# Data analysis and Visualization with R

*Remko Duursma*[1,2], *Jeff Powell*[1], *Glenn Stone*[2]
[1]Hawkesbury Institute for the Environment
[2]School of Computing, Engineering and Mathematics
University of Western Sydney

September 20, 2012

# Contents

# Chapter 1

# Basics of R

## 1.1 Installing R and so on

### 1.1.1 R installation

To install R on your system, go here : http://cran.r-project.org, click on the download link at the top, and then 'base', and finally download the installer.

Run the installer, clicking 'Next' until the 'Startup Options' pane. Select 'Yes' here and continue. On the next page, select 'Separate windows' - most users prefer this.

The next page asks for the help type : just use HTML help, the default.

Next, if you are using a Windows machine, and you are connecting to the internet via a proxy server ( *when on the UWS campus*, for example!), select 'Internet2'. If you select this option, R still works fine when you are not connecting through a proxy.

Connecting to the internet in R via a proxy server is more complicated when you are using a Mac. You need to set an environment variable that specifies the proxy server through which you connect to the internet. On the UWS campus, you can use

```
Sys.setenv(http_proxy = "http://ID:PW@proxy.uws.edu.au:3128/")
```

to set this up at the start of each session. 'ID' is your student/staff ID and 'PW' is your password. It is better to do this manually for each session if you frequently use R off campus.

### 1.1.2 Rstudio

We will use Rstudio throughout this course. Go here: www.rstudio.org to download it (Windows or Mac).

Familiarize yourself with the different windows: the source editor (a text editor, on the left), an R console (by default, on the bottom). To run a piece of code, select it in the text editor, and press `Crtl-Enter`.

### 1.1.3 Packages

Throughout this tutorial, we will be mostly be using the basic functionality of `R`. There are many add-on 'packages' in `R` that include additional functions. You can read a full list of packages on the CRAN site (http://cran.r-project.org/, navigate to 'Packages').

In Rstudio, click on the 'packages' tab in the lower-right hand panel. There you can see which packages are already installed, and you can install more by clicking on the 'Install packages' button.

Alternatively, to install a package, simply type:

```
install.packages("gplots")
```

You will be asked to select a mirror (select a site closest to your location), and the package will be installed to your local library. The package needs to be installed only once. To use the package in your current `R` session, type:

```
library(gplots)
```

You can now use the functions available in the `gplots` package. If you close and reopen `R`, you will need to load the package again.

To quickly learn about the functions included in a package, type:

```
library(help = nlme)
```

For any function in a package, a built-in help file is available. For example, to read about the function `bandplot` in the `gplots` package, type :

```
`?`(bandplot)
```

*Note:* Every help file has a few examples at the bottom of the file. These are usually the best place to quickly learn how to use the function (although, the examples are of varying quality).

## 1.2 Basic operations

### 1.2.1 R is a calculator

When you open the `R` console, all you see is a friendly > staring at you. You can simply type code, hit `Enter`, and R will write output to the screen.

Throughout this tutorial, R code will be shown together with output in the following way:

```
# I want to add two numbers:
1 + 1

## [1] 2
```

Here, I typed `1 + 1`, hit `<Enter>`, and `R` produced 2. The `[1]` means that the result only has one element (the number '2').

In this manual, the `R` output is shown after `##`. Every example can be run by you, simply copy the section (use the text selection tool in Adobe reader), and paste it into the console (with Ctrl-V on a Windows machine).

We can do all sorts of basic calculator operations. Consider the following examples:

```
# Arithmetic
12 * (10 + 1)

## [1] 132


# Scientific notation
3500 + 0.4

## [1] 3500
```

```
# pi is a built-in constant
sin(pi/2)

## [1] 1

# Absolute value
abs(-10)

## [1] 10

# Yes, you can divide by zero
1001/0

## [1] Inf

# Square root
sqrt(225)

## [1] 15

# Exponents
15^2

## [1] 225

# Round down to nearest integer (and ceiling() for up)
floor(3.1415)

## [1] 3
```

See ?Math for more mathematics.

Also note the use of # for comments: anything after this symbol on the same line is *not* read by R (*Note:* R does not have a section-comment syntax). Throughout this manual, comments are shown in green.

It gets more interesting when we introduce variables. Let's look at these simple examples:

```
# Define two objects
x <- pi/2
y <- pi/3

# ... and use them in basic calculations.
sin(x) + cos(y)

## [1] 1.5
```

Note the use of `<-` : this is an operator that assigns some content to an 'object'. We constructed two objects, `x` and `y`.

These objects now remain in memory, so we can reuse these objects until we close R. You can also assign something else to the object,

```
x <- "Hello world"
message(x)

## Hello world
```

Here I assigned a character string to `x`, note that it is not a problem to overwrite an existing object with a different type of data (unlike some other programming languages).

### 1.2.2 Vectors are strings of numbers

A very useful type of object is the `vector`, which is basically a string of numbers or bits of text (but not a combination of both). The power of R is that most functions can use a vector directly as input, which greatly simplifies coding in many applications.

Let's construct two numeric vectors, both have 7 numbers:

```
nums1 <- c(1, 4, 2, 8, 11, 100, 8)
nums2 <- c(3.3, 8.1, 2.5, 9.8, 21.2, 13.8, 0.9)
```

We can now do basic arithmetic with these `numeric vectors` :

```
# Get the sum of the two vectors:
sum(nums1) + sum(nums2)

## [1] 193.6


# Add the two vectors (results in a new vector).  For this to work, both
# vectors must be the same length (i.e., in this case, both have 7
# elements).
nums1 + nums2

## [1]   4.3  12.1   4.5  17.8  32.2 113.8   8.9


# Get mean, standard deviation, number of observations (length):
mean(nums1)

## [1] 19.14

sd(nums1)

## [1] 35.83

length(nums1)

## [1] 7


# Get sum of squared deviation from the sample mean:
sum((nums1 - mean(nums1))^2)

## [1] 7705
```

There are many more functions you can use directly on vectors. Here are a few useful ones (some of these will be used in the Exercises).

| Function | Does | Example |
|---|---|---|
| length | Returns the length of the vector | length(nums1) |
| rev | Reverses elements of a vector | rev(nums1) |
| sort | Sorts elements of a vector | sort(nums1) |
| order | Returns the order of elements in a vector | order(nums1) |
| head | Prints the first few elements of a vector | head(nums1, 3) |
| max,min | Returns the maximum or minimum value of the vector | min(nums1) |
| which.max,which.min | Which element of the vector is the max or min value? | which.min(nums1) |
| cumsum | Returns the cumulative sum of the vector | cumsum(nums1) |
| diff | Sequential difference between elements of a vector | diff(1:10) |
| unique | Lists all the unique values of a vector | unique(c(5,5,10,10,11)) |

A vector can also consist of `character` elements. Construct a character vector like this (don't forget the quotes, otherwise R will look for objects with these names).

```
words <- c("pet", "elk", "star", "apple", "the letter r")
```

Many of the same functions as summarized in the table above also apply to `character vectors` with the exception of obviously numerical ones. The `sort` function works as well, to alphabetize a vector:

```
sort(words)

## [1] "apple"        "elk"          "pet"          "star"
## [5] "the letter r"
```

And count the number of characters in each of the elements like this,

```
nchar(words)

## [1]  3  3  4  5 12
```

We will take a closer look at working with character strings a bit later (see Section 2.3)

### 1.2.3 Objects in the workspace

So far, we have constructed five objects. These objects are kept in memory for the remainder of your session (that is, until you close R).

In Rstudio, you can browse all objects that are currently loaded in memory. Find the window that has the tab 'Workspace' (typically on the top-left). Here you see a list of all the objects you have created in this `R` session. When you click on an object, a window opens that shows the contents of the object.

Alternatively, to see which objects you currently have in your `workspace`, use the following command:

```
ls()

## [1] "nums1" "nums2" "words" "x"     "y"
```

To remove objects,

```
rm(nums1, nums2)
```

And to remove all objects that are currently loaded, use this command. **Note:** you want to use this wisely!

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

Finally, when you are ending your R session, but you want to continue exactly at this point the next time, make sure to save the current workspace. In Rstudio, find the menu 'Workspace' (at the top). Here you can save the current workspace, and also load a previously saved one.

### 1.2.4 Files in the working directory

Each time you run R, it 'sees' one of your folders ('directories') and all the files in it. This folder is called the `working directory`. You can change the working directory in Rstudio in a couple of ways.

The first option is to go to the menu `Tools/Set working directory/Choose directory...`.

Or, find the 'Files' tab in one of the Rstudio windows (usually bottom-right). Here you can browse to the directory you want to use, and click 'More/Set as working directory'.

You can also set or query the current working directory by typing the following code:

```
# Set working directory to C:\\myR
setwd("C:\\myR")
# What is the current working directory?
getwd()
```

Finally, you can see which files are available in the current working directory, as well as in subdirectories:

```
# Show files in the working directory:
list.files()
# List files in some other directory:
list.files("c:/work/projects/data/")
# Show files in the subdirectory 'data' (note that it needs to exist):
list.files("./data")
# Show files in the working directory that end in csv.  (The
# ignore.case=TRUE assures that we find files that end in 'CSV' as well as
# 'csv', and the '\\' is necessary to find the '.' in '.csv'.
list.files(pattern = "\\.csv", ignore.case = TRUE)
```

### 1.2.5 Keep a clean memory

When you start `R`, it checks whether a file called '.RData' is available in the default working directory. If it is, it loads the contents from that file automatically. This means that, over time, objects can accumulate in your memory, cluttering your workspace and potentially causing problems if you are using old objects without knowing it.

I strongly recommend that each script you write starts like this:

```
# Set working directory
setwd("C:\\projects\\data")
# Clean workspace
rm(list = ls())
# (Optionally) Load some previously stored R session
load("May2011.RData")
```

This sort of workflow avoids common problems where old objects are being used unintentionally. In summary: **Always:**

- Make sure you are in the correct working directory
- Make sure your workspace is clean, or contains objects you know
- Write scripts that contain your entire workflow
- Once you have a working script, run the full script to make the final output

### 1.2.6 Accessing the help files

Every function that you use in R has its own built-in help file. To access the help file for the arithmetic mean, for example:

```
?mean
```

This opens up the help file in your default browser (but you do not need to be online to read help files).

Be very careful reading help files. Much of the steep learning curve in `R` is to do with the rather obscure and often confusing help files. A good tip for beginners is to not read the help files, but skip straight to the Example section at the bottom of the help file.

Nonetheless, becoming competent in `R` means you have to learn to decipher the help files. See section 'Understanding help files' in the 'Functions' section to get you started, including some exercises.

## 1.3 Generating data

### 1.3.1 Sequences of numbers

Let's look at a few ways to generate sequences of numbers, that we can use in the examples and exercises. There are also a number of real-world situations where you want to use these functions (see Exercises).

First, as we saw already, we can use the `c()` function to 'concatenate' (link together) a series of numbers. We can also combine vectors in this way:

```
a <- c(1, 2, 3)
b <- c(4, 5, 6)
c(a, b)

## [1] 1 2 3 4 5 6
```

A sequence of numbers can be constructed in the following ways:

```
1:10  # Numbers 1 through 10

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

5:-5  # From 5 to -5, backwards

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


# Examples using seq()
seq(from = 10, to = 100, by = 10)

##  [1]  10  20  30  40  50  60  70  80  90 100

seq(from = 23, by = 2, length = 12)

##  [1] 23 25 27 29 31 33 35 37 39 41 43 45


# Replicate numbers:
rep(2, 10)

##  [1] 2 2 2 2 2 2 2 2 2 2

rep(c(4, 5), each = 3)

## [1] 4 4 4 5 5 5
```

### 1.3.2 Random numbers

We can draw random numbers using the `runif` function:

```
# Ten random numbers between 0 and 1
runif(10)

##  [1] 0.801865 0.100353 0.004197 0.898645 0.469246 0.855754 0.923365
##  [8] 0.057752 0.433015 0.319565

# Five random numbers between 100 and 1000
runif(5, 100, 1000)

## [1] 663.0 396.1 455.2 731.8 462.8
```

The `runif` function is part of a much larger class of functions, each of which returns numbers from a different probability distributions. For example, look at the functions `rnorm` for the normal distribution, and `rexp` for the exponential distribution.

Next, we will `sample` numbers from an existing vector.

```
numbers <- 1:15
sample(numbers, size = 20, replace = TRUE)

##  [1]  9 10 13  5 10 13  8 14  1 12  5  6  3 10 12  7 10  4 13  7
```

This command samples 20 numbers from the `numbers` vector, with replacement.

## 1.4 Reading data

### 1.4.1 Reading CSV files

There are many ways to read data into R, but I am going to keep things simple and show only one option. This is a very convenient and safe way to read in data. Let's assume you have a fairly standard dataset, with variables organized in columns, and individual records in rows, and individual fields separated by a comma (a comma-separated values (CSV) file).

If you don't have your data in a CSV file (yet), see next section.

Use the following command to read in the first example dataset (`Allometry.csv`). This assumes that the file `"Allometry.csv"` is in your current working directory.

```
allom <- read.csv("Allometry.csv")
```

If the file is stored elsewhere, you have to specify the entire path:

```
allom <- read.csv("c:/projects/data/allom.csv")
```

This reads in the entire dataset, and stores it in an object, that I called `allom`. This type of object is called a `dataframe`. We will be using dataframes a lot throughout this tutorial.

To read a description of this example dataset, see Appendix, section A.1 on page 112.

To look at the entire dataset after reading, simply type:

```
allom
```

Alternativel, in Rstudio, find the dataframe in the 'Workspace' tab. If you click on it, the dataframe is shown in a separate window.

Or, if you have a very large dataset, only print the first (or last) few rows of the dataframe:

```
head(allom)

##   species diameter height leafarea branchmass
## 1    PSME    54.61  27.04  338.486    410.246
## 2    PSME    34.80  27.42  122.158     83.650
## 3    PSME    24.89  21.23    3.958      3.513
## 4    PSME    28.70  24.96   86.351     73.130
## 5    PSME    34.80  29.99   63.351     62.390
## 6    PSME    37.85  28.07   61.373     53.866

tail(allom)

##    species diameter height leafarea branchmass
## 58    PIMO    73.66  44.64  277.494     275.72
## 59    PIMO    28.19  22.59  131.857      91.76
## 60    PIMO    61.47  44.99  121.429     199.86
## 61    PIMO    51.56  40.23  212.444     220.56
## 62    PIMO    18.29  12.98   82.093      28.05
## 63    PIMO     8.38   4.95    6.551       4.37
```

Note the row numbers are shown on the left. These can also be accessed with `rownames(allom)`.

The function `read.csv` has many options, let's look at some of them. We can skip a number of rows from being read, and only read a fixed number of rows. For example, use this command to read rows 10-15, and to not read the header line (which is in line 1, and therefore not read).

```
allomsmall <- read.csv("Allometry.csv", skip = 9, nrows = 5, header = FALSE)
```

### 1.4.2 Reading other data

#### Excel spreadsheet

The most frequently asked question is how to read an Excel spreadsheet into R? The shortest answer: *don't do it*. In this manual, we assume you always first save an XLS or XLSX file as a CSV. In Excel, select `Save as...`, click on the button next to `'Save as type...'` and find `'CSV (Comma delimited) (*.csv)'`.

If you do need to read an XLS file, look at the `read.xls` function in the `gdata` package. Note that you need to have Perl installed for this to work! Also, it can be very slow for larger datasets.

It is generally good practice to store your raw data in CSV files, as these are simple text files that can be read by any type of software. Excel spreadsheets may contain formulas and formatting, which we don't need, and usually require Excel to read.

#### Tab-delimited text files

Sometimes, data files are provided as text files that are TAB-delimited. To read these files, use the following command:

```
mydata <- read.table("sometabdelimdata.txt", header = TRUE)
```

When using `read.table`, you must specify that a header (i.e., a row with column names) is present in the dataset.

### Reading typed data

You can also write the dataset in text, and read it as in the following example. This is useful if you have a small dataset that you typed in by hand (this example is from `?read.table`).

```
read.table(header=TRUE, text="
a b
1 2
3 4
")
```

```
##   a b
## 1 1 2
## 2 3 4
```

### Reading data from the clipboard

A very quick way to read a dataset from Excel is to use your clipboard. In Excel, select the data you want to read (including the header names), and press `Ctrl-C` (Windows), or the equivalent on other platforms. Then, in `R`, type:

```
mydata <- read.delim("clipboard", header = TRUE)
```

This is not a long-term solution to reading data, but is a very quick way to read (part of) a messy spreadsheet that someone shared with you.

### Other foreign formats

Finally, if you have a dataset in some unusual format, consider the `foreign` package, which provides a number of tools to read in other formats (such as SAS, SPSS, etc.).

## 1.5 Types of data

For the purpose of this tutorial, there are five types of data that the dataframe can contain. The first is `numeric`, which is exactly that (Note that in `R`, there is only one type of numerical data, unlike many other programming languages).

