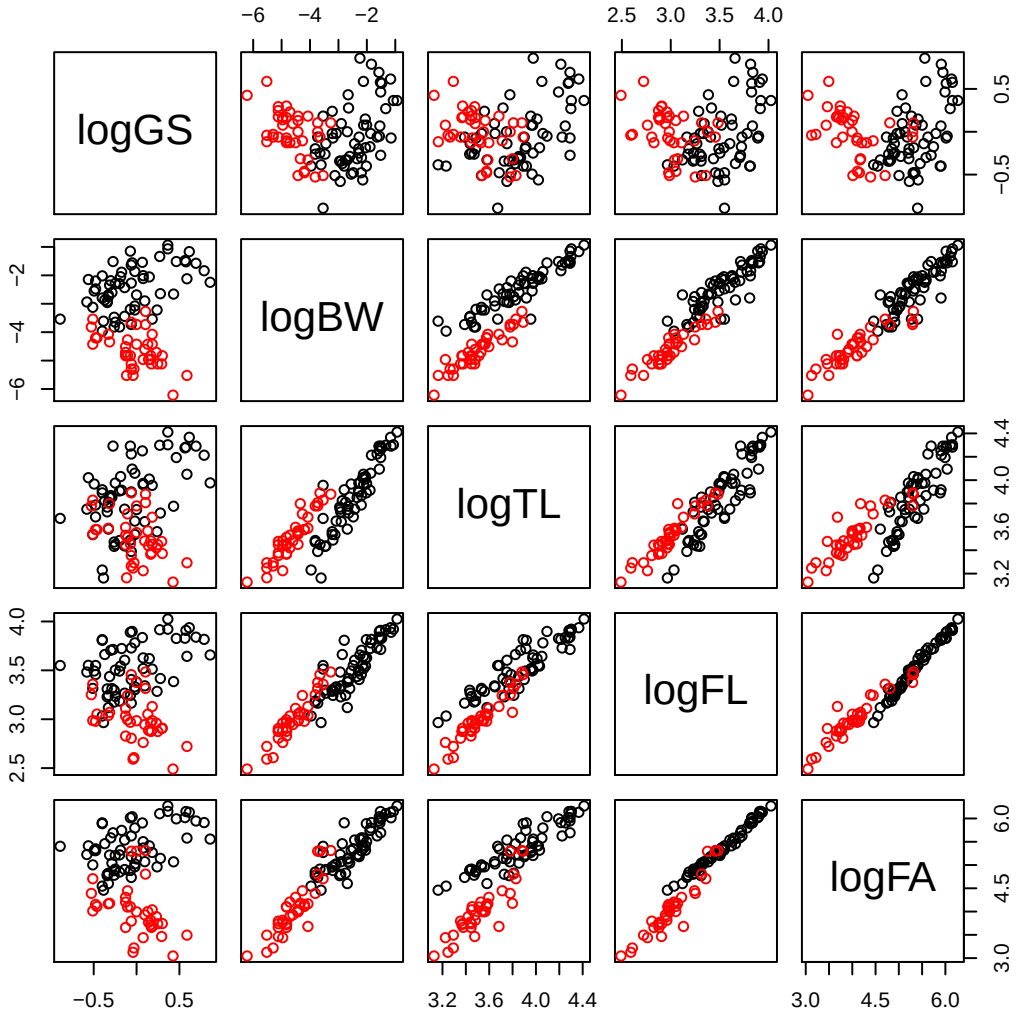
```
> genome$logGS <- log(genome$GenomeSize)
```

⋆ Using this line as a template, create a new logged version of the five variables listed above.
⋆ Using `str`, work out which column numbers the logged variables are and create a new variable called `logmorpho` containing these numbers.
⋆ Copy the `pairs` and `cor` test from earlier in your script and modify them to run these functions for the columns given in `logmorpho`.

The correlations should give the following output:

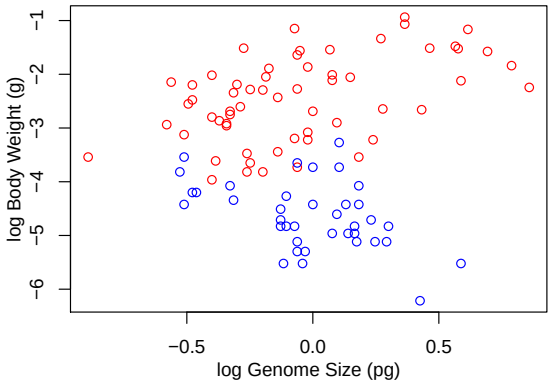
```
> cor(genome[, logmorpho], use = "pairwise")

         logGS   logBW  logTL  logFL   logFA
logGS 1.00000 0.08406 0.2224 0.1150 0.06808
logBW 0.08406 1.00000 0.8892 0.9456 0.94996
logTL 0.22244 0.88919 1.0000 0.9158 0.86207
logFL 0.11500 0.94565 0.9158 1.0000 0.97916
logFA 0.06808 0.94996 0.8621 0.9792 1.00000
```

The scatterplots should look like this and show that logging the data has very successfully removed allometric scaling effects in the data:



5.4 Regression

We'll now look at fitting the first linear model of this course, to explore whether log genome size explain log body weight. The first thing to do is to plot the data:



It is clear that the two suborders have very different relationships: to begin with we will look at dragonflies (Anisoptera). We will calculate two linear models:

The null model This is the simplest linear model: nothing is going on and the response variable just has variation around the mean: $y = \beta_1$. This is written as an R formula as `y ~ 1`.

Linear regression This models a straight line relationship between the response variable and a continuous explanatory variable: $y = \beta_1 + \beta_2 x$.

The code below fits these two models.

```
> nullModelDragon <- lm(logBW ~ 1, data = genome, subset = Suborder ==
"Anisoptera")
> genomeSizeModelDragon <- lm(logBW ~ logGS, data = genome, subset =
Suborder == "Anisoptera")
```

- ⋆ Note the long names for the models. Short names are easier to type but calling R objects names like `mod1`, `mod2`, `xxx` swiftly get confusing!
- ⋆ Enter these models into your script and run them.

Now we want to look at the output of the model. Remember from the lecture that a model has *coefficients* (the β values in the equation of the model) and *terms* which are the explanatory variables in the model. We'll look at the *coefficients* first:

```
> summary(genomeSizeModelDragon)
 Call:
 lm(formula = logBW ~ logGS, data = genome, subset = Suborder ==
     "Anisoptera")

 Residuals:
    Min      1Q  Median      3Q     Max
 -1.324 -0.612   0.097   0.519   1.324

 Coefficients:
             Estimate Std. Error t value Pr(>|t|)
 (Intercept)  -2.3995     0.0908  -26.41  < 2e-16 ***
 logGS         1.0052     0.2398    4.19  9.5e-05 ***
 ---
 Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

 Residual standard error: 0.697 on 58 degrees of freedom
   (2 observations deleted due to missingness)
 Multiple R-squared: 0.233, Adjusted R-squared: 0.219
 F-statistic: 17.6 on 1 and 58 DF,  p-value: 9.54e-05
```

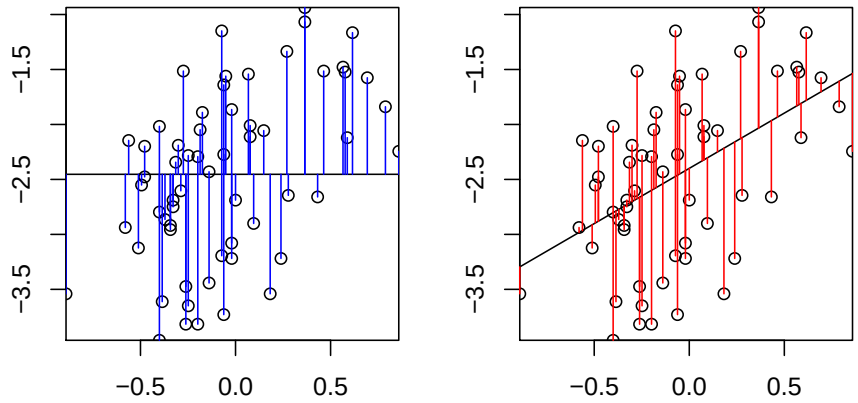
There is a lot of information there: the model description ('`Call`'), a summary of the residuals, a table of coefficients and then information on residual standard error, r squared and an F test. All of these will become clearer during this course — for the moment, concentrate on the coefficients table.

There are two rows in the coefficient table, one for each coefficient in $y = \beta_1 + \beta_2 x$ — these are the intercept and the slope of the line. The rest the details on each row are a t test of whether the slope and intercept are significantly different from zero.

Now we will look at the *terms* of the model using the `anova` function. We will have a proper look at ANOVA (Analysis of Variance) in the next chapter. Meanwhile, for our current purposes, all you need to know is that ANOVA tests how much variation in the response variable is explained by each explanatory variable. We only have one variable and so there is only one row in the output:

```
> anova(genomeSizeModelDragon)

 Analysis of Variance Table

 Response: logBW
           Df Sum Sq Mean Sq F value  Pr(>F)
 logGS      1   8.53    8.53    17.6 9.5e-05 ***
 Residuals 58  28.14    0.49
 ---
 Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

This table is comparing the variation in log body weight explained by log genome size to the total variation in log body weight. We are interested in how much smaller the residuals are for the genome size model than the null model. Graphically, how much shorter are the red residuals than the blue residuals:



We can get the sums of the squares of these residuals from the two models using the function `resid`, and then square them and add them up:

```
> sum(resid(nullModelDragon) ^ 2)
 [1] 36.67

> sum(resid(genomeSizeModelDragon) ^ 2)
 [1] 28.14
```

So we have five columns in the table:

Df This shows the degrees of freedom. Each variable takes up degrees of freedom and the ones left over are the residuals. In this case, genome size adds a slope (compare the null model $y = \beta_1$ and this model $y = \beta_1 + \beta_2 x$ — there is one more β).

Sum Sq This shows sums of squares. The bottom line is the residual sum of squares for the model and the one above is the variation explained by genome size. Using the two values from above, the sums of square residuals for the null model are 36.67. In the genome size model, the sum of square residuals are 28.14 and so $36.67 - 28.14 = 8.53$ units of variance have been explained by this model.

Mean Sq These are just the Sum Sq values divided by the degrees of freedom. The idea behind this is simple: if we explain lots of variation with one coefficient, that is good; if we explain a small amount of variation with lots of degree of freedom then that is bad.

F value This is the ratio of the Mean Sq for the variable and the residual Mean Sq. This is used to test whether the explained variation is large or small.

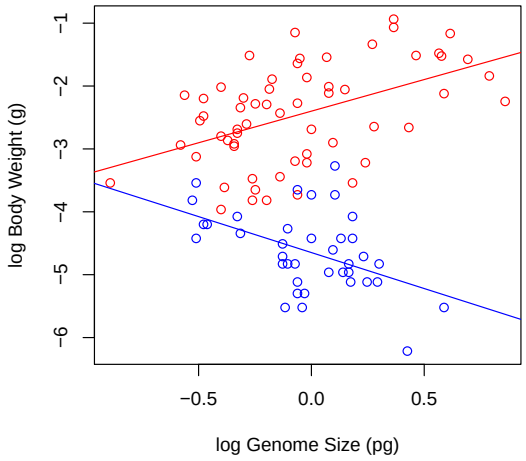


Figure 5.1: Linear regression models fitted to the body weight vs. genome size to the Dragonfly (red) and Damselfly (blue) subsets of the data.

Pr(>F) This is a p value — the probability of the variable explaining this much variance by chance.

In this case, it is clear that genome size explains a significant variation in body weight.

- ⋆ Include the `summary` and `anova` commands for `genomeSizeModelDragon` in your script, run them and check you are happy with the output.
- ⋆ Using this code as a template, create a new model called `genomeSizeModelDamsel` that fits log body weight as a function of log genome size for damselflies.
- ⋆ Write and run code to get the `summary` and `anova` tables for this new model.

5.5 Plotting the model

Now we can plot the data and add lines to show the models. For simple regression models, we can use the function `abline(modelName)` to add a line based on the model.

- ⋆ You already know how to create and customise scatterplots from previous chapters. Create a plot of log body weight as a function of log genome size, picking your favourite colours for the points.
- ⋆ Use `abline` to add a line for each model and use the `col` option in the function to colour each line to match the points. For example: `abline(genomeSizeModelDragon, col='red')`.

You should get something like Figure 5.1.

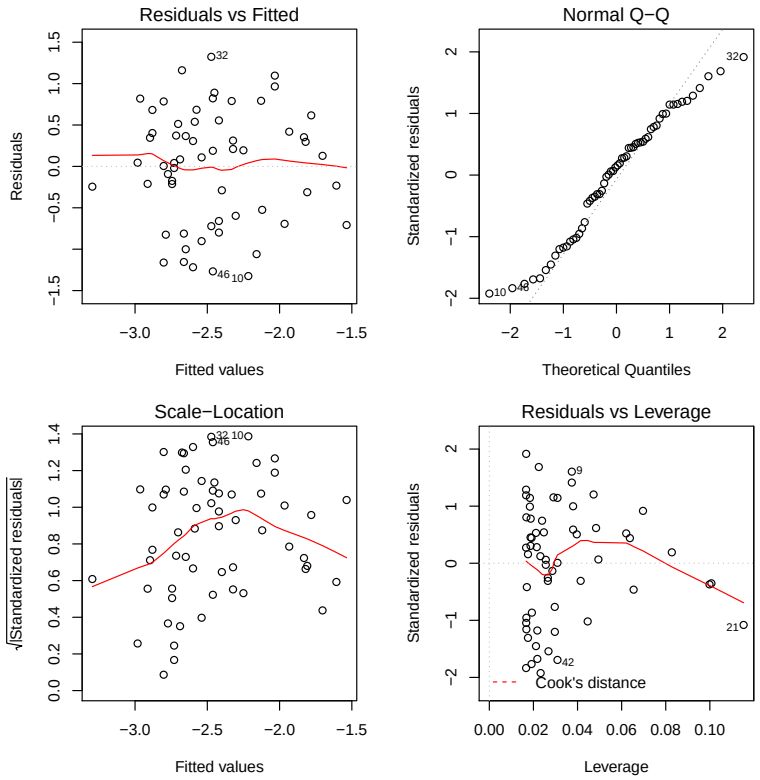


Figure 5.2: Diagnostics for the `lm` fit to the Dragonfly data subset.

5.6 Model diagnostics

Now that we have our models, we need to check that they are appropriate for the data. For this, we will inspect "diagnostic plots". Producing diagnostic plots is easy in R — if you `plot` a model, then R produces a set of diagnostic plots!

⋆ Try the following code (and include in the R script file):

```
> par(mfrow = c(2, 2), mar = c(5, 5, 1.5, 1.5))
> plot(genomeSizeModelDragon)
```

This should give the plots shown in figure 5.2. And,

```
> par(mfrow = c(2, 2), mar = c(5, 5, 1.5, 1.5))
> plot(genomeSizeModelDragon)
```

This should give the plots shown in figure 5.3.

The diagnostic plots are:

Residuals vs Fitted This plot is used to spot if the distribution of the residuals (the vertical distance from a point to the regression line) has *similar variance* for different predicted values (the y-value on the line corresponding to each x-value). There should be no obvious patterns (such as curves) or big gaps. If there was no scatter, if all the points fell exactly on the line, then all of the dots on this plot would lie on the gray horizontal dashed line. The red line is

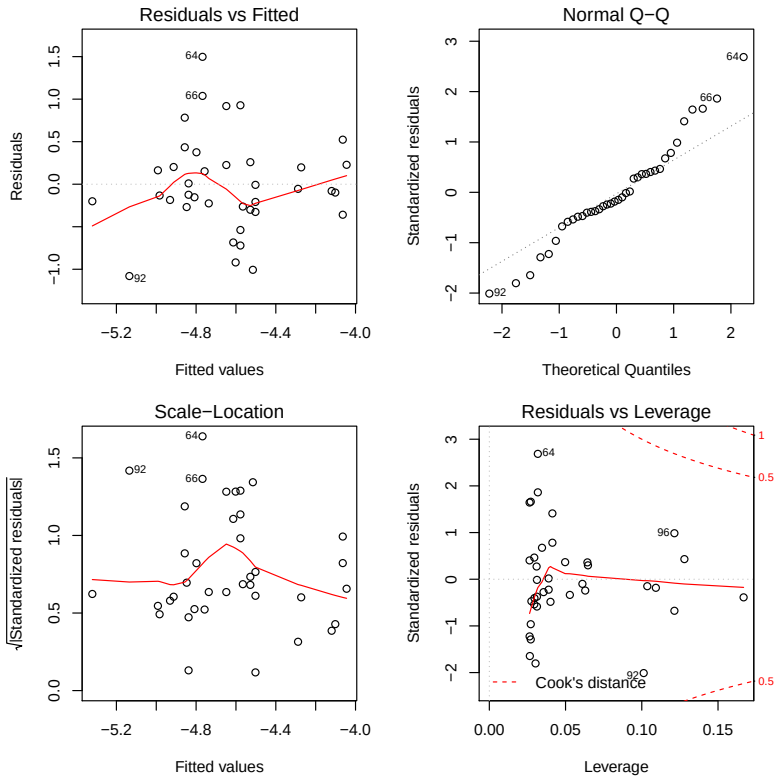


Figure 5.3: Diagnostics for the `lm` fit to the Damselfly data subset.

a smoothed curve to make it easier to see trends in the residuals. It is flat in the Dragonfly model fit (Figure 5.2), and a bit more wavy than we would like in the in the Damselfly model fit (Figure 5.3), but there are no clear trends in either, which is what you hope to see.

Normal Q–Q This plot is to check whether the residuals are *normally distributed* — are the values of the observed residuals similar to those expected under a normal distribution? Ideally, the points should form a perfectly straight line, indicating that the observed residuals exactly match the expected. Here, note that the points lie pretty close to the dashed line in both Figures 5.2 & 5.3, but deviate at the ends, especially for Damselflies. However, some deviation is to be expected near the ends — here these deviations are just about acceptable.

Scale–Location The x-axis on this plot is identical to the Residuals vs Fitted plot – these are the fitted values. The y-axis is the square root of the *standardized residuals*, which are residuals rescaled so that they have a mean of zero and a variance of one. As a result, all y-axis values are positive. Thus large residuals (both positive and negative) plot at the top, and small residuals plot at the bottom (so only their *scale* is retained). Thus, all of the numbered points (which will be the same in all plots) plot at the top here. The red line here shows the trend, just like the Residuals vs Fitted plot. The regression analysis has assumed homoscedasticity, that the variance in the residuals doesn't change as a function of the predictor. If that assumption is correct, the red line should be relatively flat. It is not quite as flat as we would like, especially for the Dragonfly analysis (Figure 5.2).

Residuals vs Leverage This plot shows the standardized residuals against leverage. "Leverage" is a measure of how much each data point influences the linear model's coefficient estimates. Because the regression line must pass through the centroid ("pivot point") of the data (Figure 5.4), points that lie far from the centroid have greater leverage, and their leverage increases if there are fewer points nearby. There are two key things to note about this plot:

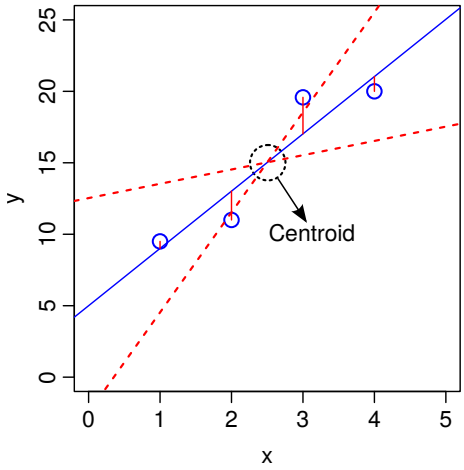


Figure 5.4: Leverage of data points on slope of a regression. The points further away from the centroid in the x-axis direction have more leverage, and can therefore move the regression line up or down (dashed red lines).

1. The standardized residuals (y-axis) are centered around zero and reach 2-3 standard deviations away from zero. They should also lie symmetrically about zero, as would be expected for a normal distribution. This is the case for the Damselfly plot (Figure 5.3) , but not so much for the Dragonfly plot Figure 5.2.
2. The contours values show *Cook's distance* (only visible in the Damsefly plot), which measures how much the regression would change if a point was deleted. Cook's distance is increased by leverage and by large residuals: a point far from the centroid with a large residual can severely distort the coefficient estimates from the regression. On this plot, you want to see that the red smoothed line stays close to the horizontal gray dashed line and that no points have a large Cook's distance (i.e, >0.5). Both are true here.

This is an important diagnostic plot in regression analyses in particular because it tells you whether your estimate of the slope coefficient in particular is strongly affected by certain data points.

Note that certain points are numbered in all the plots — these are points to pay special attention to because they are *potential* outliers. The numbers correspnd to the row number for that dataset in your data frame. You can easily identify these points in your data plot (Figure 5.1) because the order of the points along the fitted values axis (y-axis) in the diagnostic plot matches the order along the x-axis in the data plot. So, fo example here, in Figure 5.2, the two numbered points (46, 10) near the bottom correspond in the data plot (Figure 5.1) to the two red points near the center-left that lie farthest below the red line.

Thus, neither the Drangonfly nor the Damselfly diagnostic plots look perfect, but this level of deviation from assumptions of linear models is acceptable. The main worrying factors are that the QQ plot for Damselflies indicates the observed residuals are a bit more extreme than expected, and the Scale–Location plot for Dragonflies suggests some pattern in the standardized residuals wrt location of the fitted values.

⋆ Copy the code to create the diagnostic plots into your script to keep a record of the code and run it.

5.7 Reporting the model

Now we know that the models are appropriate and we have a plot, the last thing is to report the statistics. For the damselfly model, here is one summary that would do: log genome size explains significant variation in log body weight in dameselflies (F=10.5, df=1,32, p=0.0025) and shows that body weight decreases with genome size (intercept: -4.65, se=0.09; slope: -1.14, se=0.35).

Chapter 6

Linear Models: Analysis of variance

Aims of this chapter:

- Plotting boxplots and barplots using factors
- Fitting factors in linear models using analysis of variance
- Diagnostic plots for explanatory factors
- Exploring differences between levels of a factor

6.1 What is ANOVA?

A *One-way analysis of variance* (one-way ANOVA) is a technique used to compare means of three or more samples (using the F distribution — recall the lecture and Chapter 4). It can be used only for numerical data.

ANOVA tests the null hypothesis that samples in two or more groups are drawn from populations with the same mean values. To do this, ANOVA uses the F-statistic — the ratio of the variance calculated among the means (the null hypothesis) to the variance within the samples. If the group means are drawn from populations with the same mean values, the variance between the group means should be lower than the variance of the samples. A higher ratio (and F value) therefore implies that the samples were drawn from populations with different mean values.

This is same as asking whether a linear model with a predictor (or explanatory variable) with at least two categorical levels (or factors), better accounts for the variance (Explained Sum of Squares, ESS) than a null model of the form $y = \beta_1$ (Figure 6.1).

Thus, hopefully you see why ANOVA is a type of linear modelling technique. And by extension, you should note that linear regression models of the form $y = \beta_1 + \beta_2 x$ (where x is a continuous predictor variable) basically require ANOVA to determine if the the model better fits than a null model of the form $y = \beta_1$.

Typically, one-way ANOVA is used to test for differences among at least three groups, since the two-group (or levels or factors) case can be covered by a t-test (see Chapter 4). When there are only two means to compare, the t-test and the F-test are equivalent; the relation between ANOVA and t is given by $F = t^2$.

An extension of one-way ANOVA is two-way analysis of variance that examines the influence of two different categorical independent variables on one dependent variable — we will look at

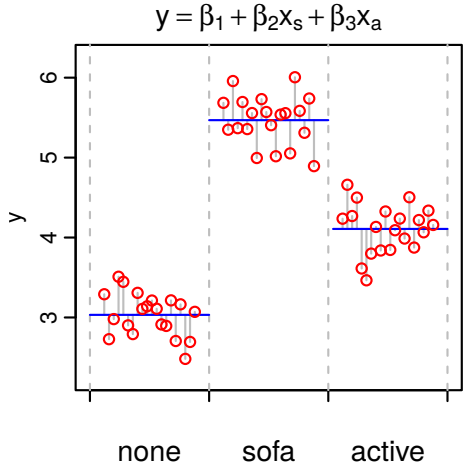


Figure 6.1: A dataset where an ANOVA would be appropriate. performing an ANOVA test on this dataset is the same as fitting the linear model $y = \beta_1 + \beta_2 x_s + \beta_3 x_a$, where x_s and x_a are two levels. There are three "treatments" here with the first treatment, the control, captured by the baseline value β_1 (the sample with the lowest value, on the far left)

multiple predictor variables in Chapter 7 onwards.

6.2 A new dataset

In this Chapter, we will use a new dataset of genome size and life history in mammals to try out a one-way ANOVA. The dataset is a composite of data taken from an online database of genome sizes and a published database of mammalian life history:

Genome size Average genome sizes for available mammal species are taken from the online database www.genomesize.com.

Life history Trait data for these species are taken from: Jones, K. E. *et al.* (2009) PanTHERIA: a species-level database of life history, ecology, and geography of extant and recently extinct mammals. Ecology 90, 2648–2648. [Link].

- ⋆ Download the file `MammalData.csv` from bitbucket and save to your `Data` directory.
- ⋆ Create a new blank script called `ANOVA_Prac.R` and add some introductory comments.
- ⋆ Use `read.csv` to load the data in the data frame `mammals` and then `str` and `summary` to examine the data.

There are nine variables. The first two are the latin binomial and taxonomic order of each species, followed by the species mean genome size ('C value', picograms), adult body mass (g), diet breadth, habitat breadth, litter size and then two factors showing whether the species are ground dwelling and their trophic level. For more information, see the link above.

You will see from the output of `summary` that there is lots of missing data for the life history traits.

6.3 Exploring the data with boxplots

We are interested in finding out whether the mean C value for species varies predictably for different levels of life history traits (a typical one-way ANOVA question). For example:

- Do carnivores or herbivores have larger genome sizes?
- Do ground dwelling mammals have larger or smaller genome sizes?

Before we fit any models, we want to plot the data to see if the means within these groupings look different. We also want to check whether the variance looks similar for each group: *constant normal variance*! A simple way is to look at box and whisker plots, showing the median and range of the data:

⋆ Use `plot(meanCvalue ~ TrophicLevel, data= mammals)` to generate a boxplot of the differences in genome sizes between trophic levels.
⋆ Looking at the plots, it is clear that there is more spread in the data above the median than below. Create a new variable `logCvalue` in the `mammals` data frame containing logged C values.
⋆ Create a boxplot of log C values within trophic groups.
⋆ Repeat the two plot commands to look at differences between ground dwelling and other species.

6.4 Differences in means with barplots

Box and whisker plots show the median and spread in the data very clearly, but we want to test whether the means are different. This is *t* test territory — how different are the means given the standard error — so it is common to use barplots and standard error bars to show these differences.

We're going to use some R code to construct a barplot by hand. We need to calculate the means and standard errors within trophic groups, but before we can do that, we need a new functions to calculate the standard error of a mean:

```
# get standard error of the mean from a set of values (x)

seMean <- function(x){
    x <- na.omit(x) # get rid of missing values

    se <- sqrt(var(x)/length(x)) # calculate the standard error

    return(se)  # tell the function to return the standard error
}
```

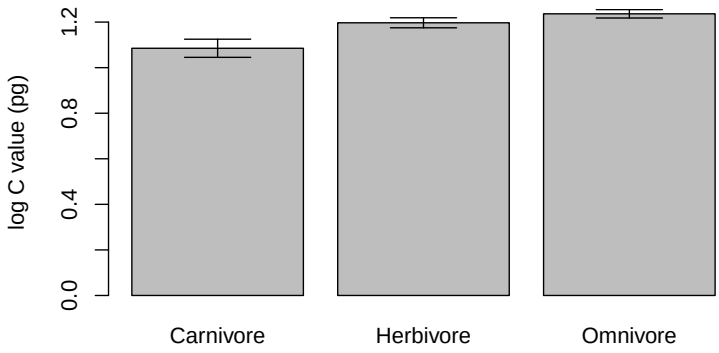
Now we can use the function `tapply`: it splits a variable up into groups from a factor and calculates statistics on each group using a function.

```
trophMeans <- tapply(mammals$logCvalue, mammals$TrophicLevel, FUN =
mean, na.rm = TRUE)

print(trophMeans)

 Carnivore Herbivore  Omnivore
     1.085     1.197     1.236
```

```
trophSE <- tapply(mammals$logCvalue, mammals$TrophicLevel, FUN = seMean)

print(trophSE)

 Carnivore Herbivore  Omnivore
   0.03983   0.02206   0.01844
```

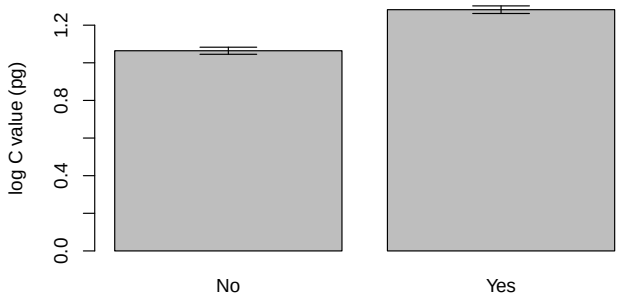
Now we have to put these values together on the plot:

```
# get the upper and lower limits of the error bars
upperSE <- trophMeans + trophSE
lowerSE <- trophMeans - trophSE

# get a barplot
# - this function can report where the middle of the bars are on the x-axis
# - set the y axis limits to contain the error bars

barMids <- barplot(trophMeans, ylim=c(0, max(upperSE)), ylab = 'log C value (pg)')

# Now use the function to add error bars
# - draws arrows between the points (x0,y0) and (x1,y1)
# - arrow heads at each end (code=3) and at 90 degree angles

arrows(barMids, upperSE, barMids, lowerSE, ang=90, code=3)
```



Now we need to draw all these pieces together into a script and get used to using them.

- ⋆ Copy all the lines of code from this section into your script.
- ⋆ Run it and check you get the graph above.
- ⋆ Use the second two chunks as a model to plot a similar graph for `GroundDwelling`. You should get something like the plot below.



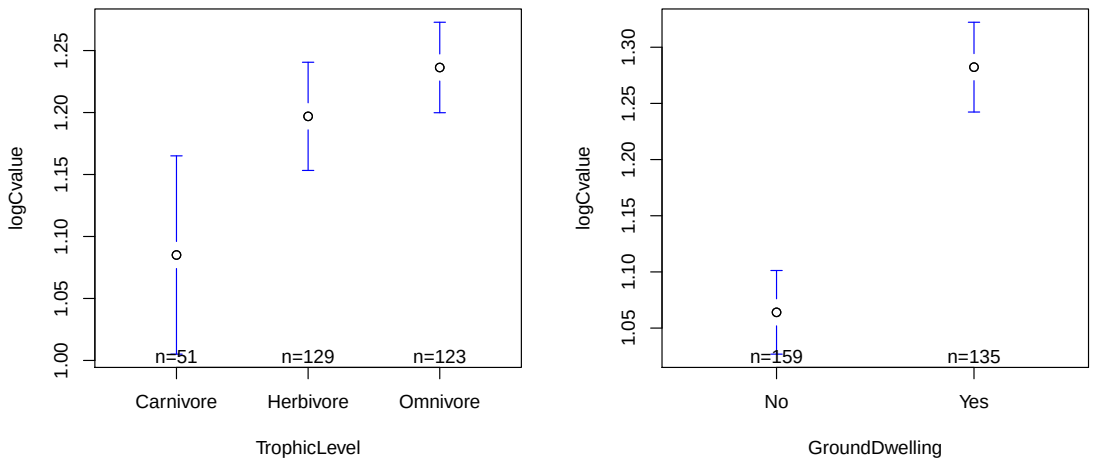
6.5 An alternative to barplots

That is a lot of work to go through for a plot. Doing it the hard way uses some useful tricks, but one strength of R is that there is a huge list of add-on packages that you can use to get new functions that other people have written.

We will use the `gplots` package to create plots of group means and confidence intervals. Rather than plotting the means $\pm$ 1 standard error, the option `p=0.95` uses the standard error and the number of data points to get 95% confidence intervals. The default `connect=TRUE` option adds a line connecting the means, which isn't useful here.

⋆ Replicate the code below into your script and run it to get the plots below.

```
#Load the gplots package
> library(gplots)

# Get plots of group means and standard errors
> par(mfrow=c(1,2))
> plotmeans(logCvalue ~ TrophicLevel, data=mammals, p=0.95, connect=FALSE)
> plotmeans(logCvalue ~ GroundDwelling, data=mammals, p=0.95, connect=FALSE)
```



6.6 Analysis of variance

Hopefully, those plots should convince you that there are differences in genome size between different trophic groups and between ground dwelling and other mammals. We'll now use a linear model to test whether those differences are significant.

- ⋆ Using your code from Chapter 5 as a guide, create a linear model called `trophicLM` which models log C value as a function of trophic group.
- ⋆ Use `anova` and `summary` to look at the analysis of variance table and then the coefficients of the model.

The ANOVA table for the model should look like the one below: trophic level explains highly significant variation in genome size ($F = 7.22, \text{df} = 2 \text{ and } 300, p = 0.0009$). *Note the style of reporting the result* - the statistic (F), degrees of freedom and p value are all provided in support. It is common to contract this style to this: $F_{2,300} = 7.22, p = 0.0009$.

```
> anova(trophicLM)

 Analysis of Variance Table

 Response: logCvalue
               Df Sum Sq Mean Sq F value  Pr(>F)
 TrophicLevel   2   0.83   0.413    7.22 0.00087 ***
 Residuals    300  17.18   0.057
 ---
 Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

However, look at the sum of squares column. Of a total of $17.18 + 0.83 = 18.01$ units of sums of squares, only 0.83 are explained by trophic level: $0.83/18.01 \approx 0.046$ or 4.6%. This ratio is called r^2, a measure of explanatory power, and shows that, although the model is very significant, it isn't very explanatory. We care about explanatory power or effect size, *not* p values.