The other four data types are summarized in the table below:

| Data type | Description | Example |
|---|---|---|
| `numeric` | Any number | `c(1, 12.3491, 10/2, 10*6)` |
| `character` | Character strings | `c("E. saligna", "HFE", "a b c")` |
| `factor` | Categorical variable | `as.factor(c("Control","Fertilized","Irrigated"))` |
| `logical` | Either TRUE or FALSE | `10 == 100/10` |
| `Date` | Special Date class | `as.Date("2010-6-21")` |

Also, `R` has a very useful built-in data type to represent missing values. This is represented by `NA` (Not Available).

*Warning:* Because `NA` represents a missing value, make sure you never use 'NA' as an abbreviation for anything (like North America...).

Chapter 2 will take a close look at each of these data types.

To find out what type of data a particular vector is, use this example:

```
x <- "sometext"
str(x)

##  chr "sometext"

# 'chr' is short for 'character'

y <- c(1, 100, 10)
str(y)

##  num [1:3] 1 100 10

# This example also shows the dimension of the vector ([1:3]).  'num' is
# short for 'numeric'
```

To test for a particular type of data, use the `is.` functions, which give TRUE if the object is of that type, for example:

```
# Test for numeric data type:
is.numeric(c(10, 189))

## [1] TRUE


# Test for character:
is.character("HIE")

## [1] TRUE
```

There are many more, we will run into some of these later.

## 1.6 Working with dataframes

This manual focuses heavily on dataframes, because this is the object you will use most of the time in day-to-day data analysis. The following sections provide a brief introduction, see many examples using dataframes throughout this manual.

### 1.6.1 Variables in the dataframe

Let's first read the `allom` data, if you have not done so already.

```
allom <- read.csv("Allometry.csv")
```

Individual variables in a dataframe can be extracted using the dollar ($) sign. Let's print all the tree diameters here, after rounding to one decimal point:

```
round(allom$diameter, 1)

##  [1] 54.6 34.8 24.9 28.7 34.8 37.9 22.6 39.4 39.9 26.2 43.7 69.8 44.5 56.6
## [15] 54.6  5.3  6.1  7.4  8.3 13.5 51.3 22.4 69.6 58.4 33.3 44.2 30.5 27.4
## [29] 43.2 38.9 52.6 20.8 24.1 24.9 46.0 35.0 23.9 60.2 12.4  4.8 70.6 11.4
## [43] 11.9 60.2 60.7 70.6 57.7 43.1 18.3 43.4 18.5 12.9 37.9 26.9 38.6  6.5
## [57] 31.8 73.7 28.2 61.5 51.6 18.3  8.4
```

This way, it is also straightforward to add new variables to the dataframe. Let's convert the tree diameter to inches, and add it to the dataframe as a new variable:

```
allom$diameterInch <- allom$diameter/2.54
```

Instead of using the dollar-notation every single time, which can result in lengthy messy code when your variable names are long, you can also use the following syntax. Let's add a new variable `volindex`, a volume index defined as the square of tree diameter times height:

```
allom$volindex <- with(allom, diameter^2 * height)
```

This is exactly the same as writing:

```
allom$volindex <- allom$diameter^2 * allom$height
```

The `with` function allows for more readable code, while at the same time making sure that the variables `diameter` and `height` are read from the dataframe `allom`.

After adding new variables, it is good practice to look at the result. To 'print' the first few rows (that is, write to screen), recall:

```
head(allom)

##   species diameter height leafarea branchmass diameterInch volindex
## 1    PSME    54.61  27.04  338.486    410.246       21.500    80640
## 2    PSME    34.80  27.42  122.158     83.650       13.701    33207
## 3    PSME    24.89  21.23    3.958      3.513        9.799    13152
## 4    PSME    28.70  24.96   86.351     73.130       11.299    20559
## 5    PSME    34.80  29.99   63.351     62.390       13.701    36319
## 6    PSME    37.85  28.07   61.373     53.866       14.902    40214
```

A simple summary of the dataframe can be printed with the `summary` function:

```
summary(allom)

##  species      diameter          height          leafarea
##  PIMO:19   Min.   : 4.83   Min.   : 3.57   Min.   :  2.6
##  PIPO:22   1st Qu.:21.59   1st Qu.:21.26   1st Qu.: 28.6
##  PSME:22   Median :34.80   Median :28.40   Median : 86.4
##            Mean   :35.56   Mean   :26.01   Mean   :113.4
##            3rd Qu.:51.44   3rd Qu.:33.93   3rd Qu.:157.5
##            Max.   :73.66   Max.   :44.99   Max.   :417.2
##    branchmass       diameterInch     volindex
##  Min.   :   1.8   Min.   : 1.9   Min.   :    83
##  1st Qu.:  16.9   1st Qu.: 8.5   1st Qu.:  9906
##  Median :  72.0   Median :13.7   Median : 34889
##  Mean   : 145.0   Mean   :14.0   Mean   : 55269
##  3rd Qu.: 162.7   3rd Qu.:20.2   3rd Qu.: 84155
##  Max.   :1182.4   Max.   :29.0   Max.   :242208
```

For the factor variable, the output is basically the same as `table(allom)`. For the numeric variables, the minimum, 1st quantile, median, mean,3rd quantile, and maximum values are printed. If the variables have missing values, the number of missing values is printed as well (see Section 'Working with missing values').

To check the types of variables in the dataframe, use the `str` function (short for 'structure'). This function is useful for other objects as well, to view in detail what the object contains.

```
str(allom)

## 'data.frame': 63 obs. of  7 variables:
##  $ species     : Factor w/ 3 levels "PIMO","PIPO",..: 3 3 3 3 3 3 3 3 3 3 ...
##  $ diameter    : num  54.6 34.8 24.9 28.7 34.8 ...
##  $ height      : num  27 27.4 21.2 25 30 ...
##  $ leafarea    : num  338.49 122.16 3.96 86.35 63.35 ...
##  $ branchmass  : num  410.25 83.65 3.51 73.13 62.39 ...
##  $ diameterInch: num  21.5 13.7 9.8 11.3 13.7 ...
##  $ volindex    : num  80640 33207 13152 20559 36319 ...
```

To see how many rows and columns your dataframe contains (handy for double-checking you read the data correctly),

```
nrow(allom)

## [1] 63

ncol(allom)

## [1] 7
```

## 1.7 Subsetting vectors and dataframes

### 1.7.1 Vectors

Let's look at reordering or taking subsets of a vector, or `indexing` as it is commonly called. This is an important skill to learn, so we will look at several examples.

Let's recall the two `numeric vectors`:

```
nums1 <- c(1, 4, 2, 8, 11, 100, 8)
nums2 <- c(3.3, 8.1, 2.5, 9.8, 21.2, 13.8, 0.9)
```

Individual elements of a vector can be extracted using '[ ]'. For example, extract the first and fifth element of a vector:

```
nums1[1]

## [1] 1

nums1[5]

## [1] 11
```

You can also use another variable to do the indexing, for example,

```
# Get last element:
nelements <- length(nums1)
nums1[nelements]

## [1] 8
```

This last example extracts the last element of a vector. To do this, we first found the length of the vector, and used that to `index` the vector to extract the last element.

We can also select multiple elements, by `indexing` the vector with another vector. Recall how to construct sequences of numbers, explained in section 1.3.1 on page 8.

```
# Select a few elements of a vector:
selectthese <- c(1, 5, 2)
nums1[selectthese]

## [1]  1 11  4
```

```
# Select every other element:
everyother <- seq(1, 7, by = 2)
nums1[everyother]

## [1]  1  2 11  8


# Select five random elements:
ranels <- sample(1:length(nums2), 5)
nums2[ranels]

## [1]  9.8  8.1  0.9 21.2  3.3
```

Next, we can look at selecting elements of a `vector` based on the values in that vector. Suppose we want to make a new vector, based on vector `nums2` but only where the value within certain bounds.

```
# Subset of nums2, where value is at least 10 :
nums2[nums2 > 10]

## [1] 21.2 13.8


# Subset of nums2, where value is between 5 and 10:
nums2[nums2 > 5 & nums2 < 10]

## [1] 8.1 9.8


# Subset of nums2, where value is smaller than 1, or larger than 20:
nums2[nums2 < 1 | nums2 > 20]

## [1] 21.2  0.9


# Subset of nums1, where value is exactly 8:
nums1[nums1 == 8]

## [1] 8 8


# Subset of nums1, where value is one of 1,4 or 11:
nums1[nums1 %in% c(1, 4, 11)]

## [1]  1  4 11


# Subset of nums1, where value is NOT 1,4 or 11:
nums1[!(nums1 %in% c(1, 4, 11))]

## [1]   2   8 100   8
```

These examples showed you the following operators: >for 'greater than', '&' for 'AND', '==' for 'equal to', '—' for 'OR', '`%in%`' for 'is an element of', and "!" for NOT. See the help page `?Logic` for more details on logic operators.

Refer to the Exercises for more examples on use of these operators.

### 1.7.2 Subsetting dataframes

There are two ways to take a subset of a dataframe: using the square brackets (`[]`) much like the above examples, or using the `subset` function. We will learn both, as they are both useful from time to time.

Dataframes can be indexed with row and column numbers, like this:

```
mydataframe[row,column]
```

Here, 'row' refers to the row number (can be a vector of any length), and column to the column number (can also be a vector). Especially for columns it is very useful that you can also refer to the column by its *name* rather than its number. All this will become clear after some examples.

Let's look at a few examples using the Allometry dataset (see Section A.1 for a description of the dataset).

```
# Read data
allom <- read.csv("allometry.csv")

# Recall the names of the variables, the number of columns, and number of
# rows:
names(allom)

## [1] "species"    "diameter"   "height"     "leafarea"   "branchmass"

nrow(allom)

## [1] 63

ncol(allom)

## [1] 5


# Extract tree diameters: take the 4th row of the 2nd variable:
allom[4, 2]

## [1] 28.7


# We can also index the dataframe by its variable name:
allom[4, "diameter"]

## [1] 28.7


# Extract the first 3 rows of 'height':
allom[1:3, "height"]

## [1] 27.04 27.42 21.23


# Extract the first 5 rows, of ALL variables:
allom[1:5, ]

##   species diameter height leafarea branchmass
## 1    PSME    54.61  27.04  338.486    410.246
## 2    PSME    34.80  27.42  122.158     83.650
## 3    PSME    24.89  21.23    3.958      3.513
## 4    PSME    28.70  24.96   86.351     73.130
## 5    PSME    34.80  29.99   63.351     62.390


# Extract ther fourth column:
allom[, 4]

##  [1] 338.486 122.158   3.958  86.351  63.351  61.373  32.078 147.271
##  [9] 141.787  45.020 145.810 349.057 176.029 319.507 234.369   4.852
## [17]   7.595  11.503  25.381  65.699 160.839  31.781 189.733 253.308
## [25]  91.538  90.454  99.737  34.465  68.150  46.326 160.993   9.806
## [33]  20.743  21.650  66.634  54.268  19.845 131.727  22.366   2.636
## [41] 411.160  15.476  14.493 169.054 139.651 376.308 417.209 103.673
## [49]  33.714 116.155  44.934  18.856 154.079  70.603 169.164   7.651
## [57]  93.072 277.494 131.857 121.429 212.444  82.093   6.551

# Get 10 random observations of leafarea:
allom[sample(1:10, 10, replace = TRUE), "leafarea"]

##  [1]   3.958 141.787 122.158 122.158 147.271  32.078   3.958  63.351
##  [9]  63.351 338.486

# Select only 'height' and 'diameter', store in new dataframe:
allomhd <- allom[, c("height", "diameter")]
```

### Using `subset()`

While the above method to index dataframes is very flexible, sometimes it leads to code that is difficult to understand (and it is easy to make mistakes). Consider the `subset` function as a convenient alternative.

With the `subset` function, you can select rows that meet a certain criterion, and columns as well. This example uses the pupae data, see section A.5.

```
# Read data
pupae <- read.csv("pupae.csv")

# Take subset of pupae, ambient temperature treatment and CO2 is 280.
subset(pupae, T_treatment == "ambient" & CO2_treatment == 280)

##    T_treatment CO2_treatment Gender PupalWeight Frass
## 1      ambient           280      0       0.244 1.900
## 2      ambient           280      1       0.319 2.770
## 3      ambient           280      0       0.221    NA
## 4      ambient           280      0       0.280 1.996
## 5      ambient           280      0       0.257 1.069
## 6      ambient           280      1       0.333 2.257
## 7      ambient           280      0       0.275 2.198
## 8      ambient           280      1       0.312 1.873
## 9      ambient           280      0       0.254 2.220
## 10     ambient           280      1       0.356 1.807
## 11     ambient           280      0       0.258 1.877
## 12     ambient           280      0       0.224 1.488
## 13     ambient           280      1       0.383 2.033
## 14     ambient           280     NA       0.344 1.913

# Take subset where Frass is larger than 2.9.  Also, keep only variables
# 'PupalWeight' and 'Frass'.
subset(pupae, Frass > 2.6, select = c(PupalWeight, Frass))

##    PupalWeight Frass
## 2        0.319 2.770
## 18       0.384 2.672
## 20       0.385 2.603
## 25       0.405 3.117
## 29       0.473 2.631
## 37       0.469 2.747

# Note that you don't have to quote the column names here.
```

Let's look at another example, using the cereal data (see section A.6). Here, we introduce `%in%`, which is short for 'is an element of'.

```
# Read data
cereal <- read.csv("cereals.csv")

# What are the Manufacturers in this dataset?
levels(cereal$Manufacturer)

## [1] "A" "G" "K" "N" "P" "Q" "R"


# Take a subset of the data with only 'A', 'N' and 'Q' manufacturers, keep
# only 'Cereal.name' and 'calories'.
cerealsubs <- subset(cereal, Manufacturer %in% c("A", "N", "Q"), select = c(Cereal.name,
    calories))
cerealsubs

##                  Cereal.name calories
## 1                  100%_Bran       70
## 2          100%_Natural_Bran      120
## 11               Cap'n'Crunch      120
## 21      Cream_of_Wheat_(Quick)      100
## 36            Honey_Graham_Ohs      120
## 42                        Life      100
## 44                       Maypo      100
## 55                 Puffed_Rice       50
## 56                Puffed_Wheat       50
## 57          Quaker_Oat_Squares      100
## 58             Quaker_Oatmeal      100
## 64              Shredded_Wheat       80
## 65     Shredded_Wheat_'n'Bran       90
## 66 Shredded_Wheat_spoon_size       90
## 69    Strawberry_Fruit_Wheats       90
```

Another example using `subset` is provided in section 2.1.

## 1.8 Exercises

### 1.8.1 Basic operations with the Cereal data

For this exercise, we will use the Cereal data, see section A.6 for a description of the dataset.

*1.* Read in the dataset, look at the first few rows with `head` and inspect the data types of the variables in the dataframe with `str`.

*2.* Add a new variable to the dataset called 'totalcarb', which is the sum of 'carbo' and 'sugars'.

*3.* How many cereals in the dataframe are 'hot' cereals?

*4.* How many unique manufacturers are included in the dataset?

*5.* Take a subset of the dataframe with only the Manufacturer 'K' (Kellogg's).

*6.* Take a subset of the dataframe of all cereals that have less than 80 calories, AND have more than 20 units of vitamins.

*7.* Take a subset of the dataframe, keep only the variables 'Cereal.name', 'calories' and 'vitamins', and delete all rows where the cereal has zero sugar.

*8.* For one of the above subsets, write a new CSV file to disk using `write.csv`.

*9.* Rename the column 'Manufacturer' to 'Producer'.

# Chapter 2

# Special data types

## 2.1 Working with factors

The *factor* is used to represent qualitative data.

In the `allom` example dataset, the `species` variable is an example of a factor. A factor variable has a number of ’levels’, which are the text values that the factor variable has in the dataset. Often, factors represent treatments of an experimental study.

For example,

```
levels(allom$species)

## [1] "PIMO" "PIPO" "PSME"
```

Shows the three species in this dataset. We can also count the number of rows in the dataframe for each species, like this:

```
table(allom$species)

##
## PIMO PIPO PSME
##   19   22   22
```

Note that the three species are always shown in the order of the `levels` of the factor: when the dataframe was read, these levels were assigned based on alphabetical order. Often, this is not a very logical order, and you may want to rearrange the levels to get more meaningful results.

In our example, let’s shuffle the levels around, using `factor`.

```
allom$species <- factor(allom$species, levels = c("PSME", "PIMO", "PIPO"))
```

Now revisit the commands above, and note that the results are the same, but the order of the `levels` of the `factor` is different.

We can also generate new factors, and add them to the dataframe. This is a common application:

```
# Add a new variable to allom : 'small' when diameter is less than 10,
# 'large' otherwise.
allom$treeSizeClass <- factor(ifelse(allom$diameter < 10, "small", "large"))

# Now, look how many trees fall in each class.  Note that somewhat
# confusingly, 'large' is printed before 'small'.
table(allom$treeSizeClass)
```

```
##
## large small
##    56     7
```

What if we want to add a new factor based on a numeric variable with more than two levels? In that case, we cannot use `ifelse`. Look at this example using `cut`.

```
# The cut function takes a numeric vectors and cuts it into a categorical
# variable.  Continuing the example above, let's make 'small','medium' and
# 'large' tree size classes:
allom$treeSizeClass <- cut(allom$diameter, breaks = c(0, 25, 50, 75), labels = c("small",
    "medium", "large"))

# And the results,
table(allom$treeSizeClass)

##
##  small medium  large
##     22     24     17
```

### Empty factor levels

It is important to realize at this stage how factors are used in R : they are not simply text variables, or 'character strings'. Each unique value of a factor variable is assigned a `level`, which is used every time you summarize your data by the factor variable.

Crucially, even when you delete data, the original factor level is still present. Consider this example,

```
# Read the Pupae data:
pupae <- read.csv("pupae.csv")

# Note that 'T_treatment' (temperature treatment) is a factor with two
# levels, with 37 and 47 observations in total:
table(pupae$T_treatment)

##
##  ambient elevated
##       37       47


# Suppose we decide to keep only the ambient treatment:
pupae_amb <- subset(pupae, T_treatment == "ambient")

# Now, the level is still present, although empty:
table(pupae_amb$T_treatment)

##
##  ambient elevated
##       37        0


# In most cases, this can be quite annoying. Use droplevels to get rid of
# empty levels:
pupae_amb2 <- droplevels(pupae_amb)
```

**Advanced**

In fact, every level of a factor is internally represented by a number (an integer), like the following example shows:

```
# Numeric representation of factors:
as.integer(allom$species)

##  [1] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 2 2 2 2 2 2 2 2 2 2 2 2 2
## [36] 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

# Each of these numbers gets a label from this vector (i.e, '3' gets the
# 3rd element of this vector)
levels(allom$species)

## [1] "PIMO" "PIPO" "PSME"

# So that the following is the same as allom$species (except it is a
# character vector, not a factor)
levels(allom$species)[as.integer(allom$species)]

##  [1] "PSME" "PSME" "PSME" "PSME" "PSME" "PSME" "PSME" "PSME" "PSME" "PSME"
## [11] "PSME" "PSME" "PSME" "PSME" "PSME" "PSME" "PSME" "PSME" "PSME" "PSME"
## [21] "PSME" "PSME" "PIPO" "PIPO" "PIPO" "PIPO" "PIPO" "PIPO" "PIPO" "PIPO"
## [31] "PIPO" "PIPO" "PIPO" "PIPO" "PIPO" "PIPO" "PIPO" "PIPO" "PIPO" "PIPO"
## [41] "PIPO" "PIPO" "PIPO" "PIPO" "PIMO" "PIMO" "PIMO" "PIMO" "PIMO" "PIMO"
## [51] "PIMO" "PIMO" "PIMO" "PIMO" "PIMO" "PIMO" "PIMO" "PIMO" "PIMO" "PIMO"
## [61] "PIMO" "PIMO" "PIMO"
```

## 2.2 Working with missing values

### 2.2.1 Basics

As we saw before, `R` can represent missing values with texttttNA, a special data type that indicates the data is simply Not Available. Many functions can handle missing data, usually in many different ways. For example, we can have the following vector :

```
myvec1 <- c(11, 13, 5, 6, NA, 9)
```

And you want to calculate the mean, you might want to either exclude the missing value (and calculate the mean of the remaining five numbers), or you might want `mean(myvec1)` to fail (produce an error). This last case is useful if you don't expect missing values, and want `R` to only calculate the mean when there are no NA's in the dataset.

The two options are in this Example:

```
# Calculate mean; fails if there are missing values
mean(myvec1)

## [1] NA

# Calculate mean after removing the missing values
mean(myvec1, na.rm = TRUE)

## [1] 8.8
```

Many functions have an argument `na.rm`, or similar. Refer to the help page of the function to learn about the various options (if any) for dealing with missing values. For example, see the help pages `?lm` and `?sd`.

The following examples show more useful tricks with missing values:

```
# Is a value missing? (TRUE or FALSE)
is.na(myvec1)

## [1] FALSE FALSE FALSE FALSE  TRUE FALSE


# Which of the elements of a vector is missing?
which(is.na(myvec1))

## [1] 5


# how many values in a vector are NA?
sum(is.na(myvec1))

## [1] 1
```

In many cases, missing values may be arise when certain operations did not produce the desired result. Consider this example,

```
# A character vector, some of these look like numbers:
myvec <- c("101", "289", "12.3", "abc", "99")

# Convert the vector to numeric:
as.numeric(myvec)

## Warning:  NAs introduced by coercion

## [1] 101.0 289.0  12.3    NA  99.0
```

The warning message 'NAs introduced by coercion' means that missing values were produced by forcing one data type (character) to another (numeric).

**Not A Number**

Another type of missing value is the result of calculations that went wrong, for example:

```
# Attempt to take the logarithm of a negative number:
log(-1)

## Warning:  NaNs produced

## [1] NaN
```

The result is `NaN`, short for Not A Number. If you see this, something went wrong!

Dividing by zero is not usually meaningful, but `R` does not produce a missing value:

```
1000/0

## [1] Inf
```

but produces 'Infinity' instead.

### 2.2.2 Missing values in dataframes

When working with dataframes, often you want to remove missing values for a particular analysis. Let's use the pupae dataset for the following examples. See section A.5 for a description of this dataset.

```
# Read the data
pupae <- read.csv("pupae.csv")

# Look in the summary statement to see if there are missing values:
summary(pupae)
```

```
##    T_treatment CO2_treatment     Gender        PupalWeight
##  ambient :37   Min.   :280   Min.   :0.000   Min.   :0.172
##  elevated:47   1st Qu.:280   1st Qu.:0.000   1st Qu.:0.256
##                Median :400   Median :0.000   Median :0.297
##                Mean   :344   Mean   :0.449   Mean   :0.311
##                3rd Qu.:400   3rd Qu.:1.000   3rd Qu.:0.356
##                Max.   :400   Max.   :1.000   Max.   :0.473
##                              NA's   :6
##      Frass
##  Min.   :0.986
##  1st Qu.:1.515
##  Median :1.818
##  Mean   :1.846
##  3rd Qu.:2.095
##  Max.   :3.117
##  NA's   :1
```

```
# Notice there are 6 NA's (missing values) for Gender, and 1 for Frass.

# Option 1: take subset of data where Gender is not missing:
pupae_subs1 <- subset(pupae, !is.na(Gender))

# Option 2: take subset of data where Frass AND Gender are not missing
pupae_subs2 <- subset(pupae, !is.na(Frass) & !is.na(Gender))

# A more rigorous subset: remove all rows from a dataset where ANY
# variable has a missing value:
pupae_nona <- pupae[complete.cases(pupae), ]
# (Note: an alternative function is na.omit(), but this has some
# side-effects you may want to avoid.)
```

## 2.3 Working with text

### 2.3.1 Basics

As your datasets often have variables that are text only (think of comments, species names, locations and so on), it is useful to learn how to modify, extract, and analyze text-based ('character') variables.

Consider the following simple examples when working with a single character string:

```
# Count number of characters in a bit of text:
sentence <- "Not a very long sentence."
nchar(sentence)
```

```
## [1] 25

# Split the sentence by space:
strsplit(sentence, " ")

## [[1]]
## [1] "Not"       "a"         "very"      "long"      "sentence."

# And count the number of words:
length(strsplit(sentence, " ")[[1]])

## [1] 5

# (Note the [[1]] is needed because strsplit returns a list!)

# Extract the first 3 characters:
substr(sentence, 1, 3)

## [1] "Not"
```

Many examples also work with vectors, for example

```
# Substring all elements of a vector
substr(c("good", "good riddance", "good on ya"), 1, 4)

## [1] "good" "good" "good"

# Number of characters of all elements of a vector
nchar(c("hey", "hi", "how", "ya", "doin"))

## [1] 3 2 3 2 4
```

### 2.3.2 Names, matching and factor levels

Vectors, dataframes and list can all have names (of columns, or elements) that can be used to find rows or columns in your data. We already saw how you can use column names to index a dataframe in Section 1.7.2. Also consider the following useful examples:

```
# Change the names of a dataframe:
hydro <- read.csv("hydro.csv")
names(hydro)  # first print the old names

## [1] "Date"    "storage"

names(hydro) <- c("Date", "Dam_Storage")  # then change the names

# Change only the first name (you can index names() just like you can a
# vector!)
names(hydro)[1] <- "Datum"
```

Sometimes it is useful to find out *which* columns are equal to a particular value:

```
match(c("diameter", "leafarea"), names(allom))

## [1] 2 4
```

Shows that the 2nd and 4th column have those names.

### 2.3.3 Text in dataframes and `grep`

Now, some useful examples when one (or more) or your variables in a dataframe is a character vector.

First note (and this is repeated throughout this manual), that when you read a dataset (with `read.csv` or `read.table` or similar), any variable that `R` cannot convert to numeric *is automatically converted to a factor*.

While this is often useful, it is not always the case. Let's look at a few examples using the `cereal` dataset, described in section A.6.

```
# Read data, tell R to treat the first variable ('Cereal.name') as
# character, not factor
cereal <- read.csv("cereals.csv", stringsAsFactors = FALSE)

# Make sure that the Cereal name is really a character vector:
is.character(cereal$Cereal.name)

## [1] TRUE
```

so far, so good. The following example uses `grep`, a very powerful function. This function (which is also available in other programming languages, Unix, etc.) can also use 'Regular expressions', a very flexible tool for text processing. We show a couple of examples, please read the Wiki page for more information (http://en.wikipedia.org/wiki/Regular_expressions).

```
# Extract cereal names (for convenience).
cerealnames <- cereal$Cereal.name

# Find the cereals that have 'Raisin' in them:
grep("Raisin", cerealnames)

##  [1] 23 45 46 50 52 53 59 60 61 71


# That result just gives you the indices of the vector that have 'Raisin'
# in them.  these are the corresponding names:
cerealnames[grep("Raisin", cerealnames)]

##  [1] "Crispy_Wheat_&_Raisins"
##  [2] "Muesli_Raisins,_Dates,_&_Almonds"
##  [3] "Muesli_Raisins,_Peaches,_&_Pecans"
##  [4] "Nutri-Grain_Almond-Raisin"
##  [5] "Oatmeal_Raisin_Crisp"
##  [6] "Post_Nat._Raisin_Bran"
##  [7] "Raisin_Bran"
##  [8] "Raisin_Nut_Bran"
##  [9] "Raisin_Squares"
## [10] "Total_Raisin_Bran"


# Now find the cereals whose name starts with Raisin:
grep("^Raisin", cerealnames)

## [1] 59 60 61

# The ^ symbol is part of a 'regular expression'.

# Or end with 'Bran'
grep("Bran$", cerealnames)

##  [1]  1  2  3 20 29 53 59 60 65 71
```

As mentioned, `grep` is a very powerful function, so *you may not want to read the help page*. One very useful option is to turn off the case-sensitivity, for example:

```
grep("raisin", cerealnames, ignore.case = TRUE)

##  [1] 23 45 46 50 52 53 59 60 61 71
```

Works just like the earlier example.

Finally, using the above tools, let's add a new variable to the `cereal` dataset that is TRUE when the name of the cereal ends in 'Bran', otherwise it is FALSE.

```
# First add a new variable, that is always FALSE
cereal$BranOrNot <- FALSE

# then set that variable to TRUE whenever cerealname ends in 'Bran'
cereal$BranOrNot[grep("Bran$", cerealnames)] <- TRUE

# Quick summary:
summary(cereal$BranOrNot)

##    Mode   FALSE    TRUE    NA's
## logical      67      10       0
```

## 2.4 Working with logical data

Some data can only take two values : true, or false. `R` has the *logical* data type for this situation, which greatly simplifies dealing with this type of data.

Like factors, logical data are coded by integer numbers (0 = FALSE, 1 = TRUE). Once you know this, some analyses become even easier. Let's look at some examples:

```
# Answers to (in)equalities are always logical:
10 > 5

## [1] TRUE

101 == 100 + 1

## [1] TRUE


# ... or use objects for comparison:
apple <- 2
pear <- 3
apple == pear

## [1] FALSE


# Logical comparisons like thesealso work for vectors, for example:
nums <- c(10, 21, 5, 6, 0, 1, 12)
nums > 5

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


# Find which of the numbers are larger than 5:
which(nums > 5)
```

```
## [1] 1 2 4 7


# Other useful functions are 'any' and 'all': Are any numbers larger than
# 25?
any(nums > 25)

## [1] FALSE

# Are all numbers less than or equal to 10?
all(nums <= 10)

## [1] FALSE


# How many numbers are larger than 5? - Short solution
sum(nums > 5)

## [1] 4

# - Long solution
length(nums[nums > 5])

## [1] 4
```

We have already seen a number of 'logical operators', like == (is equal to), and >(is greater than). See the help page `?Syntax` for a comprehensive list of operators in `R` (including the logical ones).

## 2.5 Working with dates and times

Admittedly, working with dates and times in `R` is somewhat annoying at first. The built-in help files on this subject describe all aspects of this special data type, but do not offer much for the beginning `R` user. This section covers basic operations, that you may need when analyzing and formatting datasets.

### 2.5.1 Reading dates

The built-in `Date` class in `R` is basically an integer number representing the number of days since some 'origin' (typically 1-1-1970, but this does not usually matter for the user). Converting a character string to a date is straightforward if you use the standard order of dates: YYYY-MM-DD. So, for example,

```
as.Date("2008-5-22")

## [1] "2008-05-22"
```

The output here is not interesting, `R` simply prints the date. Because dates are represented as numbers in `R`, we can do basic arithmetic:

```
# Difference between dates.
as.Date("2009-7-1") - as.Date("2008-12-1")

## Time difference of 212 days


# A date, four weeks later:
as.Date("2011-5-12") + 28

## [1] "2011-06-09"
```

Often, the date is not in the standard format as above, but it is possible to convert any reasonable sequence to a `Date` object in `R`. All we have to do is provide a character string to `as.Date` and point out to that function the order of the fields. Here is a list of examples, more fields can be found in the help file `?strptime`.

In each of the following examples, the `format` argument in the `as.Date` function specifies the format of the date in the character string. Each bit starting with % represents a particular field (for example, day of month, month number, month abbreviation, and so on).

```
# Day / month / year
as.Date("31/12/1991", format = "%d/%m/%Y")

## [1] "1991-12-31"


# Day - month - year (but only two digits for the year)
as.Date("17-4-92", format = "%d-%m-%y")

## [1] "1992-04-17"


# Year and day of year ('%j' stands for 'Julian date')
as.Date("2011 121", format = "%Y %j")

## [1] "2011-05-01"
```

Another method to construct date objects is when you do not have a character string as in the above example, but separate numeric variabes for year, month and day. In this case, use the `ISOdate` function:

```
as.Date(ISOdate(2008, 12, 2))

## [1] "2008-12-02"
```

Finally, here is a simple way to find the number of days since you were born.

```
# Today's date (and time) can be found like this,
Sys.time()

## [1] "2012-09-20 15:10:10 EST"


# Converting that to a proper Date, and using your birthdate in the second
# bit, we get:
as.Date(Sys.time()) - as.Date("1976-5-22")

## Time difference of 13270 days
```

### Sequences of dates

The `as.Date` function that we met in the previous section also works with `vectors` of dates, and the Date class can also be part of a dataframe. Let's take a look at the Hydro (see description in Section A.2) dataset, to practice working with more dates.

```
# Read dataset
hydro <- read.csv("Hydro.csv")

# Recall that, by default, anything that cannot be converted to a numeric
# variable is read as a factor.  Convert the Date variable to a character
# string:
hydro$Date <- as.character(hydro$Date)
```

```
# Now convert this to a Date variable.
hydro$Date <- as.Date(hydro$Date, format = "%d/%m/%Y")
```

We now have successfully read in the date variable. The `min` and `max` functions are useful to check the range of dates in the dataset:

```
# Minimum and maximum date (that is, oldest and most recent),
min(hydro$Date)

## [1] "2005-08-08"

max(hydro$Date)

## [1] "2011-08-08"


# ... and length of measurement period:
max(hydro$Date) - min(hydro$Date)

## Time difference of 2191 days
```

If any of the date conversions go wrong, `R` does not print an error, instead, a missing value (`NA`) is produced. You can check if any of the converted dates is `NA` like this:

```
any(is.na(hydro$Date))

## [1] FALSE
```

It is often useful to generate sequences of dates. This is very easy to do:

```
# A series of dates, by day:
seq(from = as.Date("2011-1-1"), to = as.Date("2011-1-10"), by = "day")

##  [1] "2011-01-01" "2011-01-02" "2011-01-03" "2011-01-04" "2011-01-05"
##  [6] "2011-01-06" "2011-01-07" "2011-01-08" "2011-01-09" "2011-01-10"


# Two-weekly dates:
seq(from = as.Date("2011-1-1"), length = 10, by = "2 weeks")

##  [1] "2011-01-01" "2011-01-15" "2011-01-29" "2011-02-12" "2011-02-26"
##  [6] "2011-03-12" "2011-03-26" "2011-04-09" "2011-04-23" "2011-05-07"


# Monthly:
seq(from = as.Date("2011-12-13"), length = 5, by = "months")

## [1] "2011-12-13" "2012-01-13" "2012-02-13" "2012-03-13" "2012-04-13"
```

Finally, the Date class is very handy when plotting. Let's make a simple graph of the Hydro dataset. The following code produces Figure 2.1. Note how the X axis is automatically formatted to display the date in a (fairly) pretty way.

```
with(hydro, plot(Date, storage, type = "l"))
```

### 2.5.2 Date-Time combinations

Things get more complicated when we include the time of day. For dates that include the time, `R` has two special classes called `POSIXct` and `POSIXlt`. We will use both in some examples to illustrate their use and usefulness.