The coefficients table for the model looks like this:

```
> summary(trophicLM)

 Call:
 lm(formula = logCvalue ~ TrophicLevel, data = mammals)

 Residuals:
     Min      1Q  Median      3Q     Max
 -0.5038 -0.1635 -0.0038  0.1511  0.9313

 Coefficients:
                      Estimate Std. Error t value Pr(>|t|)
 (Intercept)            1.0851     0.0335   32.38  < 2e-16 ***
 TrophicLevelHerbivore  0.1119     0.0396    2.83  0.00503 **
 TrophicLevelOmnivore   0.1513     0.0399    3.80  0.00018 ***
 ---
 Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

 Residual standard error: 0.239 on 300 degrees of freedom
   (76 observations deleted due to missingness)
 Multiple R-squared: 0.0459,    Adjusted R-squared: 0.0396
 F-statistic: 7.22 on 2 and 300 DF,  p-value: 0.000866
```

It shows the following:

- The reference level (or intercept) is for carnivores. Their mean genome size is significantly different from zero - this is not an exciting finding!
- The mean genome size for both herbivores and omnivores are both significantly different from carnivores. Both larger in fact: herbivore mean genome size = $1.085 + 0.112 = 1.197$ and omnivore mean genome size = $1.085 + 0.151 = 1.236$. These are the same group means we found above.
- The r^2 is shown and is the 4.6% we calculated above. The *adjusted* r^2 reduces the raw r^2 to account for the number of variables included in the model. That 4.6% would be even less impressive if we needed 6 explanatory variables to get it...
- The F statistic, as in the ANOVA table above.

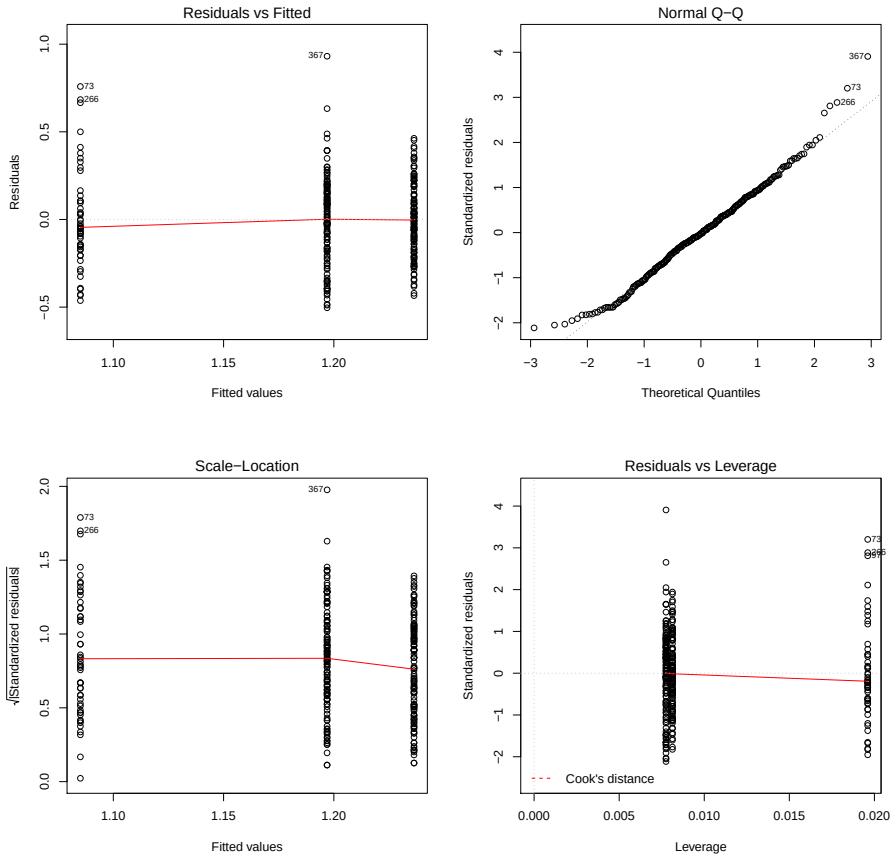
⋆ Repeat the analysis of variance above to look at the effects of ground dwelling on genome size.

6.7 Model criticism

The next question must be — and actually, we should do this before we go anywhere near the model summaries — is the model appropriate to the data.

⋆ Using chapter 5 to guide you, get the four model diagnostic plots for the trophic level model on a single figure.

The four plots are:



Note that in regression, the predicted (or fitted) values from the model take a range along the relationship $y = a + bx$ (as we saw in the Figures 5.2 & 5.3). For ANOVA, there are only a few predicted values — one for each group mean. This means that the plots above look different but we are looking for the same things: is there constant variance at each fitted value and are the residuals normally distributed? The answer for this model looks to be yes.

⋆ Check the ground dwelling model in the same way.

6.8 Testing pairwise differences between levels

The one thing that the trophic level model does not tell us is whether there is a difference in genome size between omnivores and herbivores — both are compared to carnivores, but not to each other. This is because of the multiple pairwise testing problem mentioned in Chapter 4 — if you do lots of tests then you may find small p values by chance and say something important is going on when it is just random chance. This is called a false positive or Type I error.

With a 95% confidence interval, there is a 5% chance of a false positive *per test* but there are ways of getting a 5% chance across a set (or family) of tests. For linear models, we can use Tukey's Honest Significant Difference test. We have to convert the `lm` object into an `aov` object first.

```
> TukeyTroph <- TukeyHSD(aov(trophicLM))
> print(TukeyTroph)

   Tukey multiple comparisons of means
     95% family-wise confidence level

 Fit: aov(formula = trophicLM)

 $TrophicLevel
                         diff       lwr    upr  p adj
 Herbivore-Carnivore 0.11186  0.01863 0.2051 0.0139
 Omnivore-Carnivore  0.15128  0.05741 0.2452 0.0005
 Omnivore-Herbivore  0.03942 -0.03161 0.1104 0.3923
```

The table shows the following:

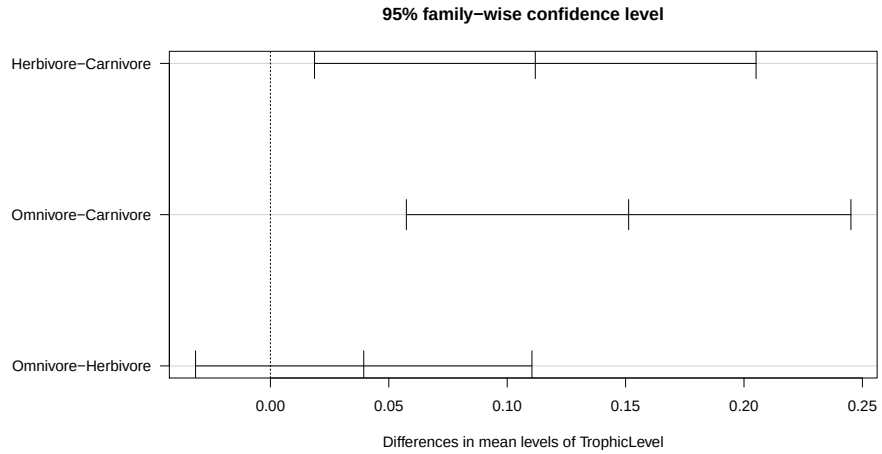
- The differences between the three possible pairs and then the lower and upper bounds of the 95% confidence interval for the difference and a p value.
- In each case, we want to know if the difference could be zero: does the 95% confidence interval for each pair include zero.
- For the first two pairs, carnivores versus omnivores and herbivores, the confidence intervals do not include zero, so they are significantly different. For the comparison between herbivores and omnivores, the interval does include zero (difference = 0.039, 95% CI's limits are -0.032 & 0.110), so these groups are not significantly different.
- The p values for the top two pairs are both larger (less significant) than in the summary table. The test has made it harder to find significant results.

You can visualise these confidence intervals by plotting the Tukey test. You have to tweak the graphics parameters to get a clean plot though.

```
> par(las=1, mar=c(4,10,3,1))
# las= 1 turns labels horizontal
# mar makes the left margin wider (bottom, left, top, right)
```

```
> plot(TukeyTroph)
```

The result should be:



⋆ Run the Tukey test in your script for both the trophic level and ground dwelling models.

6.9 Are the factors independent?

We've looked at two models, using trophic level and ground dwelling. It is worth asking whether these are independent factors. What if, for example, our herbivores are all big, ground dwellers? This is important to know because otherwise, a two-way ANOVA would not be appropriate. We will look at interactions in Chapter 8.

OK, so we want to know whether the two factors are independent. This is a job for the χ^2 test!

6.9.1 The Ch-square test and count data

The Chi-square test, also known as χ^2 test or chi-square test, is designed for scenarios where you want to statistically test how likely it is that an observed distribution of values is due to chance. It is also called a "goodness of fit" statistic, because it measures how well the observed distribution of data fits with the distribution that is expected if the variables of which measurements are made are independent. In our mammals example below, the two variables are trophic level and ground dwelling.

Note that a Chi-square test is designed to analyze categorical data. That is the data have been counted (count data) and divided into categories. It is not meant for continuous data (such as body weight, genome size, or height). For example, if you want to test whether attending class influences how students perform on an exam, using test scores (from 0-100) as data would not be appropriate for a Chi-square test. However, arranging students into the categories "Pass" and "Fail" and counting up how many fal in each categories would be appropriate. Additionally, the data in a Chi-square table (see below) should not be in the form of percentages – only count data are allowed!

6.9.2 The Ch-square test with the mammals data

We can easily build a table for the a Ch-square test on the mammals data as follows:

```
> factorTable <- table(mammals$GroundDwelling, mammals$TrophicLevel)
> print(factorTable)

      Carnivore Herbivore Omnivore
  No         26        45       64
  Yes        22        62       40
```

```
> chisq.test(factorTable)

    Pearson's Chi-squared test

 data:  factorTable
 X-squared = 8.12, df = 2, p-value = 0.01725
```

The `X-squared` value is the *test statistic*, akin to the t-value of the t-test or W value in the Wilcox test. Think about why df = 2 (how many levels are there?). The p-value is significant — we can conclude that the factors aren't independent. From the table, carnivores can be either ground dwelling or not, but herbivores tend to be ground dwelling and omnivores tend not to be. Ah well... it's OK. We will look at a better way to analyze these data in a subsequent Chapter.

⋆ Include and run the χ^2 test in your script.

6.10 Saving data

The last thing to do is to save a copy of the mammal data, including our new column of log data, for use in later chapters.

⋆ Use this code in your script to create the saved data in you `Data` directory :

```
save(mammals, file='../Data/mammals.Rdata')
```

Chapter 7

Linear Models: Multiple explanatory variables

Aims of this chapter:

- Including several explanatory variables in a model
- Interpreting summary tables for more complex models

7.1 Loading the data

⋆ Create a new blank script called `MulExpl.R` in your `Code` directory and add some introductory comments.
⋆ Use `load('../Data/mammals.Rdata')` to load the data saved at the end of Chapter 6. Look back at the end of Chapter 6 to see how you saved the RData file.
⋆ Use `ls` and `str` to check that the data has loaded correctly.

If `mammals.Rdata` is missing, just import the data again using `read.csv("../Data/MammalData.csv")`. You will then have to add the log C Value column to the imported data frame again.

The models we looked at in Chapter 6 explored whether the log genome size (C value, in picograms) of terrestrial mammals varied with trophic level and whether or not the species is ground dwelling. We will now look at a single model that includes both explanatory variables.

The first thing to do is look at the data again. In Chapter 5, we asked if carnivores or herbivores had larger genomes. Now we want to ask questions like: do ground-dwelling carnivores have larger genomes than arboreal or flying omnivores? We need to look at plots within groups.

Before we do that, there is a lot of missing data in the data frame and we should make sure that we are using the same data for our plots and models. We will subset the data down to the complete data for the three variables:

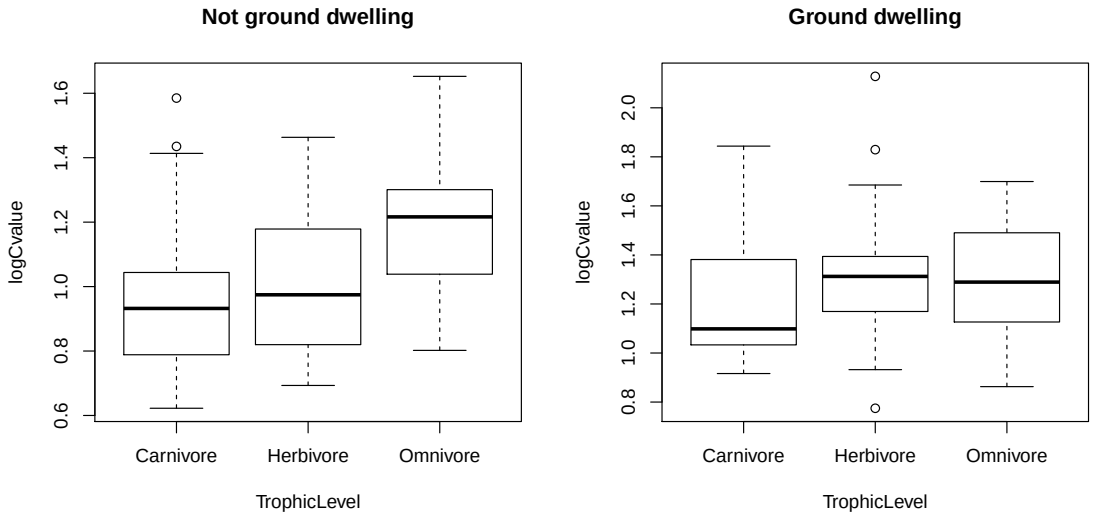
```
> mammals <- subset(mammals, select = c(GroundDwelling, TrophicLevel,
logCvalue))
> mammals <- na.omit(mammals)
> str(mammals)

'data.frame':   259 obs. of  3 variables:
 $GroundDwelling: Factor w/ 2 levels "No","Yes": 2 2 2 2 2 1 2 1 1 1 ...
```

```
$TrophicLevel  : Factor w/ 3 levels "Carnivore","Herbivore",..: 1 2 2 2 3 3 3 2 2 3 ↩
    ...
$logCvalue     : num  0.94 1.322 1.381 1.545 0.888 ...
 - attr(*, "na.action")=Class 'omit'  Named int [1:120] 2 4 7 9 10 11 14 15 20 21 ...
  .. ..- attr(*, "names")= chr [1:120] "2" "4" "7" "9" ...
```

7.2 Boxplots within groups

In Chapter 5, we used the `subset` option to fit a model just to dragonflies. You can use `subset` with plots too.

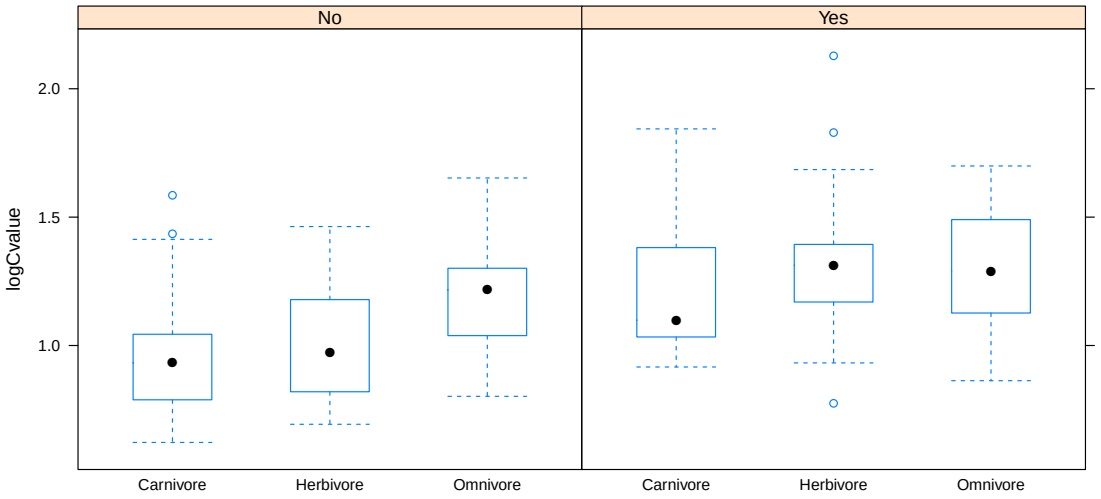
- ⋆ Add `par(mfrow=c(1,2))` to your script to split the graphics into two panels.
- ⋆ Copy the code from Chapter 6 to create a boxplot of genome size by trophic level into your script.
- ⋆ Using this, and adding a `subset` option to the code, generate the plots below.
- ⋆ You can use the option `main` to add titles to a plot.



7.3 `lattice` again

Recall that the `lattice` package provides some very neat extra ways to plot data in groups. They look pretty but the downside is that they don't use the same graphics system — all those `par` commands are useless for these graphs. The defaults look good though!

```
> library(lattice)
> bwplot(logCvalue ~ TrophicLevel | GroundDwelling, data= mammals)
```



The code `logCvalue ~ TrophicLevel | GroundDwelling` means plot the relationship between genome size and trophic level, but group within levels of ground dwelling. We are using the function `bwplot`, which is provided by `lattice` to create box and whisker plots.