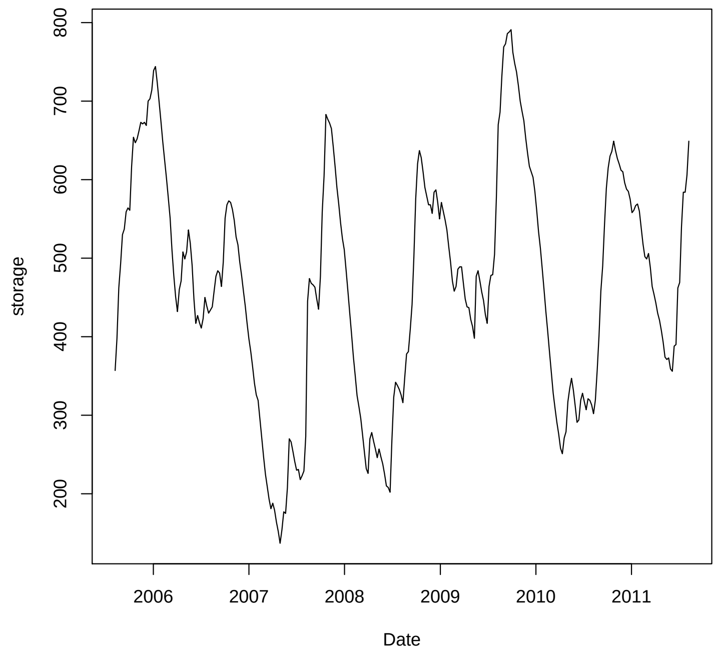


Figure 2.1: A simple plot of the hydro data.

Both POSIX classes represent the time as the number of seconds since the 1st of January, 1970. They also support the use of timezones (for example, to convert from one timezone to another). We will not use this functionality in this manual, as it easily leads to severe problems that are difficult to find and correct.

*Warning:* Always specify the timezone 'GMT' when using the POSIXct class. This ensures that the date/time you read in does not get converted to another timezone.

Let's look at some examples:

```
# The standard format of the POSIXct is YYYY-MM-DD HH:MM:SS (SS is
# optional):
as.POSIXct("2012-9-16 13:05:00", tz = "GMT")

## [1] "2012-09-16 13:05:00 GMT"

# Again, note we specify tz='GMT'. Do this ALWAYS.

# Read two times:
time1 <- as.POSIXct("2008-5-21 9:05", tz = "GMT")
time2 <- as.POSIXct("2012-9-16 13:05:00", tz = "GMT")

# Time difference:
time2 - time1

## Time difference of 1579 days


# Generate a sequence with 30min timestep: (Note that if we don't specify
# the time, it assumes midnight!)
fromtime <- as.POSIXct("2012-6-1", tz = "GMT")
```

```
halfhours <- seq(from = fromtime, length = 12, by = 30 * 60)
# Here, the 'by' field specifies the timestep in seconds.

# As with the Date classes, we can read Date/Times in pretty much any
# reasonable format. See the help page ?strptime for a full list of codes.
# Read in a common date/time format:
as.POSIXct("23-1-89 4:30", format = "%d-%m-%y %H:%M", tz = "GMT")

## [1] "1989-01-23 04:30:00 GMT"
```

Now let's use a real dataset to practice the use of the POSIXct class. We will also introduce the POSIXlt class, which is closely related and has some useful properties.

```
# Read the 2008 met dataset from the HFE.
hfemet <- read.csv("HFEmet2008.csv")

# Convert 'DateTime' to POSIXct class.  The order of the original data is
# MM/DD/YYYY HH:MM
hfemet$DateTime <- as.POSIXct(as.character(hfemet$DateTime), format = "%m/%d/%Y %H:%M",
    tz = "GMT")

# Make sure they all converted OK (if not, NAs would be produced)
any(is.na(hfemet$DateTime))

## [1] FALSE

# FALSE is good here!

# Create an object 'DATS' of class POSIXlt, so we can extract all sorts of
# data:
DATS <- as.POSIXlt(hfemet$DateTime)

# Add day of year (or 'julian date') to the dataframe (Jan 1st=1, and so
# on).
hfemet$DOY <- DATS$yday + 1
# Note we add 1 because of the definition of 'yday', see ?POSIXlt

# Add the hour and minute to the dataframe:
hfemet$hour <- DATS$hour
hfemet$minute <- DATS$min

# And the month (also add 1 so that January=1)
hfemet$month <- DATS$mon + 1

# Now produce a plot of air temperature, for the month of June.
with(subset(hfemet, month == 6), plot(DateTime, Tair, type = "l"))
```

```
# Notice that in the above, an axis with the Date is automatically added.
# To customize this, look at the help page ?axis.POSIXct (especially the
# examples).
```

The last command produced figure 2.2.

*Note* that you cannot store a `POSIXlt` object inside a dataframe! See ?POSIXlt for a list of other variables that are contained in a `POSIXlt` object.

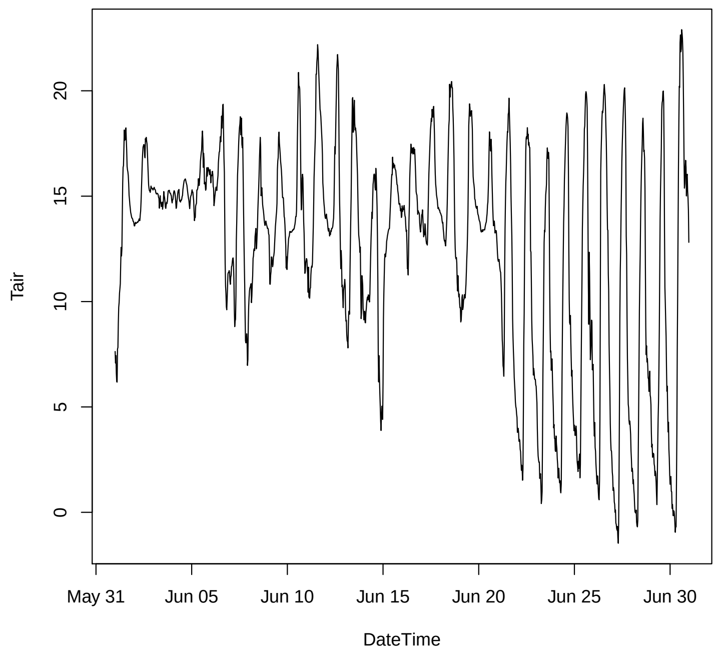


Figure 2.2: Air temperature for June at the HFE

## 2.6 Exercises

### 2.6.1 Flux data

In this exercise, you will practice useful skills with the flux tower dataset. See section A.7 for a description of the dataset.

*1.* Read the dataframe. Rename the first column to 'DateTime'.

*2.* Convert DateTime to a POSIXct class. Beware of the formatting (see `?strptime` for help.)

*3.* Did the above action produce any missing values? Were these already missing in the original dataset?

*4.* Add a variable to the dataset called 'Quality'. This variable should be "bad" when the variable 'ustar' is less than 0.15, and "good" otherwise.

*5.* Look at the 'Rain' column. There are some problems; re-read the data or find another way to display 'NA' whenever the data are bad.

### 2.6.2 Hydro dam

*1.* Start by reading in the data. Change the `class` of the first variable to a `Date` class (See Section 2.5.1).

*2.* Assume that a dangerously low level of the dam is 235 *Gwh*. How many weeks was the dam level equal to or lower than this value?

*3.* Are the successive measurements in the dataset always exactly one week apart? (Hint: use `diff`).

*5. Advanced.* For question *2.*, how many times did `storage` decrease below 235 (regardless of how long it remained below 235)? (Hint: use `diff` and `subset`).

# Chapter 3

# Visualizing data

## 3.1 The R graphics system

The graphics system in R is very flexible; just about every aspect of your plot can be controlled very precisely. However, this brings with it a serious learning curve - especially when you want to produce high quality polished figures for publication. In the following, you will learn to make simple plots, and also to control different aspects of the formatting of the plots. All of the following will use the basic built-in graphics system in R, known as the base graphics system.

## 3.2 Basic scatter plots

It is straightforward to make simple scatter plots in R.

There are two alternative ways to make a plot of two variables X and Y that are contained in a dataframe called dfr (both are used interchangeably):

```r
# Option 1:
with(dfr, plot(X, Y))
# Option 2:
plot(Y ~ X, data = dfr)
```

We will look at various options for customizing basic scatter plots throughout this chapter. Also check out the help page ?plot.default (NOT plot!).

## 3.3 Choosing a plot type

The following table summarizes a few important plot types. Consult their help pages (and especially the examples at the bottom of those pages!).

| Function | Graph type |
|---|---|
| `barplot` | Bar plot, with a dizzying number of options |
| `hist` | Histograms and (relative) frequency diagrams |
| `curve` | Plots curves given some mathematical expression |
| `pie` | Pie charts (for less scientific uses) |
| `boxplot` | Box-and-whisker plots |

An excellent resource for examples of figures made with `R` is the 'R Graph Gallery' (`http://gallery.r-enthusiasts.com/`). All example figures include the code used to produce the plot. Note that most of those plots are quite advanced.

## 3.4 Setting up a plotting device

When running *R* on a Mac OS X system, a new plotting device can be called using quartz() or X11(). Plotting on a Linux system uses `X11()`. `Cairo` functions also work on Mac OS X and Linux systems.

## 3.5 Fine-tuning the formatting of plots

### 3.5.1 A quick example

We'll start with an example using the `allom` data. First, we use the default formatting, and then we change a few aspects of the plot at a time.

We will then explain more detail about many of the settings that we introduce here.

A simple scatter plot, produces figure 3.1.

```
# Read data
allom <- read.csv("allometry.csv")

# Default scatter plot
with(allom, plot(diameter, height, col = species))
```

Now let's make sure the axis ranges start at zero, use a more friendly set of colours, and a different plotting symbol.

This code produces figure 3.2.

```
palette(c("blue", "red", "forestgreen"))
with(allom, plot(diameter, height, col = species, pch = 15, xlim = c(0, 80),
    ylim = c(0, 50)))
```

Notice that we use 'species' (a factor variable) to code the colour of the plotting symbols. More about this later.

For this figure it is useful to have the zero start exactly in the corner (compare origin to previous figure) (with `xaxs` and `yaxs`), and let's make the X-axis and Y-axis labels larger (with `cex.lab`) and print nicer labels (with `xlab` and `ylab`).

Finally, we also add a legend (with the `legend` function.)

This code produces figure 3.3.

```
par(xaxs = "i", yaxs = "i", cex.lab = 1.4)
palette(c("blue", "red", "forestgreen"))
plot(height ~ diameter, col = species, data = allom, pch = 15, xlim = c(0, 80),
    ylim = c(0, 50), xlab = "Diameter (cm)", ylab = "Height (m)")
# Add a legend
legend("topleft", levels(allom$species), pch = 15, col = palette(), title = "Species")
```

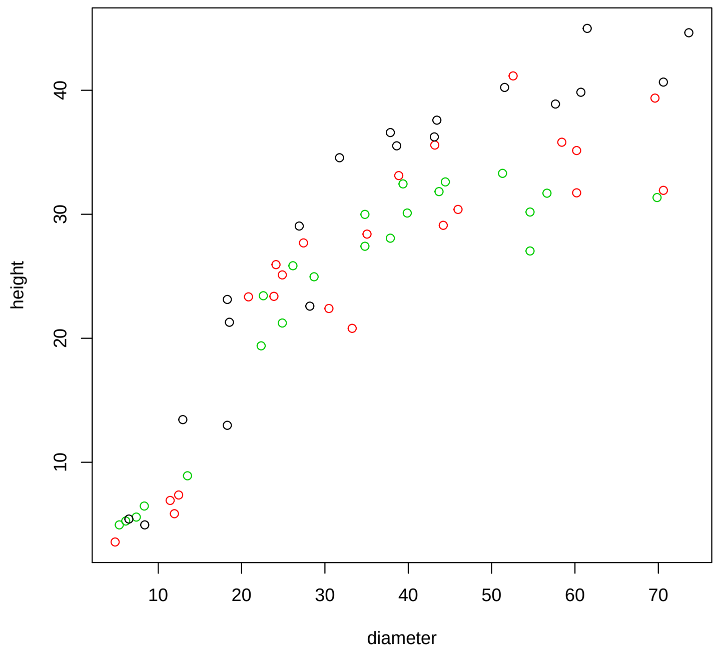


Figure 3.1: Simple scatter plot with default settings

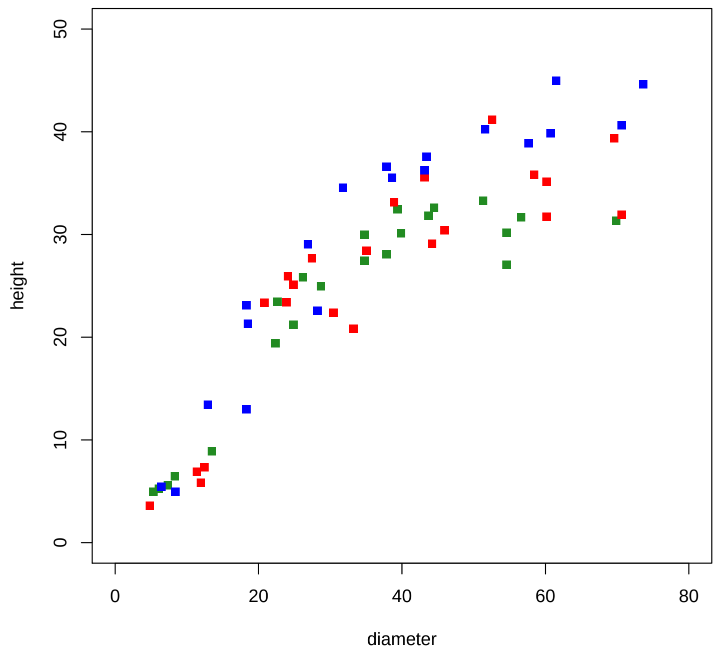


Figure 3.2: Simple scatter plot : changed plotting symbol and X and Y axis ranges.

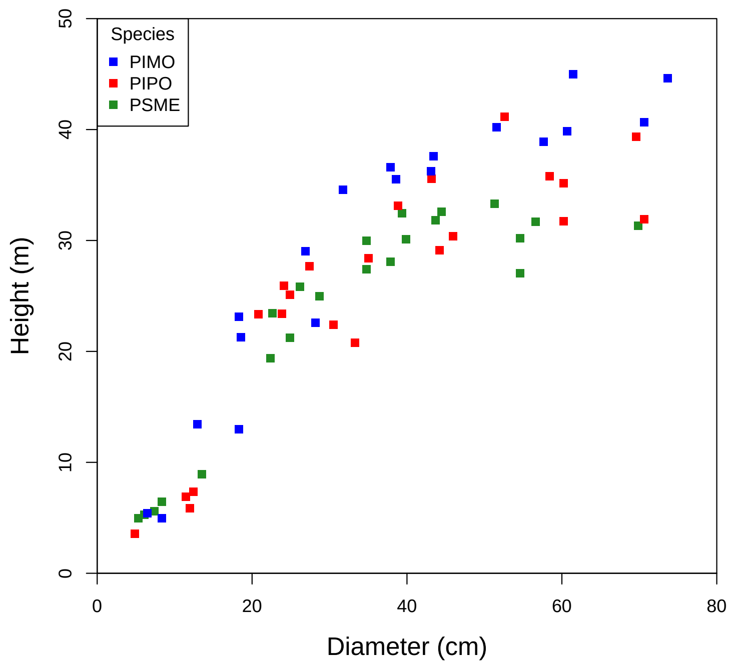


Figure 3.3: Simple scatter plot : many settings customized.

### 3.5.2 Colours, colours

You can change the color of any aspect of your figure, and R has many built-in colours, as well as a number of flexible ways to generate your own.

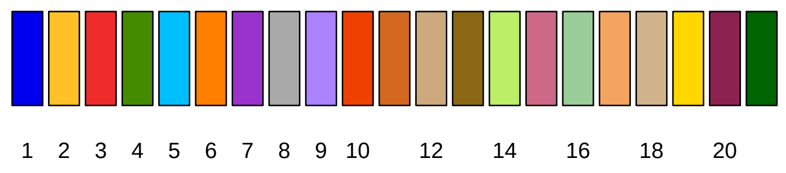
As we saw in the example above, it is quite handy to store a set of nice colors in the `palette()` function, which is automatically used when you use a factor to choose the colors of the plot. As the default set of colours in R is very ugly, do choose your own set of colours before plotting. Also, when you use the following example (resulting graph not shown), it would plot the symbols using the 3rd colour in your palette.

```
plot(x, y, col = 3)
```

To pick one or more of the 657 built-in colours in R, this website is very useful : `http://research.stowers-institute.org/efg/R/Color/Chart/`, *especially note* the link to a PDF at the top, which lists the colours by name.

The following is an example palette, with some hand-picked colours. It also shows one way to plot your current palette.

```
palette(c("blue2", "goldenrod1", "firebrick2", "chartreuse4", "deepskyblue1",
    "darkorange1", "darkorchid3", "darkgrey", "mediumpurple1", "orangered2",
    "chocolate", "burlywood3", "goldenrod4", "darkolivegreen2", "palevioletred3",
    "darkseagreen3", "sandybrown", "tan", "gold", "violetred4", "darkgreen"))

# A simple graph showing the colors.
par(cex.axis = 0.9)
n <- length(palette())
barplot(rep(1, n), col = 1:n, names.arg = 1:n, axes = FALSE)
```

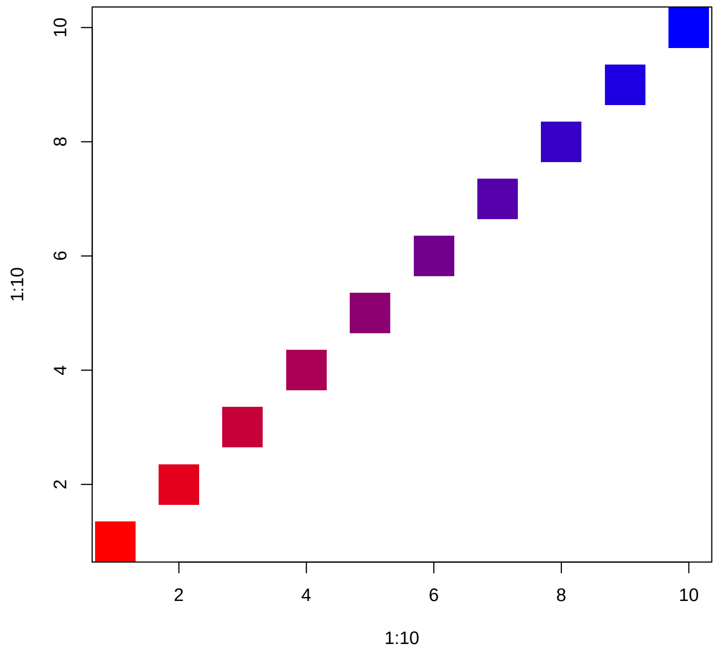


### Ranges of colours

R also has a few built-in options to set a range of colors, for example from light-grey to dark-grey.

A very useful function is `colorRampPalette`, see its help page and the following examples:

```
# Generate a palette function, with colours in between red and blue:
redbluefun <- colorRampPalette(c("red", "blue"))

# The result is a function that returns any number of colours interpolated
# between red and blue.  For example:
palette(redbluefun(10))
plot(1:10, 1:10, pch = 15, cex = 5, col = 1:10)
```



Other functions that are useful are `rainbow`, `heat.colors`, `grey`, and so on (see help page for rainbow for a few more).

### 3.5.3 Finding out about more options

To get the most of plotting in *R*, it is important to get a working knowledge of `par` options, which are used to set graphical parameters. We've used some of these as arguments in the above examples. `pch` sets the types of symbols that are used in the plot; see `points()` for a list of options. `col` sets the colour options for the plot. The size of text and points in the plot area is controlled via the `cex` argument, a character expansion variable that acts as a multiplier of the default value. For text in the margins, character expansion occurs via the `cex.axis`, `cex.lab`,`cex.main`, and `cex.sub` arguments.

### 3.5.4 Changing the layout

### 3.5.5 Figures with multiple axes

Suppose you want to plot more than one variable in one plot, but the units (or ranges) are very different. So, you decide to use two axes. Let's look at an example. For more information on how to deal with the DateTime class, see section 2.5.2.

This code produces figure 3.4.

```
# Read the hfemet data, take a subset of one day.
hfemet <- read.csv("HFEmet2008.csv")
hfemet$DateTime <- as.POSIXct(as.character(hfemet$DateTime), format = "%m/%d/%Y %H:%M",
    tz = "GMT")

# Add the Date :
hfemet$Date <- as.Date(hfemet$DateTime)

# Select one day (a cloudy day in June).
hfemetsubs <- subset(hfemet, Date == as.Date("2008-6-1"))

# Plot Air temperature and PAR (radiation) in one plot.  First we make a
# 'vanilla' plot with the default formatting.
with(hfemetsubs, plot(DateTime, Tair, type = "l"))
par(new = TRUE)
with(hfemetsubs, plot(DateTime, PAR, type = "l", col = "red", axes = FALSE,
    ann = FALSE))
```

```
# The key here is to use par(new=TRUE), it produces the next plot right on
# top of the old one.
```

Next, make the same plot again but with better formatting. Try to figure out what everything means by inspecting the help pages `?par`, `?points`, `?legend`, `?mtext`.

This code produces figure 3.5.

```
par(mar = c(5, 5, 2, 5), cex.lab = 1.2, cex.axis = 0.9)
with(hfemetsubs, plot(DateTime, Tair, type = "l", ylim = c(0, 20), lwd = 2,
    col = "blue", xlab = "Time", ylab = expression(T[air] ~ ~("" ^ "o" * C))))
par(new = TRUE)
with(hfemetsubs, plot(DateTime, PAR, type = "l", col = "red", lwd = 2, ylim = c(0,
    1000), axes = FALSE, ann = FALSE))
axis(4)
mtext(expression(PAR ~ ~(mu * mol ~ m^-2 ~ s^-1)), side = 4, line = 3, cex = 1.2)
legend("topleft", c(expression(T[air]), "PAR"), lwd = 2, col = c("blue", "red"),
    bty = "n")
```

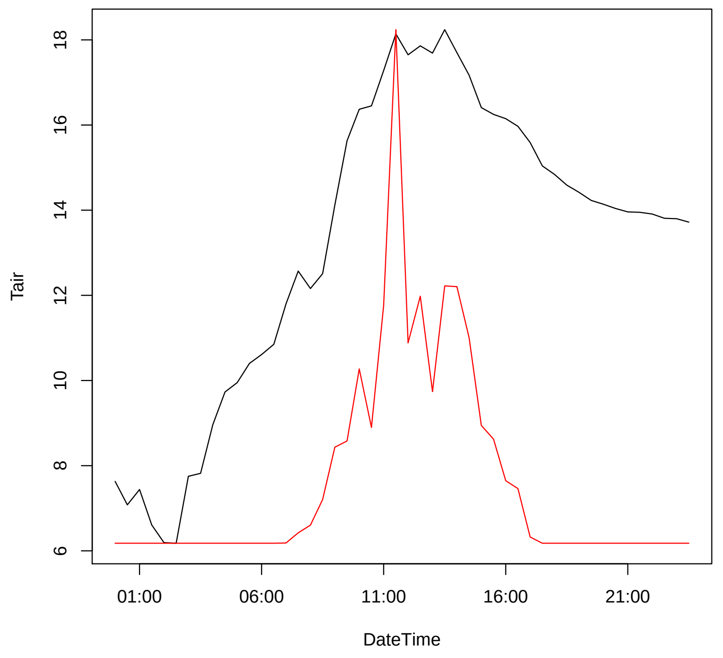


Figure 3.4: A default plot with two axes

## 3.6 Trellis graphics

Trellis graphics are very useful for visualisation of hierarchically structured data. These include multifactorial experimental designs and studies with multiple observations on the same experimental unit. Trellis graphics utilise the structure of a statistical model to easily generate a visual representation of the response according to this structure.

## 3.7 Special plots

### 3.7.1 Histograms

The histogram is used to inspect the distribution of a variable. Let's look at an example using the `vessel` data. In this dataset, it was expected that xylem vessel diameters are smaller in the top of the tree, compared to the bottom. Rather than going straight to statistical analyses, ANOVAs and so on, it is wise to attempt to visualize the data as well as possible.

The following code produces figure 3.6.

```r
# Read vessel data, and make two datasets (one for 'base' data, one for
# 'apex' data).
vessel <- read.csv("vessel.csv")
vesselBase <- subset(vessel, position == "base")
vesselApex <- subset(vessel, position == "apex")

# Simple histograms, default settings.
```

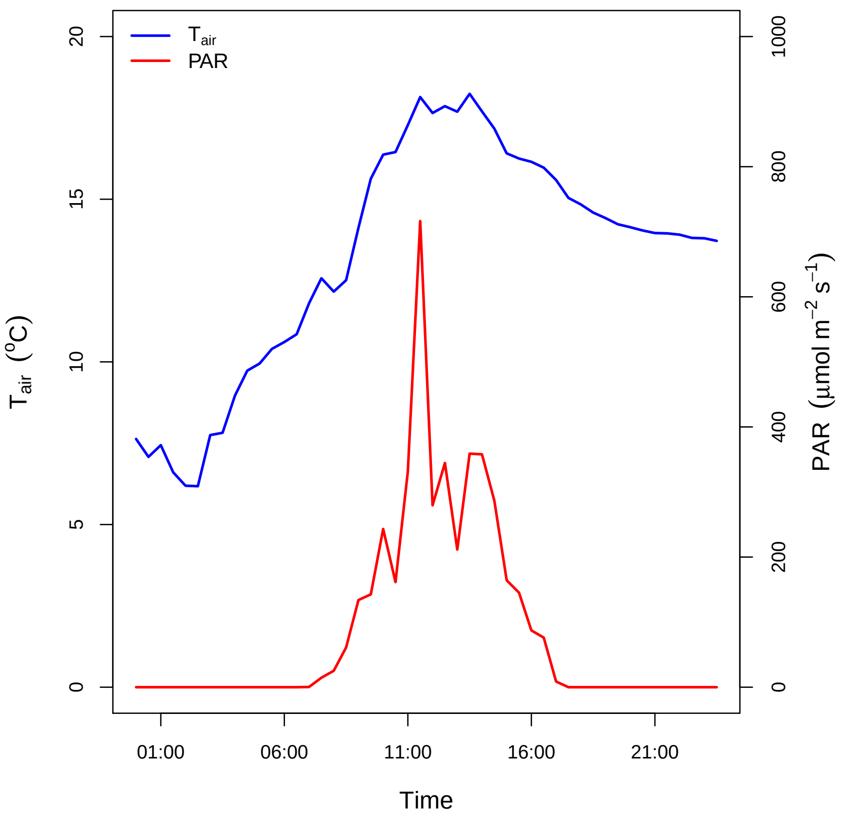


Figure 3.5: A prettified plot with two axes

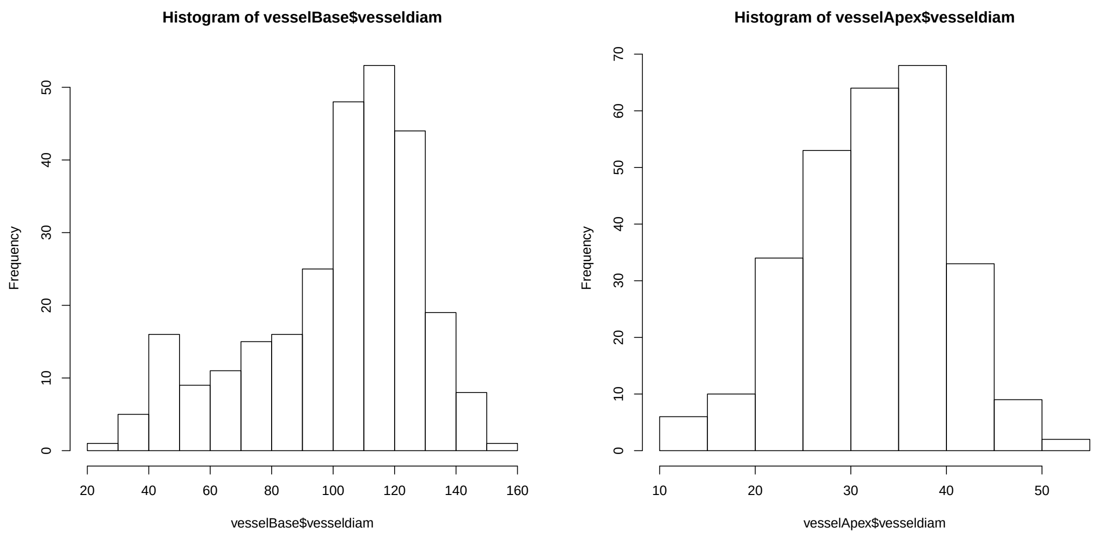


Figure 3.6: Two simple default histograms

```
hist(vesselBase$vesseldiam)
hist(vesselApex$vesseldiam)
```

In this manual, the two figures are shown side by side, to reproduce this take a look at section 3.5.4.

Next, we make the the two histograms again, but with several customized settings, to produce a high quality plot. As before, try to figure out yourself what the options mean by inspecting the help files `?hist`, `?par`.

This code produces figure 3.7. (Make sure to read the vessel data first, as shown in the previous example).

```
# On windows, also include these two lines of code: windows(8,4)
# par(mfrow=c(1,2))
par(mar = c(5, 5, 4, 1), cex.lab = 1.3, xaxs = "i", yaxs = "i")
hist(vesselBase$vesseldiam, main = "Base", col = "darkgrey", xlim = c(0, 160),
    breaks = seq(0, 160, by = 10), xlab = expression(Vessel ~ diameter ~ ~(mu *
        m)), ylab = "Number of vessels")
hist(vesselApex$vesseldiam, main = "Apex", col = "lightgrey", xlim = c(0, 160),
    breaks = seq(0, 160, by = 10), xlab = expression(Vessel ~ diameter ~ ~(mu *
        m)), ylab = "Number of vessels")
```

### 3.7.2 `symbols()`

The `symbols` function is an easy way to pack more information in a simple scatter plot, by providing a way to scale the size of the plotting symbols to another variable in the dataframe. Consider the following example.

This code produces figure 3.8.

```
# Read data
cereal <- read.csv("Cereals.csv")
```

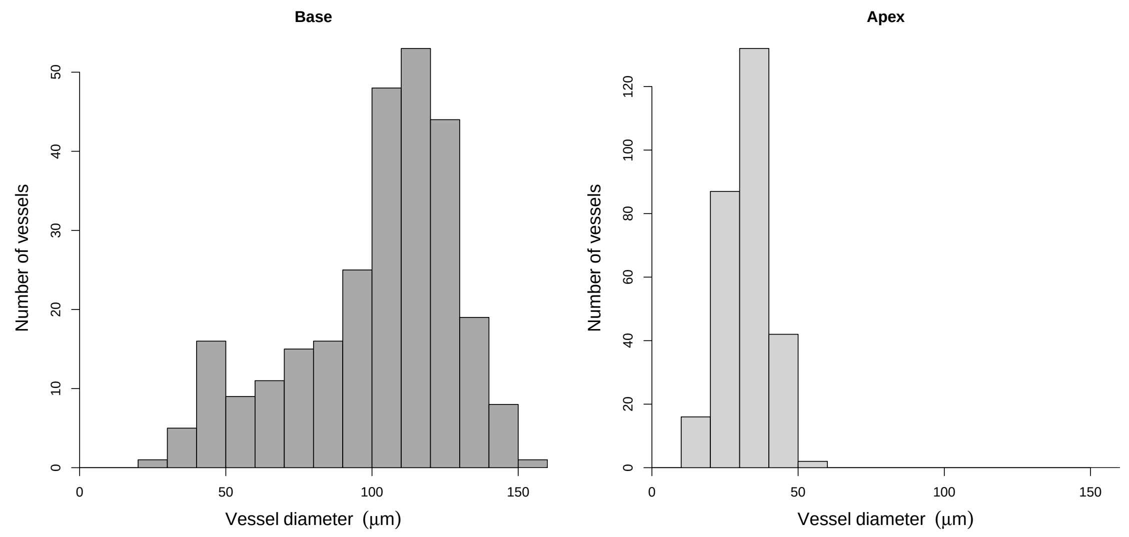


Figure 3.7: Two customized histograms

```
# Choose colours: they will be used in order of the levels of the factor:
levels(cereal$Cold.or.Hot)

## [1] "C" "H"

palette(c("blue", "red"))
# so we choose blue for cold, red for hot.

# Make the plot
with(cereal, symbols(fiber, potass, circles = fat, inches = 0.2, bg = as.factor(Cold.or.Hot),
    xlab = "Fiber content", ylab = "Potassium content"))
```

### 3.7.3 Special packages

There are a large number of packages that specialize in special plot types, take a look at the `plotrix` and `gplots` packages, for example.