- ⋆ Create the lattice plots above from within your script.
- ⋆ Rearrange this code to have three plots, showing the box and whisker plots for `GroundDwelling`, grouped within the levels of `TrophicLevel`.
- ⋆ Try reshaping the R plot window and running the command again. Lattice tries to make good use of the available space when creating lattice plots.

7.4 Barplots again

We're going to make the barplot code from Chapter 5 even more complicated! This time we want to know the mean log genome size within combinations of `TrophicLevel` and `GroundDwelling`. We can still use `tapply`, providing more than one grouping factor. We create a set of grouping factors like this:

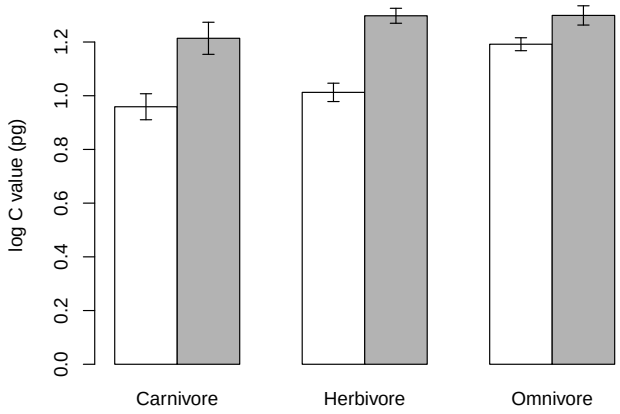
```
> groups <- list(mammals$GroundDwelling, mammals$TrophicLevel)
> groupMeans <- tapply(mammals$logCvalue, groups, FUN = mean)
> print(groupMeans)

     Carnivore Herbivore Omnivore
No      0.9589     1.012    1.192
Yes     1.2138     1.298    1.299
```

- ⋆ Copy this code into your script and run it.
- ⋆ Use this code and the script from Chapter 6 to get the set of standard errors for the groups(`groupSE`). You should get this:

```
     Carnivore Herbivore Omnivore
No     0.04842   0.03419  0.02410
Yes    0.05976   0.02787  0.03587
```

Now we can use `barplot`. The default option for a barplot of a table is to create a stacked barplot, which is not what we want. The option `beside=TRUE` makes the bars for each column appear side by side. Once again, we save the midpoints of the bars to add the error bars. The other options in the code below change the colours of the bars and the length of error bar caps.

```
    # get upper and lower standard error height
> upperSE <- groupMeans + groupSE
> lowerSE <- groupMeans - groupSE

# create barplot
> barMids <- barplot(groupMeans, ylim=c(0, max(upperSE)), beside=TRUE,
ylab= ' log C value (pg) ' , col=c( ' white ' , ' grey70 '))

> arrows(barMids, upperSE, barMids, lowerSE, ang=90, code=3, len=0.05)
```



⋆ Generate the barplot above and then edit your script to change the colours and error bar lengths to your taste.

7.5 Plotting means and confidence intervals

We'll use the `plotmeans` function again as an exercise to change graph settings and to prepare figures for write ups.

White space The default options in R use wide margins and spaced out axes and take up a lot of space that could be used for plotting data. You've already seen the `par` function and the options `mfrow` for multiple plots and `mar` to adjust margin size. The option `mgp` adjusts the placement of the axis label, tick labels and tick locations. See `?par` for help on the these options.

Main titles Adding large titles to graphs is also a bad idea — it uses lots of space to explain something that should be in the figure legend. With multiple plots in a figure, you have to label graphs so that the figure legend can refer to them. You can add labels using `text(x,y,'label')`.

Figure legends A figure legend should give a clear stand-alone description of the whole figure.

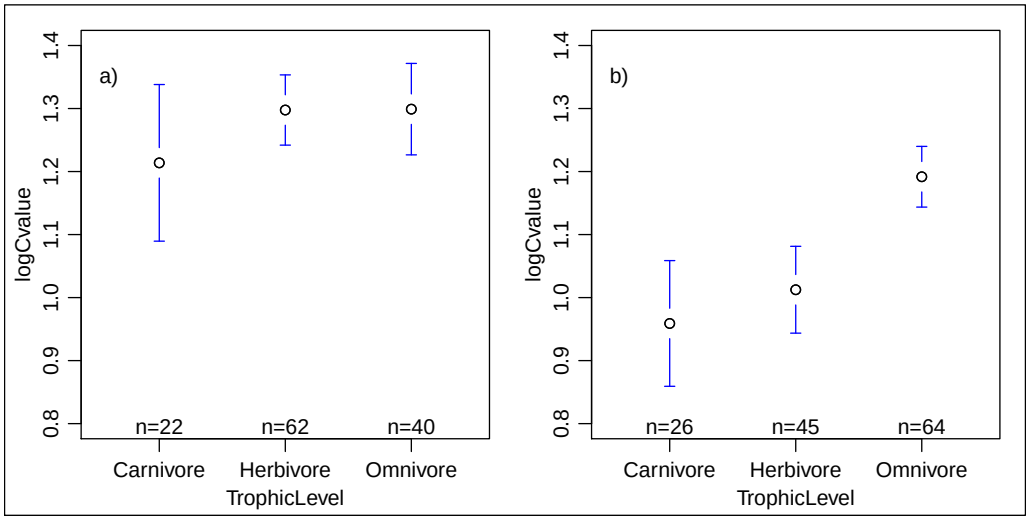


Figure 7.1: Means and 95% confidence intervals for log genome size (picograms) in mammals for different trophic levels for a) ground dwelling species and b) other species.

Referring to figures You *must* link from your text to your figures — a reader has to know which figures refer to which results. So: 'There are clear differences in mean genome size between species at different trophic levels and between ground dwelling and other species (Figure 7.1)'.

⋆ Use `plotmeans` from Chapter 6 and the `subset` option to generate the two plots below. You will need to set the `ylim` option for the two plots to make them use the same y axis.
⋆ Use `text` to add labels — the command `par('usr')` will show you the limits of the plot $(x_{min}, x_{max}, y_{min}, y_{max})$ and help pick a location for the labels.
⋆ Change the `par` settings in your code and redraw the plots to try and make better use of the space. In the example below, the box shows the edges of the R graphics window.

7.6 Multiple explanatory variables

All those graphs suggest:

- Carnivores have smaller genome size; omnivores have larger genome size.
- Herbivores are somewhere in between, but not consistently.
- All ground dwelling mammals typically have larger genome sizes.

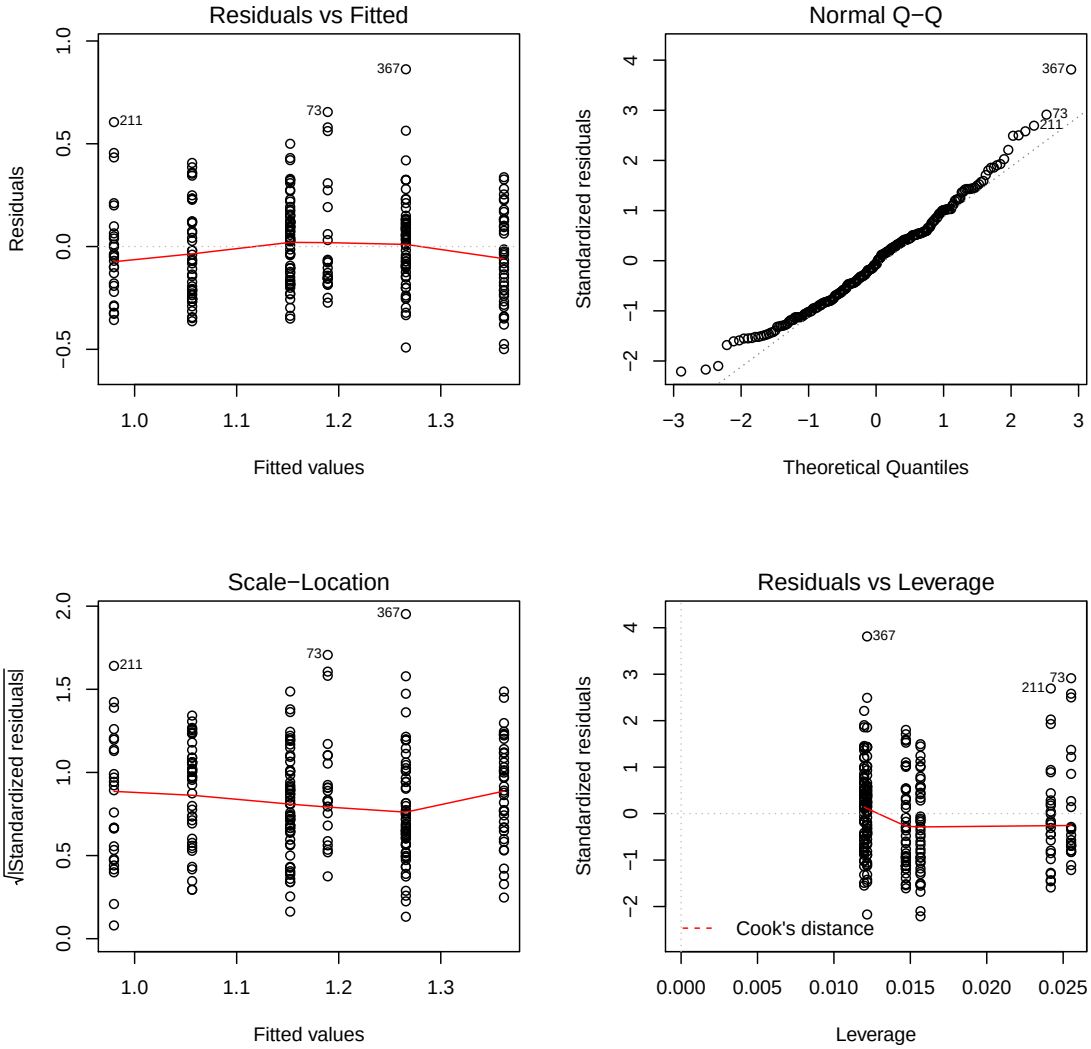
We suspected these things from the previous chapter but now we can see that they might have separate effects. We'll fit a linear model to explore this and add the two explanatory variables together.

⋆ This is an important section — read it through carefully and ask questions if you are unsure. Copy the code into your script and add comments. *Do not just jump to the next action item*!

```
> model <- lm(logCvalue ~ TrophicLevel + GroundDwelling, data = mammals)
```

We're going to do things right this time and check the model diagnostics before we rush into interpretation.

```
> par(mfrow=c(2,2))
> plot(model)
```



There are six predicted values now - three trophic levels for each of the two levels of ground dwelling. Those plots look ok so now we can look at the analysis of variance table:

```
> anova(model)

 Analysis of Variance Table

 Response: logCvalue
                 Df Sum Sq Mean Sq F value  Pr(>F)
 TrophicLevel     2   0.81   0.407    7.86 0.00049 ***
 GroundDwelling   1   2.75   2.747   53.04 4.1e-12 ***
 Residuals      255  13.21   0.052
 ---
 Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Ignore the p values! Yes, they're highly significant but we want to understand the model, not rubber stamp it with 'significant'.

- The sums of squares for the variables are both small compared to the residual sums of squares — there is lots of unexplained variation. We can calculate the r^2 as explained sums of squares over total sums of squares:

$$\frac{0.81+2.75}{0.81+2.75+13.21} = \frac{3.56}{16.77} = 0.212$$

- Trophic level explain much less variation than ground dwelling — this makes intuitive sense from the plots since there are big differences between Figure 7.1a and 7.1b, but small differences within.
- We could also calculate a significance for the whole model by merging the terms. The total explained sums of squares of $0.81+2.75 = 3.56$ uses $2+1 = 3$ degrees of freedom, so the mean sums of squares for all the terms together is $3.56/3 = 1.187$. Dividing this by the residual mean square of 0.052 gives an F of $1.187/0.052 = 22.83$.

Now we can look at the summary table to see the coefficients.

```
> summary(model)

 Call:
 lm(formula = logCvalue ~ TrophicLevel + GroundDwelling, data = mammals)

 Residuals:
     Min      1Q  Median      3Q     Max
 -0.4990 -0.1784 -0.0146  0.1250  0.8624

 Coefficients:
                     Estimate Std. Error t value Pr(>|t|)
 (Intercept)           0.9798     0.0354   27.68  < 2e-16 ***
 TrophicLevelHerbivore   0.0766     0.0397    1.93    0.055 .
 TrophicLevelOmnivore    0.1727     0.0398    4.34  2.0e-05 ***
 GroundDwellingYes       0.2095     0.0288    7.28  4.1e-12 ***
 ---
 Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

 Residual standard error: 0.228 on 255 degrees of freedom
 Multiple R-squared: 0.212, Adjusted R-squared: 0.203
 F-statistic: 22.9 on 3 and 255 DF,  p-value: 3.6e-13
```

Starting at the bottom, `summary` has again calculated r^2 for us and also an F statistic for the whole model, which matches the calculation above.

The other important bits are the four coefficients. The intercept is now the reference level for two variables: it is the mean for carnivores that are not ground dwelling. We then have differences from this value for being an omnivore or herbivore and for being ground dwelling. There is a big change in genome size associated with ground dwelling and omnivory and both of these have large effects sizes, each introducing about a 20% difference in genome size from the non-ground dwelling carnivores. In contrast, herbivory makes a small difference — about 8%. Because the difference is small and the standard error is large, the t value suggests that this difference might arise just be chance. Put another way, it isn't significant.

The table below shows how these four coefficients combine to give the predicted values for each of the group means.

	Carnivore	Herbivore	Omnivore
Not ground	$0.98 = 0.98$	$0.98+0.08 = 1.06$	$0.98+0.17 = 1.15$
Ground	$0.98+0.21 = 1.19$	$0.98+0.08+0.21 = 1.27$	$0.98+0.17+0.21 = 1.36$

7.7 Predicted values

Getting the model predictions by hand in this way is tedious and error prone. There is handy function called `predict` which uses the model directly to calculate values. The default is to give you the prediction for each point in the original data, but you can also ask for specific predictions.

The first thing to do is to set up a small data frame containing the explanatory values we want to use. The variable names and the level name have to match *exactly*, so we'll use the `levels` function to get the names. We want to look at all six combinations, so we'll use the `rep` function to set this up. The `each=2` option repeats each value twice in succession; the `times=3` options repeats the whole set of values three times.

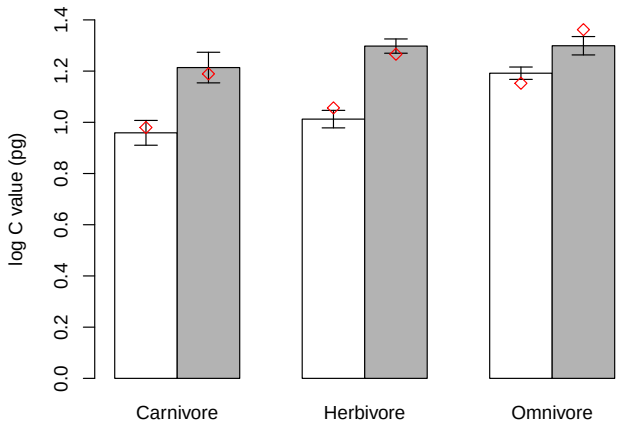
```
# data frame of combinations of variables
> gd <- rep(levels(mammals$GroundDwelling), times = 3)
> print(gd)

 [1] "No"  "Yes" "No"  "Yes" "No"  "Yes"

> tl <- rep(levels(mammals$TrophicLevel), each = 2)
> print(tl)

  [1] "Carnivore" "Carnivore" "Herbivore" "Herbivore" "Omnivore"  "Omnivore"

> predVals <- data.frame(GroundDwelling = gd, TrophicLevel = tl)
```

Now we have the data frame of values we want, we can use predict. As when we created log values, we can save the output back into a new column in the data frame.

```
> predVals$predict <- predict(model, newdata = predVals)
> print(predVals)

   GroundDwelling TrophicLevel predict
 1             No    Carnivore  0.9798
 2            Yes    Carnivore  1.1892
 3             No    Herbivore  1.0563
 4            Yes    Herbivore  1.2658
 5             No     Omnivore  1.1524
 6            Yes     Omnivore  1.3619
```

⋆ These are in the same order as the bars from your barplot. Make a copy of the barplot and arrows code and then add the code below to generate the plot.