Very advanced plots can be constructed with the `ggplot2` package. This package has its own steep learning curve (and its own book and website). For some complex graphs, though, it might be the easiest solution.

## 3.8 Exercises

### 3.8.1 Hydro dam

*1.* Read the hydro data, make sure to convert Date to a proper date class.

*2.* Make a line plot of `storage` versus `Date`, and change the color of the line in the following way: ”forest-green” if storage ¿ 500, ”orange” if storage is between 235 and 500, and ”red” if storage is below 235. (Hint: use `cut`).

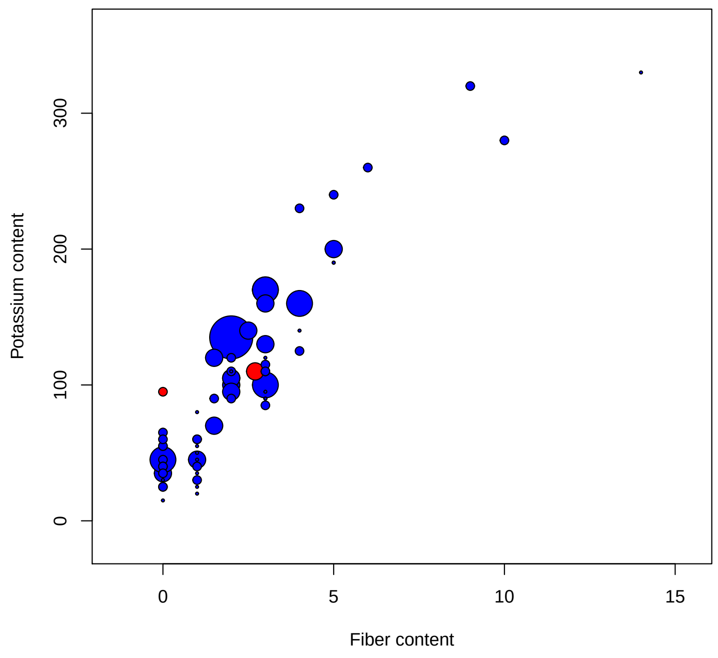


Figure 3.8: Cereal data: symbols size is a function of fat content, colour refers to hot (red) or cold (blue) cereals.

### 3.8.2 Advanced: superimposed histograms

*1.* See the example for the vessel data in section 3.7.1. Use a function from the `epade` package to produce a single plot with both histograms (so that they are superimposed).

# Chapter 4

# Basic statistics

This manual is not a proper *introduction to statistics*. There are many manuals (either printed or on the web) that document the vast array of statistical analyses that can be done with `R`. To get you started, though, I will show a few very common analyses that are easy to do in `R`.

In all of the following, I assume you have a basic understanding of linear regression, Student's t-tests, ANOVA, and confidence intervals for the mean.

## 4.1 Probability Distributions

*When* you studied first year statistics you would have come across several probability distributions. For example, you probably encountered the Binomial distribution as a model for the distribution of the number of *successes* in a sequence of independent trials. Another commonly used discrete distribution is the Poisson, as it is a useful model for many kinds of count data. Of course, the most important distribution of all is the Normal or Gaussian distribution.

`R` provides sets of functions to find densities, cumulative probabilities, quantiles, and even simulate from many important distributions. The function names all consist of a one letter prefix which species the type of function and a stem which specifies the distribution. These are shown in the tables below.

| prefix | function |
|---|---|
| `d` | density |
| `p` | cumulative probabilities |
| `q` | quantiles |
| `r` | simulate |

| stem | distribution |
|---|---|
| `binom` | Binomial |
| `pois` | Poisson |
| `norm` | Normal |
| `t` | Student's t |
| `chisq` | $\chi^2$ |
| `f` | F |

So for example,

```
# Calculate the probability of 3 heads out of 10 tosses of a fair coin
dbinom(3, 10, 0.5)

## [1] 0.1172

# Calculate the the probability of Normal random variate with mean 3 and
# sd 2 is less than or equal to 4
pnorm(4, 3, 2)

## [1] 0.6915
```

```
# Find the t-value that corresponds to a 2.5% right hand tail probability
# with 5 degrees of freedom
qt(0.975, 5)

## [1] 2.571

# Simulate 5 Poisson random variables with mean 3
rpois(5, 3)

## [1] 1 3 3 3 5
```

## 4.2 Descriptive Statistics

Much of this has been covered in earlier chapters but we repeat some of it here for completeness. Descriptive statistics summarise some of the properties of a given data set. Generally, we are interested in measures of location (central tendency) and scale (variance). Let's look at the Pupae dataset (described in section A.5).

```
# Read data
pupae <- read.csv("pupae.csv")
# Extract the weights
weight <- pupae$PupalWeight
# Find the number of observations
length(weight)

## [1] 84

# Find the average (mean) weight
mean(weight)

## [1] 0.311

# Find the Variance
var(weight)

## [1] 0.004114
```

R will compute the sample variance. The standard deviation can be calculated as the square root of the variance, or R provides a function.

```
# Standard Deviation
var.wgt <- var(weight)
sqrt(var.wgt)

## [1] 0.06414

# Standard Deviation
sd(weight)

## [1] 0.06414
```

Robust measures of the location and scale are the median and inter-quartile range; R has functions for these.

```
# median and inter-quartile range
median(weight)

## [1] 0.2975

IQR(weight)
```

```
## [1] 0.09975
```

The median is the 50th percentile or the second quartile. The `quantile` function can compute quartiles and arbitrary percentiles/quantiles.

```
quantile(weight)

##     0%    25%    50%    75%   100%
## 0.1720 0.2562 0.2975 0.3560 0.4730

quantile(weight, probs = seq(0, 1, 0.1))

##     0%    10%    20%    30%    40%    50%    60%    70%    80%    90%
## 0.1720 0.2398 0.2490 0.2674 0.2892 0.2975 0.3230 0.3493 0.3710 0.3910
##   100%
## 0.4730
```

**Missing Values**: All of the above functions will return `NA` if the data contains *any* missing values. However, they also provide an option to remove missing values (`NA`s) before their computations.

```
weightNA <- weight
weightNA[40] <- NA
mean(weightNA)

## [1] NA

mean(weightNA, na.rm = TRUE)

## [1] 0.3113
```

The `summary` function provides a lot of this information in one go. (see later).

```
summary(weight)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.172   0.256   0.298   0.311   0.356   0.473
```

The `moments` package provides higher moments if required, for example, skewness and kurtosis. (This package in not part of the default installation so will need to be installed separately using the menus or `install.packages`)

```
# load the moments package
require(moments)
skewness(weight)

## [1] 0.3852

kurtosis(weight)

## [1] 2.579
```

Sometimes you may wish to calculate descriptive statistics for subgroups in the data. There are a few ways to do this. `tapply` allows you to apply any function to subgroups defined by a second (grouping) variable which is usually a factor. `aggregate` uses the formula concept to collapse dataframes down to summarized versions. For example, to calculate mean weights by gender;

```
# tapply
gender <- pupae$Gender
tapply(weight, gender, mean)

##      0      1
## 0.2725 0.3513

# aggregate
```

```
aggregate(PupalWeight ~ Gender, data = pupae, mean)

##   Gender PupalWeight
## 1      0      0.2725
## 2      1      0.3513
```

`aggregate` can be used for much more powerful summaries, and we show just one example here, to calculate means by temperature treatment and CO2 treatment;

```
aggregate(PupalWeight ~ T_treatment + CO2_treatment, data = pupae, mean)

##   T_treatment CO2_treatment PupalWeight
## 1     ambient           280      0.2900
## 2    elevated           280      0.3049
## 3     ambient           400      0.3420
## 4    elevated           400      0.2990
```

## 4.3 Inference for a single population

*Inference* is about answering questions about population parameters based on a sample. The mean of a random sample from a population is an estimate of the population mean. Since it is a single number it is called a point estimate. It is often desirable to estimate a range within which the population parameter lies with high probability. This is called a confidence interval.

One way to get confidence intervals in R is to use the quantile functions for the relevant distribution. Remember that a $100(1-\alpha)\%$ confidence interval for the mean on Normal population is given by,

$$\bar{x} \pm t_{\alpha/2,n-1}\frac{s}{\sqrt{n}}$$

where $\bar{x}$ is the sample mean, $s$ the sample standard deviation and $n$ is the sample size. $t_{\alpha/2,n-1}$ is the $\alpha/2$ tail point of a $t$-distribution on $n-1$ degrees of freedom. That is, if $T$ has a $t$-distribution on $n-1$ degrees of freedom.

$$P(T \leq t_{\alpha/2,n-1}) = 1 - \alpha/2$$

The R code for this confidence interval can be written as,

```
alpha <- 0.05  # 95% confidence interval
xbar <- mean(weight)
s <- sd(weight)
n <- length(weight)
half.width <- qt(1 - alpha/2, n - 1) * s/sqrt(n)
# Confidence Interval c(xbar - half.width, xbar + half.width) or
# equivalently
xbar + c(-1, 1) * half.width

## [1] 0.2971 0.3249
```

In this code, we have assumed a Normal distribution for the population. You were probably taught that if $n$ is *large*, say $n > 30$, then we can use a Normal approximation. That is, replace `qt(1-alpha/2, n-1)` with `qnorm(1-alpha/2)`, but there is no need, R can use the $t$-distribution for any $n$.

## Hypothesis testing

There may be a reason to ask whether a dataset is consistent with a certain mean. For example, are the pupae weights consistent with a population mean of 0.29? (This is used much less commonly than the two-sample test in the next section.) Again for Normal populations R provides a function to do this called `t.test`. So for a null hypothesis that the population mean is 0.29;

```
t.test(weight, mu = 0.29)

##
##  One Sample t-test
##
## data:  weight
## t = 3.004, df = 83, p-value = 0.00352
## alternative hypothesis: true mean is not equal to 0.29
## 95 percent confidence interval:
##  0.2971 0.3249
## sample estimates:
## mean of x
##     0.311
```

Note that here we get the t-statistic, degrees of freedom ($n-1$) and a p-value for the test, with the specified alternative hypothesis (not equal, ie. two-sided). In addition, `t.test` gives us a 95% confidence interval (compare to the above), and the estimated mean, $\bar{x}$.

We can use `t.test` to get any confidence interval, and/or to do one-sided tests;

```
t.test(weight, mu = 0.29, conf.level = 0.9)

##
##  One Sample t-test
##
## data:  weight
## t = 3.004, df = 83, p-value = 0.00352
## alternative hypothesis: true mean is not equal to 0.29
## 90 percent confidence interval:
##  0.2994 0.3227
## sample estimates:
## mean of x
##     0.311

t.test(weight, mu = 0.29, alternative = "greater", conf.level = 0.9)

##
##  One Sample t-test
##
## data:  weight
## t = 3.004, df = 83, p-value = 0.00176
## alternative hypothesis: true mean is greater than 0.29
## 90 percent confidence interval:
##  0.302   Inf
## sample estimates:
## mean of x
##     0.311
```

Note that the confidence interval is one-sided when the test is one-sided.

The `t.test` is appropriate for data that is approximately normally distributed. You can check this using a histogram or a QQ-plot. (see xxx.) If the data is not very close to a normal distribution then the `t.test` is probably also fine provided the sample is relatively large. If the data is not normal and the sample size is not very large, there are a couple of alternatives, transform the data (often a log transform is enough) or use a *nonparametric* test, in this case the Wilcoxon signed rank test is appropriate. R provides the function `wilcox.test` for the latter, its interface is similar to `t.test` and it tests the hypothesis that the data is symmetric about the hypothesized mu. eg.

```
wilcox.test(weight, mu = 0.29)

##
##  Wilcoxon signed rank test with continuity correction
##
## data:  weight
## V = 2316, p-value = 0.009279
## alternative hypothesis: true location is not equal to 0.29

wilcox.test(weight, mu = 0.29, alternative = "greater")

##
##  Wilcoxon signed rank test with continuity correction
##
## data:  weight
## V = 2316, p-value = 0.004639
## alternative hypothesis: true location is greater than 0.29
```

### Test for proportions

Sometimes you want to test whether observed proportions are consistent with a hypothesized population proportion. For example, to test whether 60 heads out of 100 coin tosses is consistent with the coin being fair we can `prop.test`.

```
prop.test(x = 60, n = 100, p = 0.5)

##
##  1-sample proportions test with continuity correction
##
## data:  60 out of 100, null probability 0.5
## X-squared = 3.61, df = 1, p-value = 0.05743
## alternative hypothesis: true p is not equal to 0.5
## 95 percent confidence interval:
##  0.4970 0.6952
## sample estimates:
##   p
## 0.6

prop.test(60, 100, p = 0.5, alternative = "greater")

##
##  1-sample proportions test with continuity correction
##
## data:  60 out of 100, null probability 0.5
## X-squared = 3.61, df = 1, p-value = 0.02872
## alternative hypothesis: true p is greater than 0.5
## 95 percent confidence interval:
##  0.5128 1.0000
## sample estimates:
##   p
## 0.6
```

(Note: here you might think that I chose to do the one-sided test *greater* because 60 is greater than the expected number of 50 for a fair coin - I didn't! Don't use your data to decide on the null or alternative hypothesis - it renders the p-values meaningless).

## 4.4 Inference for two populations

More commonly, we wish to compare two (or more) populations. The situation with more than two populations will be discussed in a later chapter, but all of the above methods generalize to two populations.

The `pupae` dataset has pupal weights and gender for example. We may wish to compare the weights of males (gender=0) and females (gender=1). There are some with missing gender (gender=NA) so these will be ignored. There are several ways to use `t.test` to compare the pupal weights of males and females. One way assumes the data are in different `R` variables.

```
pupae <- read.csv("pupae.csv")
weight <- pupae$PupalWeight
gender <- pupae$Gender
weight.male <- weight[gender == 0]
weight.female <- weight[gender == 1]
t.test(weight.male, weight.female, var.equal = TRUE)

##
##  Two Sample t-test
##
## data:  weight.male and weight.female
## t = -7.357, df = 76, p-value = 1.854e-10
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.10013 -0.05747
## sample estimates:
## mean of x mean of y
##    0.2725    0.3513
```

The above code runs the *standard* two-sample t-test assuming equal variances. The `alternative=` argument can be used to consider one-sided alternatives.

There is also a *formula* interface to `t.test`. The formula interface is important because we will use it in many other functions, like linear regression and linear modelling. For the `t.test` we can use the formula interface on the extracted variables, or without extracting the variables.

```
t.test(weight ~ gender, var.equal = TRUE)

##
##  Two Sample t-test
##
## data:  weight by gender
## t = -7.357, df = 76, p-value = 1.854e-10
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.10013 -0.05747
## sample estimates:
## mean in group 0 mean in group 1
##          0.2725          0.3513

t.test(PupalWeight ~ Gender, data = pupae, var.equal = TRUE)

##
```

```
##  Two Sample t-test
##
## data:  PupalWeight by Gender
## t = -7.357, df = 76, p-value = 1.854e-10
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.10013 -0.05747
## sample estimates:
## mean in group 0 mean in group 1
##          0.2725          0.3513
```

The latter form is particularly convenient, as the data can be left in the `data.frame` and `R` has automatically ignored the missing values in Gender.

## Paired Data

The `t.test` can also be used when the data are paired, for example, measurements taken before and after some treatment on the same items. The pulse data set contains data like this. So to compare pulse rates before and after exercise, including only those subjects that ran (`Run=1`),

```
pulse <- read.table("ms212.txt", header = TRUE)
pulse.before <- with(pulse, Pulse1[Ran == 1])
pulse.after <- with(pulse, Pulse2[Ran == 1])
t.test(pulse.after, pulse.before, paired = TRUE)

##
##  Paired t-test
##
## data:  pulse.after and pulse.before
## t = 16.53, df = 45, p-value < 2.2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  45.13 57.65
## sample estimates:
## mean of the differences
##                   51.39
```

## Unequal Variances

The default for the two-sample `t.test` is actually to *not* assume equal variances. The theory for this kind of test is quite complex, and the t-test is now only approximate and an adjustment is made to the degrees of freedom called the Satterthwaite or Welch approximation.

```
t.test(PupalWeight ~ Gender, data = pupae)

##
##  Welch Two Sample t-test
##
## data:  PupalWeight by Gender
## t = -7.413, df = 74.63, p-value = 1.587e-10
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.09997 -0.05762
## sample estimates:
```

```
## mean in group 0 mean in group 1
##          0.2725          0.3513
```

Since this form makes less assumptions you could ask why do we ever use the equal variances form? If the assumption is reasonable, then this (equal variances) form will have more power, ie. will reject the null hypothesis more often when it is actually false.

### Assumed Normality

Of course, the two-sample t-test assumes normality of the data, which you can check using a histogram or a QQ-plot, or that the sample sizes are large enough that the *central limit theorem* applies.(Although the paired test assumes only that the differences are normal.) The `wilcox.test` can be used when this assumption is suspect. In the case of two-samples (unpaired) the test used is called the Wilcoxon rank sum test (also known as the Mann-Whitney test).

```
wilcox.test(pulse.after, pulse.before, paired = TRUE, exact = FALSE)

##
##  Wilcoxon signed rank test with continuity correction
##
## data:  pulse.after and pulse.before
## V = 1081, p-value = 3.624e-09
## alternative hypothesis: true location shift is not equal to 0

wilcox.test(PupalWeight ~ Gender, data = pupae, exact = FALSE)

##
##  Wilcoxon rank sum test with continuity correction
##
## data:  PupalWeight by Gender
## W = 152.5, p-value = 1.704e-09
## alternative hypothesis: true location shift is not equal to 0
```

## 4.5 Simple Linear regression

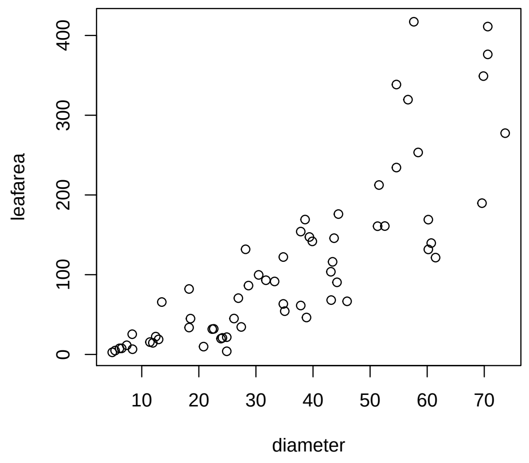
You can fit linear models of varying complexity with the `lm` function. The simplest model is a straight-line relationship between an $x$ and a $y$ variable. In this situation, the assumption is that the $y$-variable (the response) is a linear function of the $x$-variable (the independent variable), plus random noise or measurement error. For the simplest case, both $x$ and $y$ are assumed to be continuous valued variables. In statistical notation we write this as,

$$y = \alpha + \beta x + \varepsilon \tag{4.1}$$

Here $\alpha$ and $\beta$ are (population) parameters that need to be estimate. There is another parameter; $\sigma$, the standard deviation of the noise or measurement error. Note that in simple linear regression this is assumed to be constant (does not depend on $x$).

Lets look at the allometry data;

```
# Read data
allom <- read.csv("Allometry.csv")
plot(leafarea ~ diameter, data = allom)
```



We can see from this graph that leaf area generally increases with tree diameter. So we can use `lm` to estimate the parameters in equation 4.1. We call this fitting the model.

```
# Fit linear regression of 'leafarea' on 'diameter', Results are stored in
# an object called model
model <- lm(leafarea ~ diameter, data = allom)
# Print a summary of the regression:
summary(model)
```

```
##
## Call:
## lm(formula = leafarea ~ diameter, data = allom)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -112.1  -40.5    4.7   30.3  201.4
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -51.324     15.975   -3.21   0.0021 **
## diameter       4.633      0.395   11.72   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 60.2 on 61 degrees of freedom
## Multiple R-squared: 0.692,Adjusted R-squared: 0.687
## F-statistic:  137 on 1 and 61 DF,  p-value: <2e-16
```

```
# Or just the coefficients (intercept and slope):
coef(model)
```

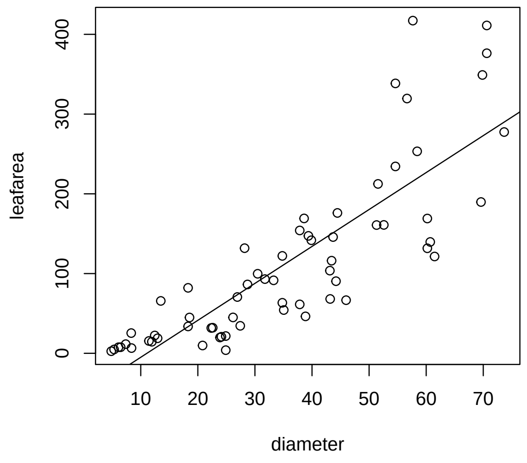
```
## (Intercept)    diameter
##     -51.324       4.633
```

`lm` uses the formula interface that we discussed earlier. The function `summary` prints a lot of information about the fit. In this case, it shows that the intercept is -51.3245, this is the predicted leaf area for a tree with diameter of zero - not very useful in this case. It also has a standard error and a t-statistic for this intercept, along with a p-value which shows that the intercept is significantly different from zero. The second line in the coefficients table shows the slope is 4.6333, and that this is highly significantly different from zero.

In addition, we have the `Residual standard error` of 60.2, which is an estimate of $\sigma$, and an `R-squared` of 0.692 (which is the correlation squared). Finally, an `F-statistic` says whether the overall fit is significant, which in this case, is the same as the test for $\beta$ (the F-statistic is the square of the t-statistic in this case).

It is straight-forward to add the regression line to an existing plot.

```
plot(leafarea ~ diameter, data = allom)
abline(model)
```

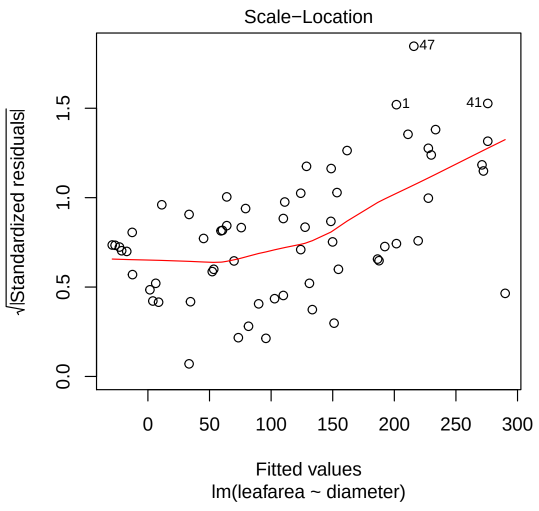


## Diagnostic plots

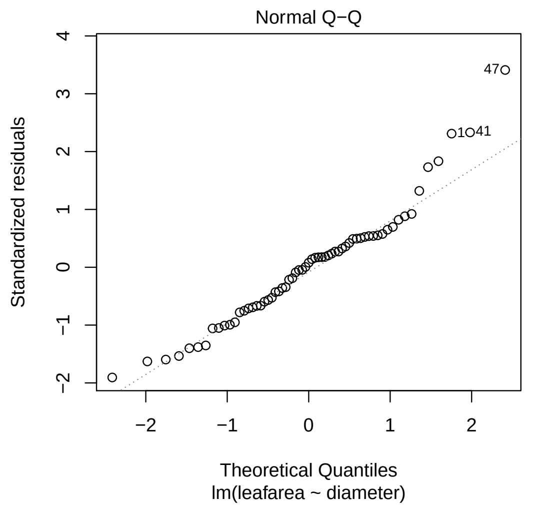
`R` provides a lot of ways to examine how well a model fits the data. These are usually based on the residuals, the difference between the
$haty = \hat{\alpha} + \hat{\beta}x$ fitted values and the actual data points. (Incidentally, the fitted values and residuals can be extracted using `fitted(model)` and `residuals(model)` respectively.

The two simplest and most useful diagnostic plots are the scale-location plot and the QQ-plot.

```
plot(model, which = 3)
```



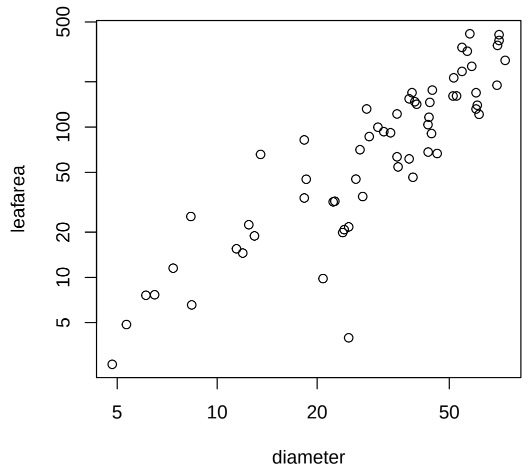
```
plot(model, which = 2)
```



The scale-location plot, has the square root of the *standardized* residuals against the fitted values. In an ideal situation, there should be no structure in this plot. Any curvature indicates that the model maybe under or over-fitting, and a general spread-out (or contracting) from left to right indicates heteroscadasticity (non-constant variance). The QQ-plot enables us to check for departures from normality. Ideally, the standardized residuals should lie on a straight line.

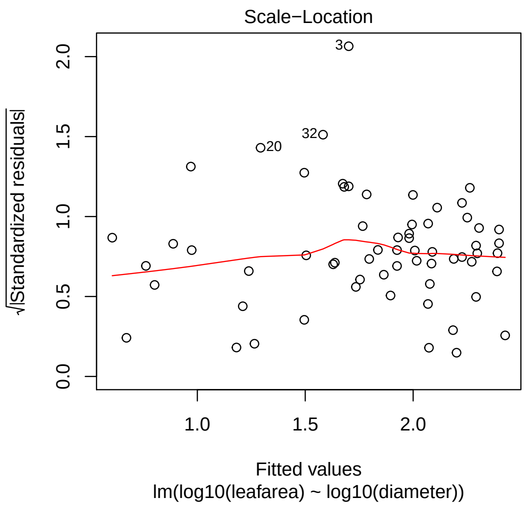
In this case, there is some evidence of heteroscadasticity, and possibly curvature. First of all, lets plot the data on a log-log scale.

```
plot(leafarea ~ diameter, data = allom, log = "xy")
```

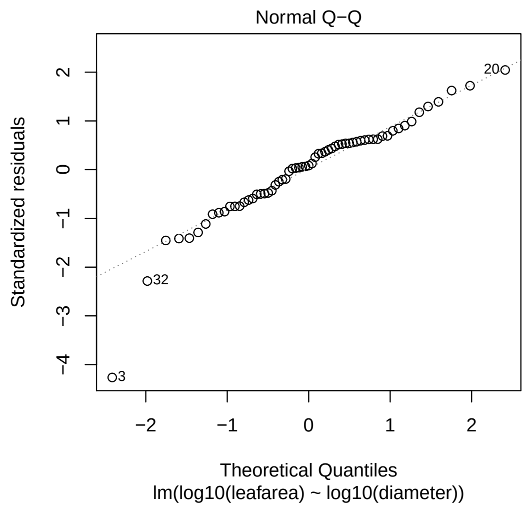


This looks much more constant variance, so we go ahead and refit the model to log transformed variables.

```
model <- lm(log10(leafarea) ~ log10(diameter), data = allom)
summary(model)

##
## Call:
## lm(formula = log10(leafarea) ~ log10(diameter), data = allom)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -1.1038 -0.1415  0.0216  0.1543  0.5245
##
## Coefficients:
##                 Estimate Std. Error t value Pr(>|t|)
## (Intercept)       -0.447      0.160   -2.79    0.007 **
## log10(diameter)    1.539      0.107   14.39   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.261 on 61 degrees of freedom
## Multiple R-squared: 0.772,Adjusted R-squared: 0.769
## F-statistic:  207 on 1 and 61 DF,  p-value: <2e-16

plot(model, which = 3)
```



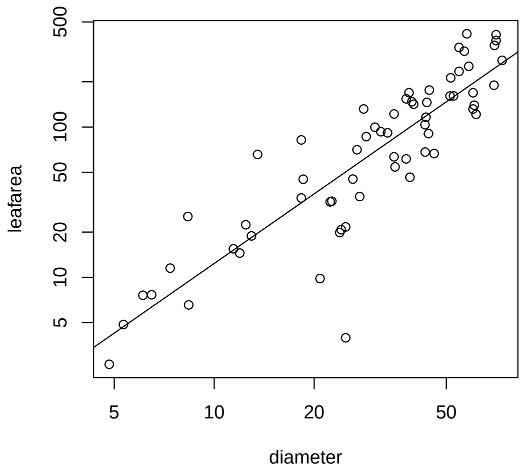
```
plot(model, which = 2)
```



The diagnostic plots look much better, except for a couple of points, at the lower left corner of the QQ-plot. These outliers have been marked, with their row number from the dataframe.

Note that the `abline` function will only work as intended (shown before) on the log-log plot if we use log to the base 10 (`log10`), in the model fit.

```
plot(leafarea ~ diameter, data = allom, log = "xy")
abline(model)
```



We will return to linear models in a later chapter.

# Chapter 5

# Linear modelling

## 5.1 One-way ANOVA

Remember that in chapter 4 we learnt how to compare two means using a two sample t-test. In the simplest case, we assumed that the two samples came from populations with the same variance. One-way ANOVA[1] is concerned with comparing means across more than two populations. We will not go into the theory here, but basically we compare the variation amongst the group means to the variation within the groups using an F-statistic. R provides two ways to do this aov and lm. We will focus exclusively on the latter as it can be generalized to other models more easily.

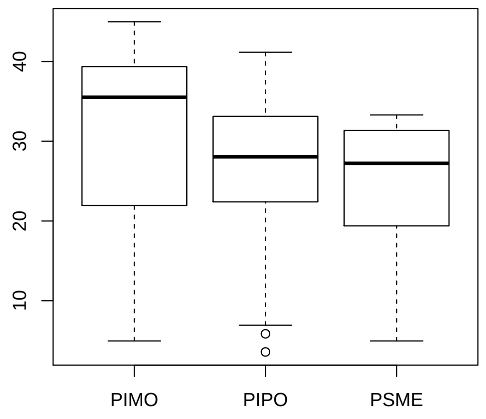
To use lm for an ANOVA, we need a data frame containing the (continuous) response variable and a factor variable that defines the groups. For example, in the Allometry dataset, the species variable is a factor that defines the species groups. We can compute (for example) the mean heights by species

```
allom <- read.csv("Allometry.csv")
aggregate(height ~ species, data = allom, mean)

##   species height
## 1    PIMO  29.61
## 2    PIPO  25.64
## 3    PSME  23.28
```

We might want to ask, does the mean height vary by species? Firstly, a graphical summary of the data is always useful. For this situation, I prefer box plots.

```
boxplot(height ~ species, data = allom)
```

[1] ANOVA = Analysis Of Variance



Doesn't look like there is much difference there, but lets do the test anyway. First we fit a linear model.

```
model <- lm(height ~ species, data = allom)
print(model)

##
## Call:
## lm(formula = height ~ species, data = allom)
##
## Coefficients:
## (Intercept)  speciesPIPO  speciesPSME
##       29.61        -3.97        -6.33
```

Notice the coefficients, these represent something called *contrasts*. In this case, Intercept represents the mean of the *first* species, PIMO. The other two coefficients are the differences between each species and the first.

We can get more details using summary.

```
summary(model)

##
## Call:
## lm(formula = height ~ species, data = allom)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -24.66  -5.66   3.76   8.25  15.52
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)    29.61       2.59   11.42   <2e-16 ***
## speciesPIPO    -3.97       3.54   -1.12    0.267
## speciesPSME    -6.33       3.54   -1.79    0.079 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 11.3 on 60 degrees of freedom
## Multiple R-squared: 0.0512,Adjusted R-squared: 0.0196
## F-statistic: 1.62 on 2 and 60 DF,  p-value: 0.207
```

This gives use a t-statistic (and p-value) for each coefficient, compared to zero. Not surprisingly the Intercept is significantly different from zero. The other coefficients are not really significant, although the PSME coefficient could be. It is not really considered a good way to look for significant results using multiple t-tests. If there are a lot of groups then there is a problem called multiple testing.

At the end of the summary print out, is the F-statistic, and a p-value. This tells us whether the whole model is significant. In this case, it is comparing to model with three coefficients to a model that just has the same mean for all groups. This corresponds to the ANOVA. In this case, it is not significant - ie. there is no evidence of different means for each group.