```
> points(barMids, predVals$predict, col= ' red ' , pch=5)
```



The red points do not match to the calculated means. This is because the model only includes a single difference between ground and non-ground species — it has to be the same for each trophic group. The Chapter 8 will look at interactions, which allows these values to differ.

Chapter 8

Linear Models: Multiple variables and interactions

Aims of this chapter:

- Creating more complex models, including ANCOVA
- Looking at interactions between variables
- Plotting predictions from models

We will look at two models in this chapter:

1. Model 1: Is mammalian genome size predicted by interactions between trophic level and whether species are ground dwelling?
2. ANCOVA: Is body size in Odonata predicted by interactions between genome size and taxonomic suborder?

So far, we have only looked at the independent effects of variables. For example, in the trophic level and ground dwelling model from Chapter 7, we only looked for specific differences for being a omnivore *or* being ground dwelling, not for being specifically a *ground dwelling omnivore*. These independent effects of a variable are known as *main effects* and the effects of combinations of variables acting together are known as *interactions* — they describe how the variables *interact*.

8.1 Formulae with interactions in R

We've already seen a number of different model formulae in R. They all use this syntax:
`response variable ~ explanatory variable(s)`
but we are now going to add two extra pieces of syntax:

`y ~ a + b + a:b` The `a:b` means the interaction between `a` and `b` — do combinations of these variables lead to different outcomes?

`y ~ a * b` This a shorthand for the model above. The `*` means fit `a` and `b` as main effects and their interaction `a:b`.

8.2 Model 1: Mammalian genome size

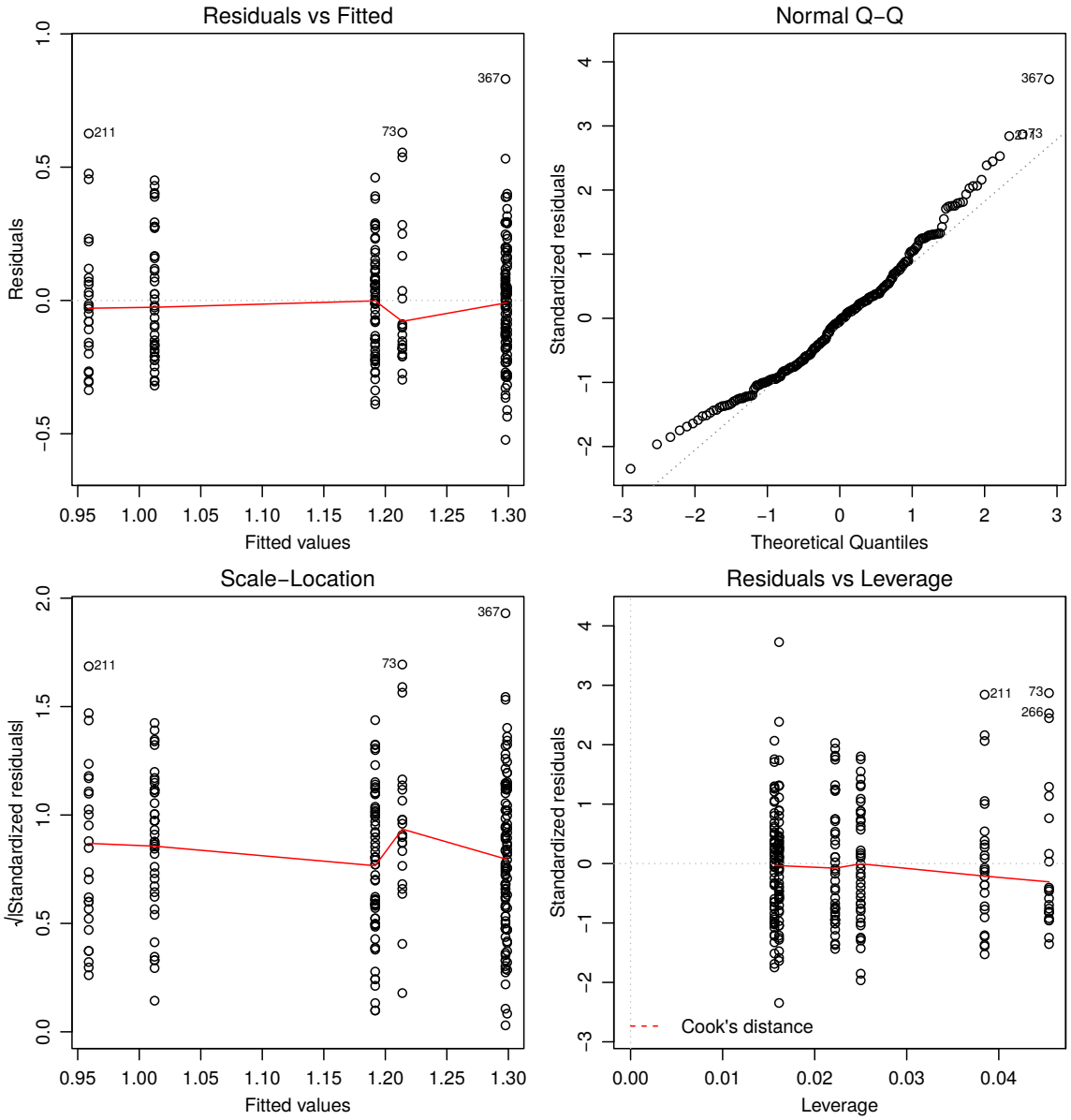
- ⋆ Make sure you have changed the working directory to `Code` in your stats coursework directory.
- ⋆ Create a new blank script called 'Interactions.R' and add some introductory comments.
- ⋆ Use `load('mammals.Rdata')` to load the data.

If `mammals.Rdata` is missing, just import the data again using `read.csv("../Data/MammalData.csv")`. You will then have to add the log C Value column to the imported data frame again.

Let's refit the model from Chapter 8.2, but including the interaction between trophic level and ground dwelling. We'll immediately check the model is appropriate:

```
> model <- lm(logCvalue ~ TrophicLevel * GroundDwelling, data= mammals)
> par(mfrow=c(2,2), mar=c(3,3,1,1), mgp=c(2, 0.8,0))
> plot(model)
```

This gives:



Now, we'll examine the `anova` and `summary` outputs for the model:

```
> anova(model)

 Analysis of Variance Table

 Response: logCvalue
                             Df Sum Sq Mean Sq F value  Pr(>F)
 TrophicLevel                 2   0.81   0.407    8.06  0.0004 ***
 GroundDwelling               1   2.75   2.747   54.40 2.3e-12 ***
 TrophicLevel:GroundDwelling  2   0.43   0.216    4.27  0.0150 *
 Residuals                  253  12.77   0.050
 ---
 Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Compared to the model from Chapter 7, there is an extra line at the bottom. The top two are the same and show that trophic level and ground dwelling both have independent main effects. The extra line shows that there is also an interaction between the two. It doesn't explain a huge amount of variation, about half as much as trophic level, but it is significant.

Again, we can calculate the r^2 for the model:

$$\frac{0.81+2.75+0.43}{0.81+2.75+0.43+12.77} = 0.238$$

The model from Chapter 7 without the interaction had an $r^2 = 0.212$ — our new model explains 2.6% more of the variation in the data.

The summary table is as follows:

```
>summary(model)

 Call:
 lm(formula = logCvalue ~ TrophicLevel * GroundDwelling, data = mammals)

 Residuals:
    Min      1Q  Median      3Q     Max
 -0.523 -0.171 -0.010   0.119   0.831

 Coefficients:
                                      Estimate Std. Error t value Pr(>|t|)
 (Intercept)                            0.9589     0.0441   21.76  < 2e-16 ***
 TrophicLevelHerbivore                  0.0535     0.0554    0.97  0.33460
 TrophicLevelOmnivore                   0.2328     0.0523    4.45  1.3e-05 ***
 GroundDwellingYes                      0.2549     0.0651    3.92  0.00012 ***
 TrophicLevelHerbivore:GroundDwellingYes  0.0303     0.0786    0.39  0.69979
 TrophicLevelOmnivore:GroundDwellingYes  -0.1476     0.0793   -1.86  0.06384 .
 ---
 Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

 Residual standard error: 0.225 on 253 degrees of freedom
   (120 observations deleted due to missingness)
 Multiple R-squared: 0.238, Adjusted R-squared: 0.223
 F-statistic: 15.8 on 5 and 253 DF,  p-value: 1.5e-13
```

The lines in this are:

- The reference level (intercept) for non ground dwelling carnivores. (The reference level is decided just by the alphabetic order of the levels)
- Two differences for being in different trophic levels.
- One difference for being ground dwelling
- Two new differences that give specific differences for ground dwelling herbivores and omnivores.

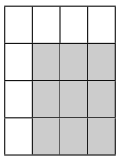
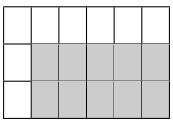
The first four lines, as in the model from Chapter 6, allow us to find the predicted values for each group *as long as the size of the differences don't vary between levels*. The last two lines then allow the size of differences to vary between levels.

The table below show how these combine to give the predictions for each group combination, with those two new lines show in red:

	Not ground	Ground
Carnivore	$0.96 = 0.96$	$0.96+0.25 = 1.21$
Herbivore	$0.96+0.05 = 1.01$	$0.96+0.05+0.25+0.03 = 1.29$
Omnivore	$0.96+0.23 = 1.19$	$0.96+0.23+0.25-0.15 = 1.29$

So why are there two new coefficients? For interactions between two factors, there are always $(n-1)\times(m-1)$ new coefficients, where n and m are the number of levels in the two factors. In this model, $(3-1)\times(2-1) = 2$. It is easier to understand why graphically: the prediction for

the white boxes below can be found by adding the main effects together but for the grey boxes we need to find specific differences and so there are $(n-1)\times(m-1)$ interaction coefficients to add.

$n=4, m=4$  $n=3, m=6$ 

If we put this together, what is the model telling us?

- Herbivores have the same genome sizes as carnivores, but omnivores have larger genomes.
- Ground dwelling mammals have larger genomes.
- These two findings suggest that ground dwelling omnivores should have extra big genomes. However, the interaction shows they are smaller than expected and are, in fact, similar to ground dwelling herbivores.

Note that although the interaction term in the `anova` output is significant, neither of the two coefficients in the `summary` has a $p < 0.05$. There are two weak differences (one very weak, one nearly significant) that together explain significant variance in the data.

⋆ Copy the code above into your script and run the model.
⋆ Make sure you understand the output!

Just to make sure the sums above are correct, we'll use the same code as in Chapter 7 to get R to calculate predictions for us:

```
# a data frame of combinations of variables
> gd <- rep(levels(mammals$GroundDwelling), times = 3)
> print(gd)
 [1] "No"  "Yes" "No"  "Yes" "No"  "Yes"

> tl <- rep(levels(mammals$TrophicLevel), each = 2)
> print(tl)

 [1] "Carnivore" "Carnivore" "Herbivore" "Herbivore" "Omnivore"  "Omnivore"

# New data frame
> predVals <- data.frame(GroundDwelling = gd, TrophicLevel = tl)

# predict using the new data frame
> predVals$predict <- predict(model, newdata = predVals)
> print(predVals)

   GroundDwelling TrophicLevel predict
 1             No    Carnivore  0.9589
 2            Yes    Carnivore  1.2138
 3             No    Herbivore  1.0125
 4            Yes    Herbivore  1.2977
 5             No     Omnivore  1.1918
 6            Yes     Omnivore  1.2990
```

⋆ Run these predictions in your script.

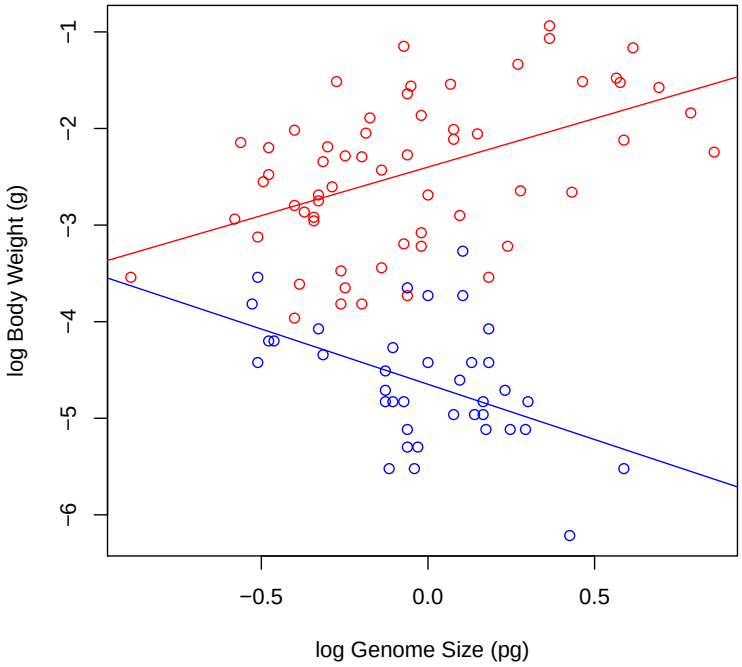
If we plot these data points onto the barplot from Chapter , they now lie exactly on the mean values, because we've allowed for interactions. The triangle on this plot shows the predictions for ground dwelling omnivores from the main effects ($0.96+0.23+0.25=1.44$), the interaction of -0.15 pushes the prediction back down.

8.3 ANCOVA: Body Weight in Odonata

We'll go all the way back to the regression analyses from Chapter 5. Remember that we fitted two separate regression lines to the data for damselflies and dragonflies. We'll now use an interaction to fit these in a single model. This kind of linear model — with a mixture of continuous variables and factors — is often called an *analysis of covariance*, or ANCOVA. That is, ANCOVA is a general linear model which blends ANOVA and regression. ANCOVA evaluates whether population means of a dependent variable are equal across levels of a categorical independent variable, while statistically controlling for the effects of other continuous variables that are not of primary interest, known as covariates.

- ⋆ Load the data: `odonata <- read.csv('../Data/GenomeSize.csv')`.
- ⋆ Create two new variables in the `odonata` data set called `logGS` and `logBW` containing log genome size and log body weight.

The models we fitted before looked like this:



We can now fit the model of body weight as a function of both genome size and suborder:

```
> odonModel <- lm(logBW ~ logGS * Suborder, data = odonata)
```

Again, we'll look at the `anova` table first:

```
> anova(odonModel)

 Analysis of Variance Table

 Response: logBW
               Df Sum Sq Mean Sq F value  Pr(>F)
 logGS          1    1.1     1.1    2.71     0.1
 Suborder       1  112.0   112.0  265.13 < 2e-16 ***
 logGS:Suborder 1    9.1     9.1   21.65 1.1e-05 ***
 Residuals     94   39.7     0.4
 ---
 Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Interpreting this gives the following:

- There is no significant main effect of log genome size. The *main* effect is the important thing here — genome size is hugely important but does very different things for the two different suborders. If we ignored `Suborder`, there isn't an overall relationship: the average of those two lines is pretty much flat.
- There is a very strong main effect of Suborder: the mean body weight in the two groups are very different.
- There is a strong interaction between suborder and genome size. This is an interaction between a factor and a continuous variable and shows that the *slopes* are different for the different factor levels.

The summary table looks like this:

```
> summary(odonModel)

 Call:
 lm(formula = logBW ~ logGS * Suborder, data = odonata)

 Residuals:
     Min      1Q  Median      3Q     Max
 -1.3243 -0.3225  0.0073  0.3962  1.4976

 Coefficients:
                         Estimate Std. Error t value Pr(>|t|)
 (Intercept)              -2.3995     0.0848  -28.31  < 2e-16 ***
 logGS                     1.0052     0.2237    4.49  2.0e-05 ***
 SuborderZygoptera        -2.2489     0.1354  -16.61  < 2e-16 ***
 logGS:SuborderZygoptera  -2.1492     0.4619   -4.65  1.1e-05 ***
 ---
 Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

 Residual standard error: 0.65 on 94 degrees of freedom
   (2 observations deleted due to missingness)
 Multiple R-squared: 0.755, Adjusted R-squared: 0.747
 F-statistic: 96.5 on 3 and 94 DF,  p-value: <2e-16
```

The first thing to note is that the r^2 value is really high. The model explains three quarters (0.752) of the variation in the data. Next, there are four coefficients:

- The intercept is for the first level of `Suborder`, which is Anisoptera (dragonflies).
- The next line, for log genome size, is the slope for Anisoptera.
- We then have a coefficient for the second level of `Suborder`, which is Zygoptera (damselflies). As with the first model, this difference in factor levels is a difference in mean values and shows the difference in the intercept for Zygoptera.
- The last line is the interaction between `Suborder` and `logGS`. This shows how the slope for Zygoptera differs from the slope for Anisoptera.

How do these hang together to give the two lines shown in the model? We can calculate these by hand:

$$\begin{aligned} \text{Body Weight} &= -2.40 + 1.01 \times \text{logGS} && \text{[Anisoptera]} \\ \text{Body Weight} &= (-2.40 - 2.25) + (1.01 - 2.15) \times \text{logGS} && \text{[Zygoptera]} \\ &= -4.65 - 1.14 \times \text{logGS} \end{aligned}$$

⋆ Add the code into your script and check that you understand the outputs.

We'll use the `predict` function to get the predicted values from the model and add lines to the plot above.

First, we'll create a set of numbers spanning the range of genome size:

```
#get the range of the data
> rng <- range(odonata$logGS)
#get a sequence from the min to the max with 100 equally spaced values
> span <- seq(rng[1], rng[2], length = 100)
```

We can now use the model to predict the values of body weight at each of those points for each of the two suborders. We've added `se.fit=TRUE` to the function to get the standard error around

the regression lines.

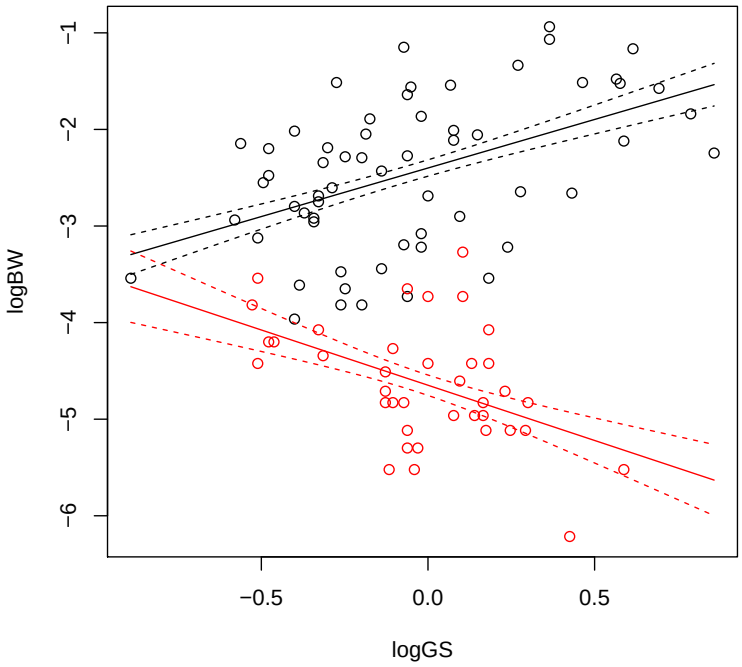
```
#get a data frame of new data for the order
> ZygoVals <- data.frame(logGS = span, Suborder = "Zygoptera")

#get the predictions and standard error
> ZygoPred <- predict(odonModel, newdata = ZygoVals, se.fit = TRUE)

#repeat for anisoptera
AnisoVals <- data.frame(logGS = span, Suborder = "Anisoptera")

AnisoPred <- predict(odonModel, newdata = AnisoVals, se.fit = TRUE)
```

Both `AnisoPred` and `ZygoPred` contain predicted values (called `fit`) and standard error values (called `se.fit`) for each of the values in `span` for each of the two suborders.

We can add the predictions onto a plot like this:

```
# plot the scatterplot of the data
> plot(logBW ~ logGS, data = odonata, col = Suborder)
# add the predicted lines
> lines(AnisoPred$fit ~ span, col = "black")
> lines(AnisoPred$fit + AnisoPred$se.fit ~ span, col = "black", lty = 2)
> lines(AnisoPred$fit - AnisoPred$se.fit ~ span, col = "black", lty = 2)
```

⋆ Copy the prediction code into your script and run the plot above. Copy and modify the last three lines to add the lines for the Zygoptera. Your final plot should look like this.



Chapter 9

Linear Models: Model simplification

Aims of this chapter:

- Simplifying complex models by removing non-explanatory terms

In biology, we often use statistics to compare competing hypotheses in order to work out the simplest explanation for some data. This often involves collecting several explanatory variables that describe different hypotheses and then fitting them together in a single model, and often including interactions between those variables.

In all likelihood, not all of these model *terms* will be important. If we remove unimportant terms, then the explanatory power of the model will get worse, but might not get significantly worse.

> "It can scarcely be denied that the supreme goal of all theory is to make the irreducible basic elements as simple and as few as possible without having to surrender the adequate representation of a single datum of experience."
> *Albert Einstein*

Or to paraphrase:

> "Everything should be made as simple as possible, but no simpler."

The approach we will look at is to start with a *maximal model* — the model that contains everything that might be important — and simplify it towards the *null model* — the model that says that none of your variables are important. Hopefully, there is a point somewhere in between where you can't remove any further terms without making the model significantly worse: this is called the *minimum adequate model*.

9.1 A maximal model

We'll be using the mammal dataset for this practical, so once again:

⋆ Make sure you have changed the working directory to your stats module `Code` folder.
⋆ Create a new blank script called 'ModelSimplif.R'.
⋆ Load the mammals data into a data frame called `mammals`.

In Chapters 7 & 8, we looked at how the categorical variables `GroundDwelling` and `TrophicLevel` predicted genome size in mammals. In this chapter, we will add in two more continuous variables: litter size and body mass. The first thing we will do is to log both variables and reduce the dataset to the rows for which all of these data are available:

```
#get logs of continuous variables
> mammals$logLS <- log(mammals$LitterSize)
> mammals$logCvalue <- log(mammals$meanCvalue)
> mammals$logBM <- log(mammals$AdultBodyMass_g)

# reduce dataset to five key variables
> mammals <- subset(mammals, select = c(logCvalue, logLS, logBM,
TrophicLevel, GroundDwelling))

# remove the row with missing data
> mammals <- na.omit(mammals)
```

- ⋆ Copy the code above into your script and run it
- ⋆ Check that the data you end up with has this structure:

```
 'data.frame':  240 obs. of  5 variables:
 $logCvalue     : num  0.94 1.322 1.381 1.545 0.888 ...
 $logLS         : num  1.1 1.12 0 0 1.52 ...
 $logBM         : num  10.83 4.87 11.46 10.86 3.23 ...
 $TrophicLevel  : Factor w/ 3 levels "Carnivore","Herbivore",..: 1 2 2 2 3 3 3 2 2 3 ↩
     ...
 $GroundDwelling: Factor w/ 2 levels "No","Yes": 2 2 2 2 2 1 2 1 1 1 ...
  - attr(*, "na.action")=Class 'omit'  Named int [1:139] 2 4 7 9 10 11 14 15 20 21 ...
   .. ..- attr(*, "names")= chr [1:139] "2" "4" "7" "9" ...
```

Now we'll fit a model including all of these variables and all of the interactions:

```
> model <- lm(formula = logCvalue ~ logLS * logBM * TrophicLevel *
GroundDwelling, data = mammals)
```

- ⋆ Run this model in your script.
- ⋆ Look at the output of `anova(model)` and `summary(model)`.

Scared? Don't be! There are a number of points to this exercise:

1. These tables show exactly the kind of output you've seen before. Sure, there are lots of rows but each row is just asking whether a model term (`anova`) or a model coefficient (`summary`) is significant.
2. Some of the rows are significant, others aren't: some of the model terms are not explanatory.
3. The two tables show slightly different things - lots of stars for the `anova` table and only a few for the `summary` table.
4. That last line in the `anova` table: `logLS:logBM:TrophicLevel:GroundDwelling`. This is an interaction of four variables capturing how the slope for litter size changes for different body masses for species in different trophic groups and which are arboreal or ground dwelling. Does this seem easy to understand?

The real lesson here is that it is easy to fit complicated models in R. *Explaining and understanding them is a different matter*. The temptation is always to start with the most complex possible model but this is rarely a good idea.

9.2 A better maximal model

Instead of all possible interactions, we'll consider two-way interactions: how do pairs of variables affect each other? There is a shortcut for this: `y ~ (a + b + c)^2` gets all two way combinations of the variables in the brackets, so is a quicker way of getting this model:

`y ~ a + b + c + a:b + a:c + b:c`.

So let's use this to fit a simpler maximal model:

```
> model <- lm(logCvalue ~ (logLS + logBM + TrophicLevel + GroundDwelling)^2, data = ↩
    mammals)
```

The `anova` table for this model looks like this:

```
> anova(model)

 Analysis of Variance Table

 Response: logCvalue
                            Df Sum Sq Mean Sq F value  Pr(>F)
 logLS                       1   0.99   0.989   25.72 8.2e-07 ***
 logBM                       1   3.03   3.032   78.83 < 2e-16 ***
 TrophicLevel                2   0.48   0.239    6.21  0.0024 **
 GroundDwelling              1   0.11   0.110    2.87  0.0915 .
 logLS:logBM                 1   0.27   0.275    7.15  0.0081 **
 logLS:TrophicLevel          2   0.19   0.095    2.48  0.0862 .
 logLS:GroundDwelling        1   0.14   0.136    3.55  0.0609 .
 logBM:TrophicLevel          2   0.09   0.044    1.14  0.3230
 logBM:GroundDwelling        1   0.88   0.883   22.96 3.0e-06 ***
 TrophicLevel:GroundDwelling 2   0.04   0.022    0.58  0.5607
 Residuals                 225   8.65   0.038
 ---
 Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The first lines are the *main effects*, which are all significant or near significant. Then there are the six interactions. One of these is very significant: `logBM:GroundDwelling`, which suggests that the slope of log C value with body mass differs between ground dwelling and non-ground dwelling species. The other interactions are non-significant although some are close.

- ⋆ Run this model in your script.
- ⋆ Look at the output of `anova(model)` and `summary(model)`.
- ⋆ Check the model diagnostic plots.

9.3 Model simplification

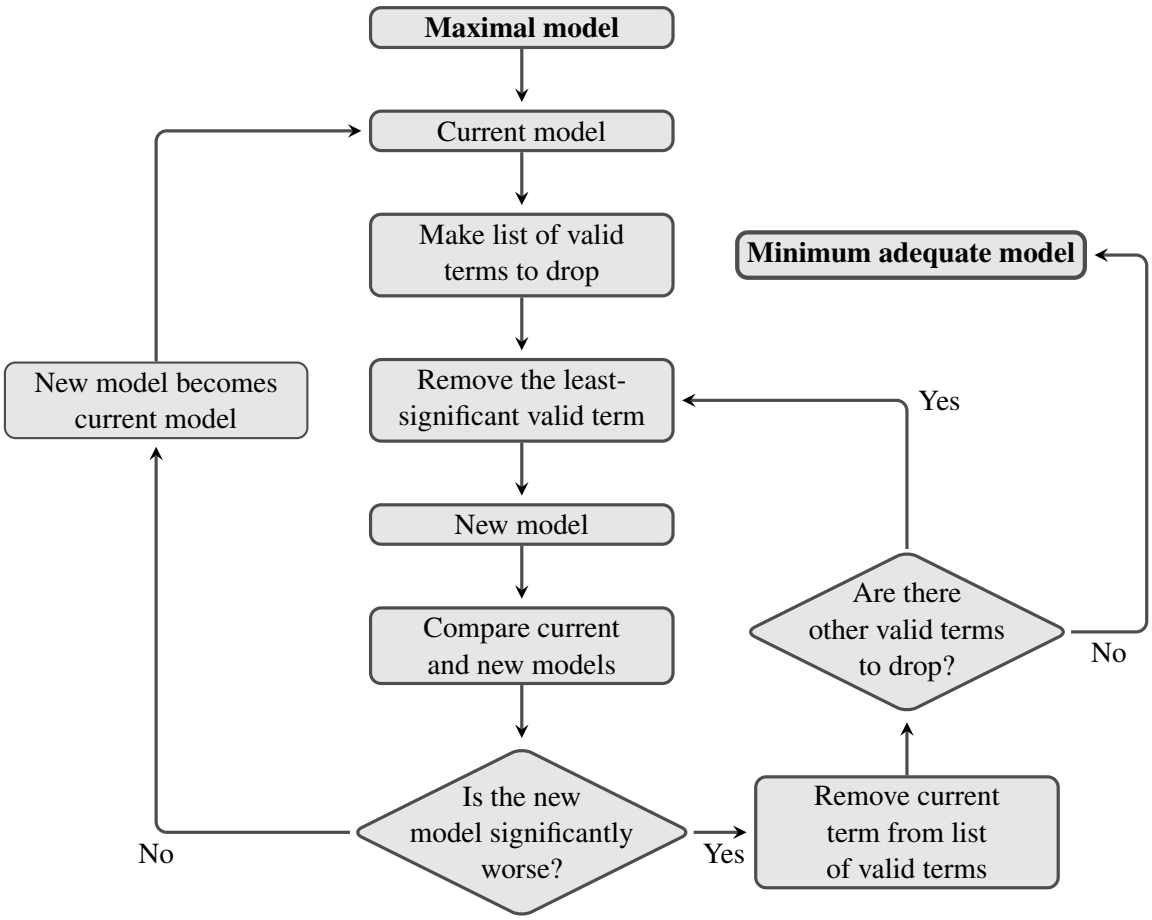
Model simplification is not a simple process. Each time you remove a term from a model, the model will change: the model will get worse, since some of the sums of squares are no longer explained, but the remaining variables may take over.

The first question is: *what terms can you remove from a model*? Obviously, you only want to remove non-significant terms, but there is another rule – you cannot remove a main effect or an interaction while those main effects or interactions are present in a more complex interaction. For example, in the model `y ~ a + b + c + a:b + a:c + b:c`, you cannot drop `c` without dropping both `a:c` and `b:c`.

The R function `drop.scope` tells you what you can drop from a model. Some examples:

```
> drop.scope(model)
 [1] "logLS:logBM"                 "logLS:TrophicLevel"
 [3] "logLS:GroundDwelling"        "logBM:TrophicLevel"
 [5] "logBM:GroundDwelling"        "TrophicLevel:GroundDwelling"

> drop.scope(y ~ a + b + c + a:b)
 [1] "c"   "a:b"
> drop.scope(y ~ a + b + c + a:b + b:c + a:b:c)
 [1] "a:b:c"
```

Model simplification is an iterative process. The flow diagram below shows how it works: at each stage you try and find an acceptable simplification. If successful, then you start again with the new simpler model and try and find a way to simplify this, until eventually, you can't find anything more to remove.



As always, we can use an F test to compare two models and see if they have significantly different explanatory power. Here, significance is a bad thing — it means that we've removed a term that makes the model significantly worse.

The last thing we need to do is work out how to remove a term from a model. We could type out the model again, but there is a shortcut using the function `update`:

```
# a simple model
> f <- y ~ a + b + c + b:c
```

```
# remove b:c from the current model
> update(f, . ~ . - b:c)
 y ~ a + b + c

# model g as a response using the same explanatory variables.
> update(f, g ~ .)
 g ~ a + b + c + b:c
```

Yes, the syntax is a little odd. The function uses a model or a formula and then allows you to alter the current formula. The dots in the code `. ~ .` mean 'use whatever is currently in the response or explanatory variables'. It gives a simple way of changing a model.

Putting this together, let's try a simplification. From the previous `anova` and `drop.scope` output, we know that the interaction `TrophicLevel:GroundDwelling` is not significant and a valid term.

```
# remove TrophicLevel:GroundDwelling
> model2 <- update(model, . ~ . - TrophicLevel:GroundDwelling)

# use anova to compare the two models
> anova(model, model2)

 Analysis of Variance Table

 Model 1: logCvalue ~ (logLS + logBM + TrophicLevel + GroundDwelling)^2
 Model 2: logCvalue ~ logLS + logBM + TrophicLevel + GroundDwelling +
            logLS:logBM + logLS:TrophicLevel + logLS:GroundDwelling +
            logBM:TrophicLevel + logBM:GroundDwelling
   Res.Df  RSS Df Sum of Sq    F Pr(>F)
 1    225 8.65
 2    227 8.70 -2   -0.0446 0.58   0.56
```

This tells us that `model2` is not significantly worse than `model`. We can now look at this model and see what else can be removed:

```
> anova(model2)

 Analysis of Variance Table

 Response: logCvalue
                      Df Sum Sq Mean Sq F value  Pr(>F)
 logLS                 1   0.99   0.989   25.82 7.8e-07 ***
 logBM                 1   3.03   3.032   79.12 < 2e-16 ***
 TrophicLevel          2   0.48   0.239    6.24  0.0023 **
 GroundDwelling        1   0.11   0.110    2.88  0.0909 .
 logLS:logBM           1   0.27   0.275    7.17  0.0079 **
 logLS:TrophicLevel    2   0.19   0.095    2.49  0.0854 .
 logLS:GroundDwelling  1   0.14   0.136    3.56  0.0604 .
 logBM:TrophicLevel    2   0.09   0.044    1.14  0.3216
 logBM:GroundDwelling  1   0.88   0.883   23.05 2.9e-06 ***
 Residuals           227   8.70   0.038
 ---
 Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

> drop.scope(model2)

 [1] "logLS:logBM"          "logLS:TrophicLevel"   "logLS:GroundDwelling"
 [4] "logBM:TrophicLevel"   "logBM:GroundDwelling"
```

- ⋆ Run this first simplification in your script.
- ⋆ Look at the output above and decide what is the next possible term to delete

- ⋆ Using the code above as a model, create `model3` as the next simplification! (remember to use `model2` in your update call and not `model`)

Now for a difficult exercise:

- ⋆ Using the code above to guide you, try and find a minimal adequate model that you are happy with. In each step, the output of `anova(model, modelN)` should be non-significant (where *N* is the current step).
- ⋆ It can be important to consider both `anova` and `summary` tables. It can be worth trying to remove things that look significant in one table but not the other — some terms can explain significant variation on the `anova` table but the coefficients are not significant.
- ⋆ Remember to remove *terms*: with categorical variables, several coefficients in the `summary` table may come from one term in the model and have to be removed together.
- ⋆ When you have got your final model, save the model as an R data file: `save(modelN, file='myFinalModel.Rda')`.

Chapter 10

Generalised Linear Models

Aims of this chapter:

- Use generalised linear models (GLMs) to handle count data.
- Analyse some genetics practical data.
- This chapter will step through the analysis carefully. These are not simple analyses so you should concentrate on understanding the process and the biology and think about how to present your results.

10.1 What is a GLM?

the generalized linear model (GLM) is a generalization of ordinary linear regression analyses to accommodate response variables to have non-normal error distributions (e.g., count data, as in the genetics practical data — see below).

10.2 The data

We will use mutation data collected by a previous year's batch in the Genetics Practical. So let's actually use some of the skills you've learned to do some statistical modelling on data you might collect. That is, you can aim to repeat these analyses with similar data you collect.

The students were basically counting colonies looking for mutations. There were a number of bacterial strains which were different mutants of *Salmonella*. Each group applied a mutagen Nitroguanisine (NG) as well as histidine and streptomycine. A control plate was also tested.

The data file in CSV format is available from the bitbucket site, as usual in the `Data` directory. It is called `PracData.csv`.

⋆ Save the `PracData.csv` dataset into your `Data` directory.
⋆ Create a new script called MyGLM in your `Code` directory. Use the code below to load and check your data.
⋆ Start R and change the working directory to `Code`.

```
> colonies <- read.csv("../Data/PracData.csv")
> str(colonies)
```

```
'data.frame':  680 obs. of  5 variables:
$Student.ID : Factor w/ 34 levels "A1","A10","A11",..: 1 1 1 2 2 2 4 4 4 4 ...
$Strain     : Factor w/ 5 levels "421","712","881",..: 4 3 5 1 2 3 4 2 3 5 ...
$Treatment  : Factor w/ 4 levels "Control","His",..: 1 1 1 3 3 1 1 3 3 1 ...
$ColonyCount: int  0 0 0 0 0 0 0 0 0 0 ...
$HaloLawn   : Factor w/ 2 levels "N","Y": NA NA NA NA NA NA NA NA NA NA ...
```

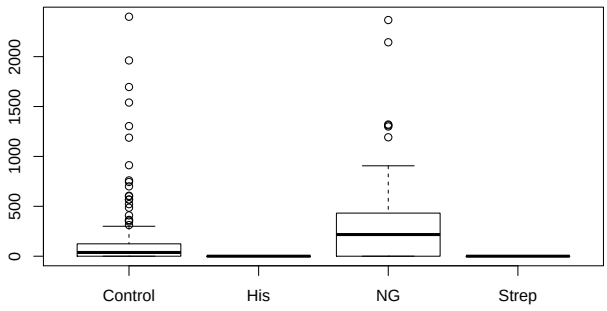
Now that we've got the data loaded, we need to look at it and try and see what is going on.

10.3 Plotting the data

We have a continuous response variable (`ColonyCount`) and two categorical explanatory variables (`Strain` and `Treatment`). We also have observations of halos and bacterial lawns around the treated areas (`HaloLawn`), which we will come back to at the end of this chapter.

So, with two factors as the explanatory variables, we will use box and whisker plots and boxplots to explore the data. First, we'll look at the effects of the four treatments.

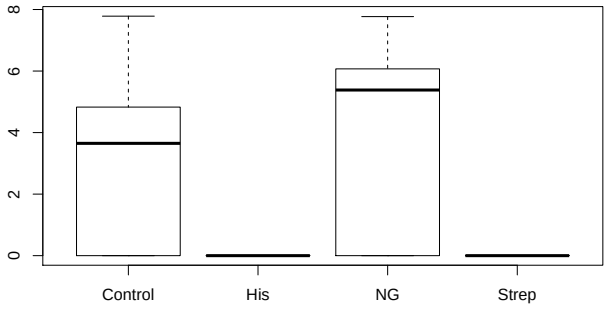
```
> boxplot(ColonyCount ~ Treatment, data=colonies)
```



There are two immediate things to note.

1. The distributions of colony counts are very *skewed* — many small counts and a few large counts. We've already seen that taking a log of data sometimes works in these cases. However, as the tables above show, we have zero counts for all treatments and $\log(0)$ is undefined. A common trick is therefore to use $\log(n+1)$ (add 1 and take a log) when dealing with count data like this:

```
> colonies$logCC <- log(colonies$ColonyCount + 1)
> boxplot(logCC ~ Treatment, data=colonies)
```



I hope you'll agree that this still doesn't look very convincingly like normal data, but we'll come back to this point.

2. The colony counts are vastly different between the different treatments. It is hard to say for sure from the two plots, but it looks like colonies never grow under the histidine and streptomycine treatments. We can check that:

```
> tapply(colonies$ColonyCount, colonies$Treatment, min, na.rm = TRUE)

 Control     His      NG   Strep
       0       0       0       0

> tapply(colonies$ColonyCount, colonies$Treatment, max, na.rm = TRUE)

 Control     His      NG   Strep
    2400       0    2367       0
```

There is indeed no variation at all in colony count for histidine and streptomycine — colonies never grow in these treatments. We don't really need statistics for this observation and, in fact, variation is needed for statistics to work. So, for the rest of this analysis, we will reduce the dataset to the control and nitroguanisine treatments.

⋆ Update your script to contain the code for these plots.

We'll use a new piece of code here to get the right subset. `var %in% c('a','b','c')` finds all entries in `var` whose values are equal to `'a'`, `'b'` or `'c'`.

```
> coloniesCN <- subset(colonies, Treatment %in% c("Control", "NG"),
drop = TRUE)
> str(coloniesCN)

 'data.frame':  340 obs. of  6 variables:
 $Student.ID : Factor w/ 34 levels "A1","A10","A11",..: 1 1 1 2 2 2 4 4 4 4 ...
 $Strain     : Factor w/ 5 levels "421","712","881",..: 4 3 5 1 2 3 4 2 3 5 ...
 $Treatment  : Factor w/ 4 levels "Control","His",..: 1 1 1 3 3 1 1 3 3 1 ...
 $ColonyCount: int  0 0 0 0 0 0 0 0 0 0 ...
 $HaloLawn   : Factor w/ 2 levels "N","Y": NA NA NA NA NA NA NA NA NA NA ...
 $logCC      : num  0 0 0 0 0 0 0 0 0 0 ...
```

You'll see that, although we have removed two treatments, their names still appear in the list of levels in the `str` output. R retains a list of all the levels that were originally in a factor, even when those levels aren't used any more. This will be annoying later, so we'll use the `droplevels` function to strip them out.

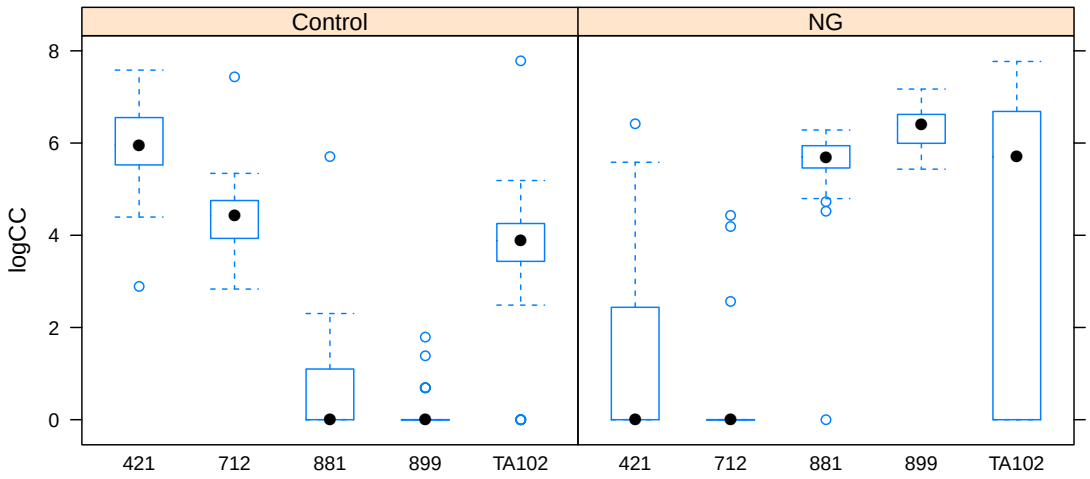
```
> coloniesCN <- droplevels(coloniesCN)
> str(coloniesCN)

 'data.frame':  340 obs. of  6 variables:
 $Student.ID : Factor w/ 34 levels "A1","A10","A11",..: 1 1 1 2 2 2 4 4 4 4 ...
 $Strain     : Factor w/ 5 levels "421","712","881",..: 4 3 5 1 2 3 4 2 3 5 ...
 $Treatment  : Factor w/ 2 levels "Control","NG": 1 1 1 2 2 1 1 2 2 1 ...
 $ColonyCount: int  0 0 0 0 0 0 0 0 0 0 ...
 $HaloLawn   : Factor w/ 0 levels: NA NA NA NA NA NA NA NA NA NA ...
 $logCC      : num  0 0 0 0 0 0 0 0 0 0 ...
```

⋆ Add these commands to subset your data to your code file.

10.4 Looking at strains too

Now we'll look to see how counts differ between the strains. A simple way to visualise this is to use the `lattice` package again to get plots grouped by treatment.

```
> library(lattice)
> bwplot(logCC ~ Strain | Treatment, data=coloniesCN)
```



First impressions from this figure:

1. The strains are doing *very* different things under the two treatments. Hopefully this now leaps out at you as suggesting that the two variables (Strain and Treatment) are *interacting*.
2. The distributions are still pretty ugly — the variances differ hugely between combinations and four combinations have a median of zero.

We could also use a barplot of means here. We'll use the original data to get the means, but can use a log scale on the y axis (`log='y'`).

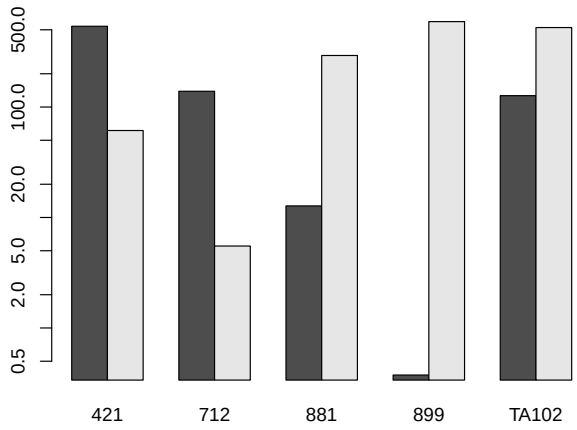
```
> tab <- tapply(coloniesCN$ColonyCount, list(coloniesCN$Treatment,
coloniesCN$Strain), mean, na.rm=TRUE)
> print(tab)
```

```
           421     712    881     899 TA102
Control 538.20 138.867  12.73   0.375 126.7
NG       61.29   5.517 292.71 593.000 523.9
```

An then,

```
> barplot(tab, beside=TRUE, log= ' y ' )
```

Which should give,

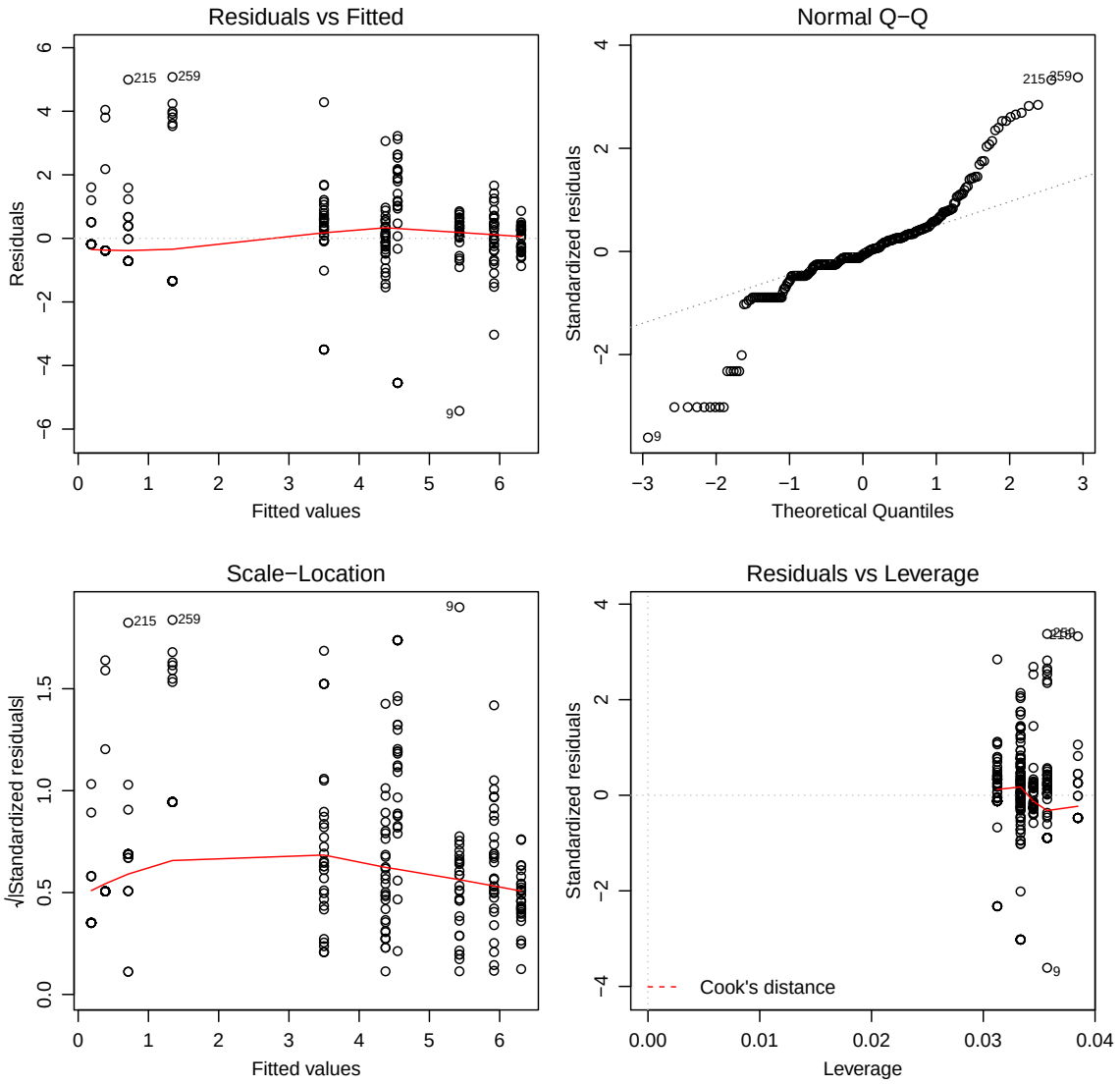


Lets have a look at a first model.

10.5 A linear model

We'll fit a model of colony count as the interaction between strain and treatment and then look at the diagnostic plots. We'd do this anyway, but we're already suspicious about the variance.

```
> modLM <- lm(logCC ~ Strain * Treatment, data=coloniesCN)
> par(mfrow=c(2,2), mar=c(3,3,3,1), mgp=c(2,0.8,0))
> plot(modLM)
```

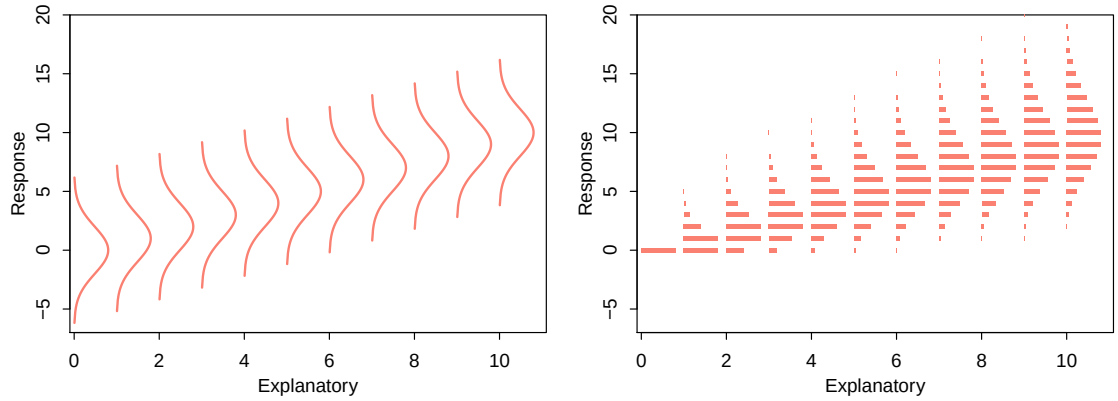


⋆ Run this code and have a close look at the plots.

That normal QQ plot is not good. Our suspicions were justified and it doesn't look like we can use a simple log transformation. We're not even going to look at the `anova` and `summary` tables — if the diagnostic plots are bad enough, then the model outputs are not to be trusted.

10.6 Generalised linear models

In the linear models lecture, we looked at the expectation of *constant normal variance* in linear models. Whatever the combination of explanatory variables for a particular prediction, the residuals around that prediction have similar variance and are roughly normally distributed. The panel on the left shows this basic idea.



As we have seem, count data does not have this distribution, even when logged. The panel on the right shows the expected distribution of count data as the mean count increases with an explanatory variable. There are three key differences between the two panels:

1. Counts can *never* be negative but can be zero.
2. Counts are always *integers* — whole numbers — rather than being continuous.
3. The variance of count data is *not constant*. As the average predicted count gets larger, so does the variance. Unlike the normal distribution, where variance can take any value, for count data the variance is expected to be equal to the mean.

So, we have data that is unsuitable for a linear model because it doesn't show constant normal variance. This is where generalised linear models come in — we can change the model for the expected residuals to use a different distribution. For count data, this is the *Poisson* distribution.

We need to change the function we use to fit models to `glm`, but otherwise the process is very similar. The whole point of the GLM is to model the original count data more appropriately, so we will abandon the logged data too. GLMs can cope with a range of different distributions, so we have to specify the `family` of the distribution we want to use.

```
> modPois <- glm(ColonyCount ~ Strain * Treatment, data=coloniesCN,
family= 'poisson')
```

First, we'll look at the summary table for this model. We have 5 levels of strain and 2 levels of factor in the subset so we get an intercept (i), 4 differences for strains(s_{2-5}), one difference for treatment (t_2) and then four differences for the interaction ($s_{2-5}t_2$). These combine like this:

	Control	Nitroguanisine
421	i	$i+t_2$
712	$i+s_2$	$i+s_2+t_2+s_2t_2$
881	$i+s_3$	$i+s_3+t_2+s_3t_2$
889	$i+s_4$	$i+s_4+t_2+s_4t_2$
TA102	$i+s_5$	$i+s_5+t_2+s_5t_2$

The summary table looks like this — very similar to the `summary` table for a linear model.

```
> summary(modPois)
```

```
Call:
glm(formula = ColonyCount ~ Strain * Treatment, family = "poisson",
    data = coloniesCN)

Deviance Residuals:
   Min      1Q  Median      3Q     Max
-32.37  -10.33   -3.32    0.84   97.84

Coefficients:
                      Estimate Std. Error z value Pr(>|z|)
(Intercept)            6.28823    0.00787   799.0   <2e-16 ***
Strain712             -1.35472    0.01738   -78.0   <2e-16 ***
Strain881             -3.74421    0.05549   -67.5   <2e-16 ***
Strain899             -7.26906    0.28878   -25.2   <2e-16 ***
StrainTA102           -1.44651    0.01757   -82.3   <2e-16 ***
TreatmentNG           -2.17268    0.02539   -85.6   <2e-16 ***
Strain712:TreatmentNG -1.05295    0.08446   -12.5   <2e-16 ***
Strain881:TreatmentNG  5.30786    0.06152    86.3   <2e-16 ***
Strain899:TreatmentNG  9.53871    0.28989    32.9   <2e-16 ***
StrainTA102:TreatmentNG 3.59226   0.03090   116.2   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for poisson family taken to be 1)

    Null deviance: 134445  on 293  degrees of freedom
Residual deviance:  61579  on 284  degrees of freedom
  (46 observations deleted due to missingness)
AIC: 62910

Number of Fisher Scoring iterations: 7
```

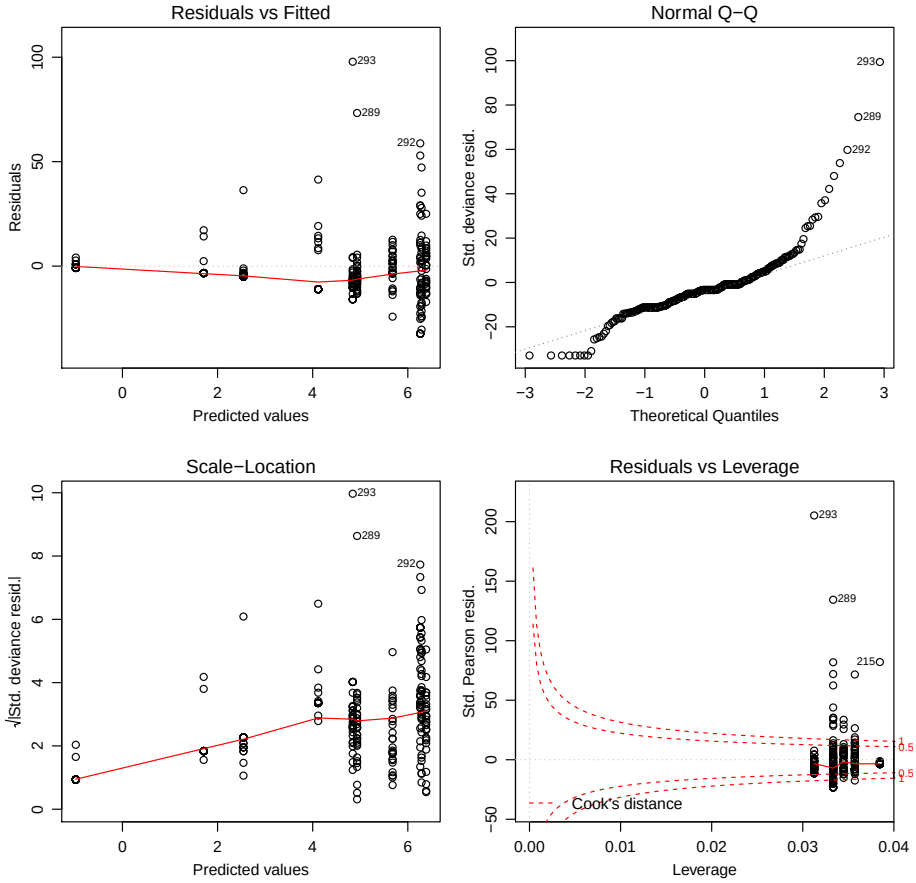
So, interpreting this table quickly. Under the control treatment, strain 421 (the intercept) has the highest number of colonies and all the other strains have lower numbers to some degree — the differences are negative. The overall effect of nitrogaunasine is to decrease the number of colonies — again a negative coefficient — but then the positive interactions show big increases in colony counts for nitroguanisine for specific strains. Everything is hugely significant.

⋆ Copy the code in this section into your script and explore the model.

10.7 Overdispersion

There's a problem. You may have already spotted it:

```
> par(mfrow = c(2, 2), mar = c(3, 3, 3, 1), mgp = c(2, 0.8, 0))
> plot(modPois)
```



Actually, there are two problems. First, that QQ plot is still a bit dubious. More of the points are close to the line than in the linear model but there are some extreme positive residuals. Second, the magnitude of the residuals is enormous, and this is really clear in the plot in the bottom right hand corner. This plot identifies outliers and any points outside of the red dotted line are possible problems.

The problem here is *overdispersion*. The Poisson distribution predicts that the variance at a point in the model is equal to the prediction — the mean count at that point. Our count data shows much more variance than this — particularly that there are some huge counts given the means.

There is a simple way to check the dispersion of count data using the `summary` table: the ratio of the residual deviance to the residual degrees of freedom should be approximately 1. This expectation is actually given in the table:

```
(Dispersion parameter for poisson family taken to be 1)
```

In this case, the ratio is $61579/284 = 216.8$. That's very strongly overdispersed. Fortunately, we can allow for this by using a different model.

10.8 Generalised linear models using quasipoisson

The quasipoisson family uses the data to estimate the dispersion of the model, but is otherwise very similar to using the Poisson family.

```
> modQPois <- glm(ColonyCount ~ Strain * Treatment, data=coloniesCN,
family= 'quasipoisson')
```

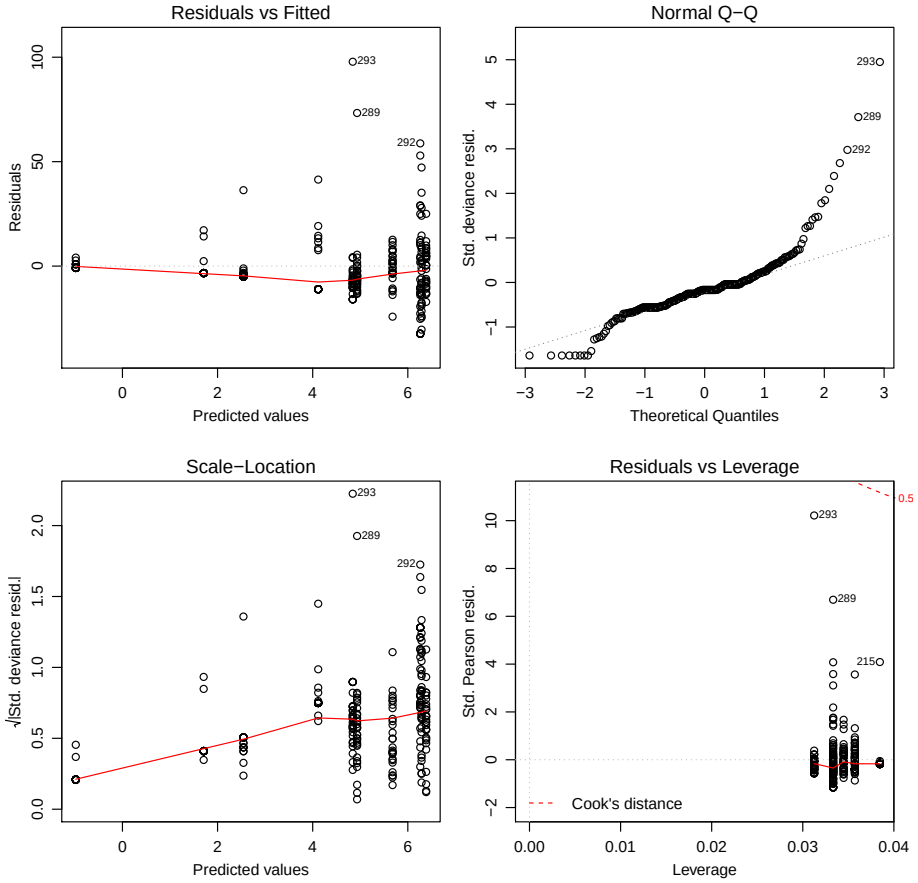
The summary table now looks like this:

```
> summary(modQPois)

 Call:
 glm(formula = ColonyCount ~ Strain * Treatment, family = "quasipoisson",
     data = coloniesCN)

 Deviance Residuals:
    Min      1Q  Median      3Q     Max
 -32.37  -10.33   -3.32    0.84   97.84

 Coefficients:
                       Estimate Std. Error t value Pr(>|t|)
 (Intercept)              6.288      0.158   39.78  < 2e-16 ***
 Strain712               -1.355      0.349   -3.88  0.00013 ***
 Strain881               -3.744      1.115   -3.36  0.00089 ***
 Strain899               -7.269      5.800   -1.25  0.21113
 StrainTA102             -1.447      0.353   -4.10  5.4e-05 ***
 TreatmentNG             -2.173      0.510   -4.26  2.8e-05 ***
 Strain712:TreatmentNG   -1.053      1.696   -0.62  0.53529
 Strain881:TreatmentNG    5.308      1.236    4.30  2.4e-05 ***
 Strain899:TreatmentNG    9.539      5.822    1.64  0.10246
 StrainTA102:TreatmentNG  3.592      0.621    5.79  1.9e-08 ***
 ---
 Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

 (Dispersion parameter for quasipoisson family taken to be 403.4)

     Null deviance: 134445  on 293  degrees of freedom
 Residual deviance:  61579  on 284  degrees of freedom
   (46 observations deleted due to missingness)
 AIC: NA

 Number of Fisher Scoring iterations: 7
```

This is pretty similar to the previous table but there two differences. First, the dispersion parameter line has changed. Second, all the p values have got less significant – this is the effect of controlling for the overdispersion.

We'll look at the model diagnostic plots next:

```
> par(mfrow = c(2, 2), mar = c(3, 3, 3, 1), mgp = c(2, 0.8, 0))
> plot(modQPois)
```



The residuals and leverage plot is now ok. The QQ plot is not better, but is still an improvement over the original linear model. We can't improve the model fit any more — it isn't perfect but we'll accept those imperfections. It is worth thinking about the imperfections though — what might give rise to occasional larger than expected counts of colonies?

We'll look at the `anova` table next. Technically, this is now analysis of deviance not analysis of variance but the concept is the same. Different tests are appropriate for different families of distribution, but we can use F here:

```
> anova(modQPois, test = "F")

 Analysis of Deviance Table

 Model: quasipoisson, link: log

 Response: ColonyCount

 Terms added sequentially (first to last)


                Df Deviance Resid. Df Resid. Dev     F  Pr(>F)
 NULL                              293     134445
 Strain          4    13923       289     120521  8.63 1.4e-06 ***
 Treatment       1     6055       288     114467 15.01 0.00013 ***
 Strain:Treatment  4    52888       284      61579 32.78 < 2e-16 ***
 ---
 Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Can we simplify the model? The interaction is the only term we can drop and looks highly significant, but we can check by deleting it.

```
> drop.scope(modQPois)

 [1] "Strain:Treatment"

> modQPois2 <- update(modQPois, . ~ . - Strain:Treatment)
> anova(modQPois, modQPois2, test = "F")

 Analysis of Deviance Table

 Model 1: ColonyCount ~ Strain * Treatment
 Model 2: ColonyCount ~ Strain + Treatment
   Resid. Df Resid. Dev Df Deviance    F Pr(>F)
 1       284      61579
 2       288     114467 -4   -52888 32.8 <2e-16 ***
 ---
 Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

No, that makes the model much worse, so we now have our final model.

⋆ Fit this new model in your script and check you've got the same results.

10.9 Model predictions

We can get model predictions and standard errors using the `predict` function. There is a difference though. GLMs use an internal transformation to model the data using a *link function* and the coefficients in the summary above are on the scale of the link transformation. For quasipoisson, this is a *log link*, which you can see in the output of `anova`. You can use `predict` to get predictions on the scale of the original *response*.

```
# use expand.grid to get all combinations of factors
> df <- expand.grid(Strain = levels(coloniesCN$Strain), Treatment =
levels(coloniesCN$Treatment))
> predict(modQPois, newdata = df, type = "response")

      1       2       3       4       5       6       7       8       9      10
538.200 138.867  12.731   0.375 126.687  61.286   5.517 292.714 593.000 523.900
```

Those are the same values as the means we calculated for the barplot. Adding standard errors to barplots is more difficult for GLMs and we won't go into it here.

10.10 Reporting the model

Reporting complicated statistics is a difficult business. There is a lot of detail involved and you want the reader to understand what you have done well enough to repeat the analysis if needed. You also have to summarise and explain the results without pages of R output.

Here are some pointers:

- What does the data show? Present a graph or a table to show the data you are about to model. *Always* include a figure or table legend and *always* refer to that figure or legend from the text.

- Have you transformed the data or used a subset? If so, why?
- What kind of model or statistical test have you used?
- With linear models, what is the response variable and what are the explanatory variables.
- Have you simplified the model and, if so, what was the most complex model you tried?
- How did you check the suitability of the model? Are there any problems with the model and, if so, what might cause them?
- If you summarise stats in text, you must include all the information about the test.
  - For F tests, this is F, the two degrees of freedom and the p value. For example: 'There is a significant interaction between treatment and strain ($F_{4,284} = 32.7, p < 0.0001$)'.
  - For t tests, this is the coefficient, the standard error, t, the degrees of freedom and p. For example, 'Across strains, the main effect of nitroguanisine is to reduce colony counts relative to the control (estimate=-2.17, s.e= 0.51, $t = -4.26$, df=284, $p < 0.0001$)'.
- With more complex models, it is common to present either the anova table or the coefficients table as a summary of the model output. Just include the tables from R output, not the information around it. See Table 1 for an example.
- *Never* just include chunks of raw output from R.
- Most importantly, what is the interpretation of the model. What is it telling you about the data?

Table 1: Coefficients from a GLM of treatment and strain as predictors of colony count.

	Estimate	Std. Error	t value	p
(Intercept)	6.29	0.16	39.78	<0.0001
Strain712	-1.35	0.35	-3.88	0.0001
Strain881	-3.74	1.11	-3.36	0.0009
Strain899	-7.27	5.80	-1.25	0.2111
StrainTA102	-1.45	0.35	-4.10	<0.0001
TreatmentNG	-2.17	0.51	-4.26	<0.0001
Strain712:TreatmentNG	-1.05	1.70	-0.62	0.5353
Strain881:TreatmentNG	5.31	1.24	4.30	<0.0001
Strain899:TreatmentNG	9.54	5.82	1.64	0.1025
StrainTA102:TreatmentNG	3.59	0.62	5.79	<0.0001

10.11 Halos and lawns

We'll keep this one simple since it is harder to analyse. The response variable (`HaloLawn`) is binary — the plates either have a lawn or not. We'll just look at a contingency table of how many plates have halos or lawns under each combination of treatment and strain.

```
> table(Halo = colonies$HaloLawn, Strain = colonies$Strain, Treatment =
colonies$Treatment)
```

```
, , Treatment = Control

    Strain
Halo 421 712 881 899 TA102
   N   0   0   0   0     0
   Y   0   0   0   0     0

, , Treatment = His

    Strain
Halo 421 712 881 899 TA102
   N   1   1   0   0     1
   Y  29  29  26  32    31

, , Treatment = NG

    Strain
Halo 421 712 881 899 TA102
   N   0   0   0   0     0
   Y   0   0   0   0     0

, , Treatment = Strep

    Strain
Halo 421 712 881 899 TA102
   N   5  30   5  21     0
   Y  25   0  25   9    32
```

So, lawns and halos are never recorded from nitroguanisine or the control. They're nearly always found with histidine and different strains have different response to streptomysin. Again, treatment and strain interact. Although you can use a χ^2 test with two dimensional contingency tables to look for independence between factors, you can't with a three-way table.