## 5.2 Two-way ANOVA

Sometimes there a two (or more) *treatment* factors. The age and memory dataset (see section A.10 in the appendix) has the number of words remembered from a list for two age groups and five memory techniques. (This data is balanced so we could use `aov`, but will use `lm` for consistency.) Below we show a table of counts for each of the combinations, and fit a linear model of *main effects*.

```
memory <- read.table("eysenck.txt", header = TRUE)
xtabs(~Age + Process, data = memory)

##          Process
## Age       Adjective Counting Imagery Intentional Rhyming
##   Older          10       10      10          10      10
##   Younger        10       10      10          10      10

model1 <- lm(Words ~ Age + Process, data = memory)
summary(model1)

##
## Call:
## lm(formula = Words ~ Age + Process, data = memory)
##
## Residuals:
##    Min     1Q Median     3Q    Max
##  -9.10  -2.23  -0.25   1.80   9.05
##
## Coefficients:
##                    Estimate Std. Error t value Pr(>|t|)
## (Intercept)          11.350      0.763   14.87  < 2e-16 ***
## AgeYounger            3.100      0.623    4.97  2.9e-06 ***
## ProcessCounting      -6.150      0.985   -6.24  1.2e-08 ***
## ProcessImagery        2.600      0.985    2.64   0.0097 **
## ProcessIntentional    2.750      0.985    2.79   0.0064 **
## ProcessRhyming       -5.650      0.985   -5.73  1.2e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.12 on 94 degrees of freedom
## Multiple R-squared: 0.658,Adjusted R-squared: 0.64
## F-statistic: 36.2 on 5 and 94 DF,  p-value: <2e-16
```

The summary here gives us individual t-statistics for each coefficient. The defaults have used Age group "Older" as the base, and the "Adjective" method as the base for Process. So the other coefficients are relative to the ”Older/Adjective” group. The F-statistic at the end is for the overall model - ie. is the model statistically better than a model that just has a mean count.

If we want to see whether Age has an effect and/or Process has an effect we need F-statistics for these *terms*. We can use `anova` for this.

```
anova(model1)

## Analysis of Variance Table
##
## Response: Words
##           Df Sum Sq Mean Sq F value  Pr(>F)
## Age        1    240     240    24.8 2.9e-06 ***
## Process    4   1515     379    39.0 < 2e-16 ***
## Residuals 94    913      10
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

In this form, the F-statistic is formed for testing significance of the addition of each term sequentially, in the order specified in the model. That is, adding the Age term to a model with an overall mean only, then adding Process to this model. Other software sometimes uses a different approach, namely to compare models by dropping terms one at a time. We can do this in `R` with `drop1`.

```
drop1(model1, test = "F")

## Single term deletions
##
## Model:
## Words ~ Age + Process
##         Df Sum of Sq  RSS AIC F value  Pr(>F)
## <none>                913 233
## Age      1       240 1153 254    24.8 2.9e-06 ***
## Process  4      1515 2428 323    39.0 < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

For a (balanced) design like this it makes no difference!

## Interactions

A more important question when we have more than one factor/treatment in an experiment is whether treatments interact. For example, is the pattern of Process effects different for the two Age groups, or just additive. In `R` we use more complex formulas for this kind of model. An interaction is indicated using a ”:”, and there is a short-hand for *main effects and interaction* using a ”*”.

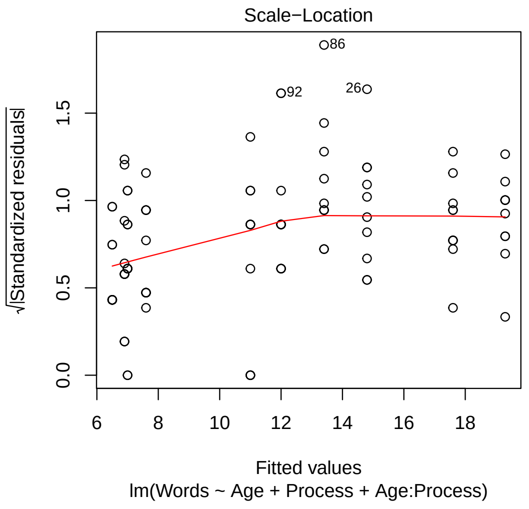
```
### These models are the same
model2 <- lm(Words ~ Age + Process + Age:Process, data = memory)
model2.1 <- lm(Words ~ Age * Process, data = memory)
anova(model2)

## Analysis of Variance Table
##
## Response: Words
##             Df Sum Sq Mean Sq F value  Pr(>F)
## Age          1    240     240   29.94   4e-07 ***
## Process      4   1515     379   47.19 < 2e-16 ***
## Age:Process  4    190      48    5.93 0.00028 ***
## Residuals   90    722       8
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The ANOVA shows that the interaction here is very significant.

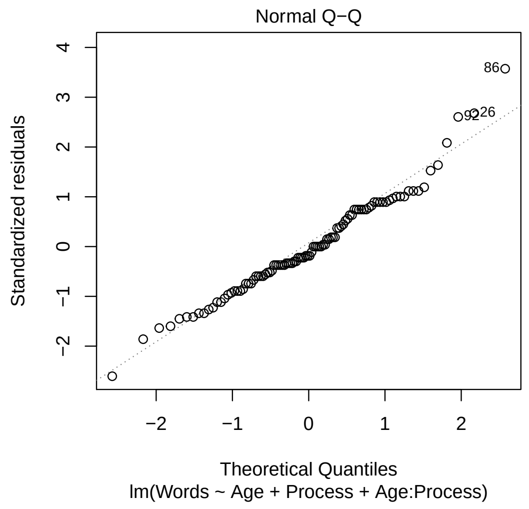
## Diagnostics

ANOVA assumes Normality, and we should check this by looking at residuals etc.

```
plot(model2, which = 3)
```



```
plot(model2, which = 2)
```

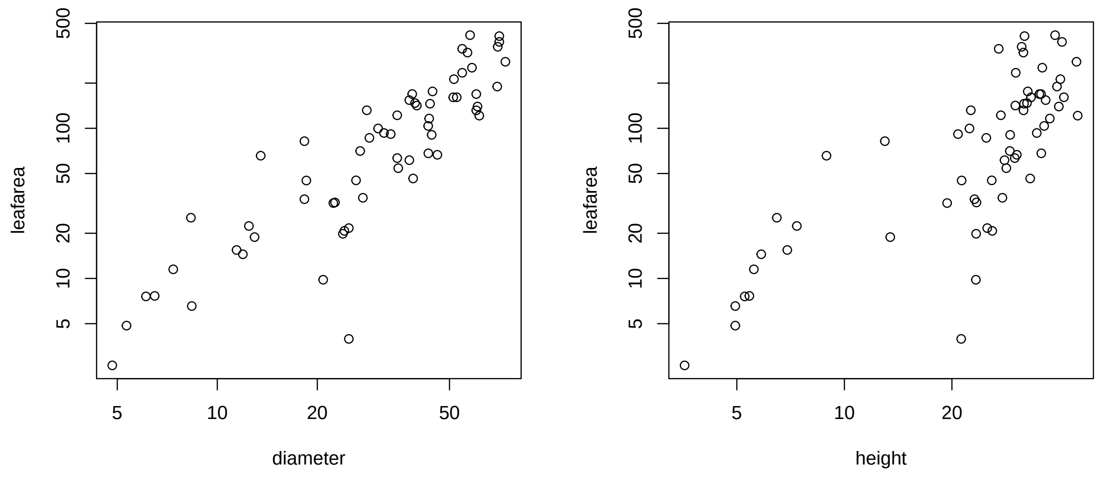


The QQ-plot shows some slight non-normality in the upper right. Taking logs doesn't really improve this problem, which probably due to the Words being counts. We will return to this later.

## 5.3 Multiple linear regression

In chapter 4 we looked at simple linear regression. This is a way to study the relationship between a continuous response and a continuous independent variable or *predictor*. We can use multiple regression to study the relationship between a response an more than one independent variable. For example, returning to the Allometry dataset, we might consider that leaf area depends on both tree diameter and height.

```
plot(leafarea ~ diameter + height, data = allom, log = "xy")
```



These plots are log-log plots, and it looks like there is a relationship. However, since diameter and height are probably related we need to formally model things to understand what is going on.

In statistical notation, we are now considering a model that looks like In statistical notation we write this as,

$$y = \alpha + \beta_1 x_1 + \beta_2 x_2 + \varepsilon \tag{5.1}$$

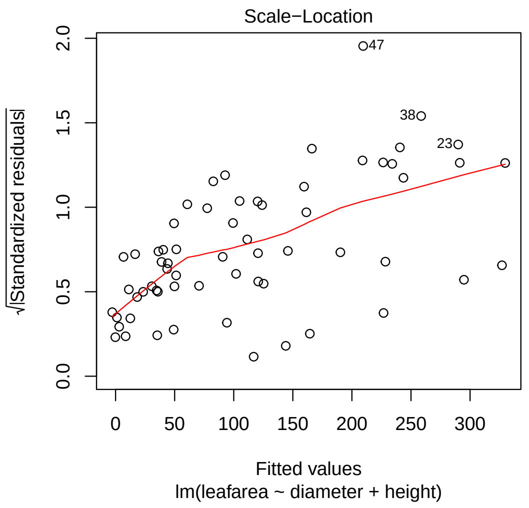
and trying to make inferences about the *two* slopes $\beta_1$ and $\beta_2$.

Again we can do this in R using the *enormously* useful lm function! First lets use the original scale and do some diagnostic plots.

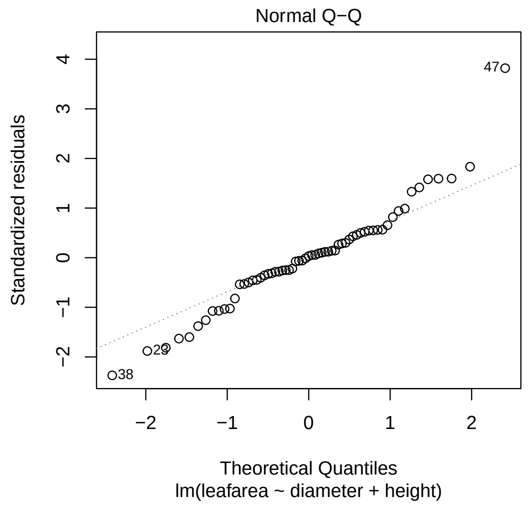
```
fit <- lm(leafarea ~ diameter + height, data = allom)
summary(fit)

##
## Call:
## lm(formula = leafarea ~ diameter + height, data = allom)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -126.89  -24.43    1.78   27.68  207.68
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -17.876     17.639   -1.01   0.3149
## diameter       6.914      0.756    9.14  5.7e-13 ***
## height        -4.403      1.279   -3.44   0.0011 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
##
## Residual standard error: 55.4 on 60 degrees of freedom
## Multiple R-squared: 0.743,Adjusted R-squared: 0.734
## F-statistic: 86.8 on 2 and 60 DF,  p-value: <2e-16
```

```
plot(fit, which = 3)
```



```
plot(fit, which = 2)
```

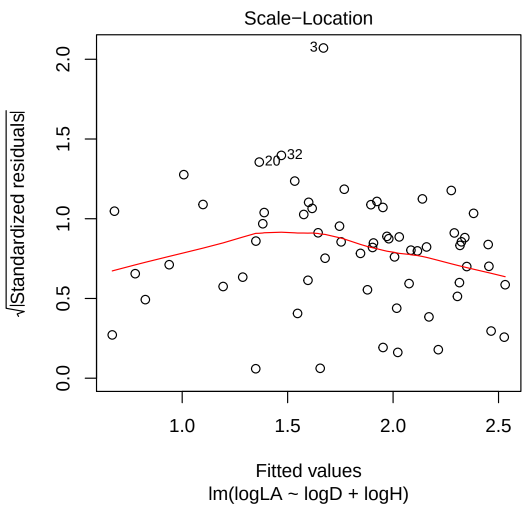


The summary shows that both coefficients (*beta*'s) are significant, although diameter is more so, but the diagnostic plots are not that nice. The scale-location plot shows a somewhat increasing trend - ie the variance is not constant, and the QQ-plot shows some departures. We probably need a log transform again, but this time lets add it to the data frame.

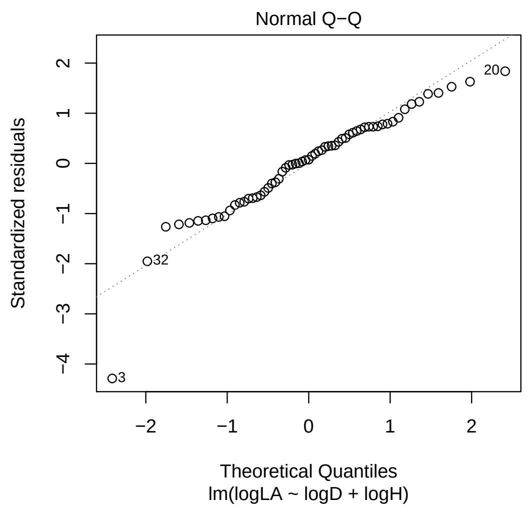
```
allom <- base::within(allom, {
    logLA = log10(leafarea)
    logD = log10(diameter)
    logH = log10(height)
})
fit <- lm(logLA ~ logD + logH, data = allom)
summary(fit)
```

```
##
## Call:
## lm(formula = logLA ~ logD + logH, data = allom)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -1.0723 -0.1701  0.0176  0.1735  0.4521
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   -0.413      0.155   -2.66    0.010 *
## logD           2.190      0.304    7.21  1.1e-09 ***
## logH          -0.734      0.322   -2.28    0.026 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.253 on 60 degrees of freedom
## Multiple R-squared: 0.79,Adjusted R-squared: 0.783
## F-statistic:  113 on 2 and 60 DF,  p-value: <2e-16
```

```
plot(fit, which = 3)
```



```
plot(fit, which = 2)
```



The coefficients (slopes/*beta*s) are still significant and the diagnostic plots are better, although the QQ-plot is affected by some outliers.

Of course, we are not restricted to two independent variables, and we can have interactions. In this case, interactions correspond basically to creating a new variable, which is the product of the old. For example,

```
fit <- lm(logLA ~ logD * logH, data = allom)
summary(fit)

##
## Call:
## lm(formula = logLA ~ logD * logH, data = allom)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -1.0724 -0.1701  0.0178  0.1735  0.4519
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.41519    0.64371   -0.65   0.5214
## logD         2.19222    0.75013    2.92   0.0049 **
## logH        -0.73309    0.51042   -1.44   0.1562
## logD:logH   -0.00135    0.44214    0.00   0.9976
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.255 on 59 degrees of freedom
## Multiple R-squared: 0.79,Adjusted R-squared: 0.78
## F-statistic: 74.2 on 3 and 59 DF,  p-value: <2e-16
```

In this case, the interaction adds nothing.

## 5.4 Linear Models

So far we have looked at ANOVA, including two-way anova where a response variable is modeled as dependent on two treatments or factors, and *regression* including multiple linear regression where a response variable is modeled as dependent on two continuous variables. Of course, these are just special cases of the *linear model*.

We will avoid discussing the mathematical formulation as this is difficult to do without using matrices. However, we can use the concepts from the previous sections in R formulas, and the *all powerful* function lm to fit quite complex models.

Returning to the allometry data set. We might expect that leaf area depends also on species, which is a three level factor;

```
fit <- lm(logLA ~ species + logD + logH, data = allom)
summary(fit)

##
## Call:
## lm(formula = logLA ~ species + logD + logH, data = allom)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -1.1027 -0.0963 -0.0001  0.1381  0.3850
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -0.2800     0.1495   -1.87  0.06609 .
## speciesPIPO  -0.2922     0.0730   -4.00  0.00018 ***
## speciesPSME  -0.0910     0.0725   -1.25  0.21496
## logD          2.4434     0.2799    8.73  3.7e-12 ***
## logH         -1.0097     0.2987   -3.38  0.00130 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.225 on 58 degrees of freedom
## Multiple R-squared: 0.839,Adjusted R-squared: 0.828
## F-statistic: 75.5 on 4 and 58 DF,  p-value: <2e-16
```

The summary of this fit shows that the PIPO species differs markedly from the base level species which in this case was PIMO. However, PSME does not differ. The linear effects in log diameter and log height remain significant.

An ANOVA table will tell us whether adding species improves the model overall.

```
anova(fit)

## Analysis of Variance Table
##
## Response: logLA
##              Df Sum Sq Mean Sq F value  Pr(>F)
## species       2   0.54    0.27    5.84 0.00507 **
## logD          1  14.20   14.20  305.64 < 2e-16 ***
## logH          1   0.58    0.58   12.47 0.00085 ***
## species:logD  2   0.06    0.03    0.61 0.54494
## species:logH  2   0.38    0.19    4.04 0.02322 *
## Residuals    54   2.51    0.05
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

We see that is does.

Perhaps the slopes on log diameter etc. vary by species. This is an interaction between a factor and a continuous variable. We can fit this as;

```
fit <- lm(logLA ~ species * logD + species * logH, data = allom)
summary(fit)
```

```
##
## Call:
## lm(formula = logLA ~ species * logD + species * logH, data = allom)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -1.0554 -0.0881 -0.0075  0.1148  0.3412
##
## Coefficients:
##                  Estimate Std. Error t value Pr(>|t|)
## (Intercept)        -0.370      0.253   -1.46   0.1498
## speciesPIPO        -0.407      0.348   -1.17   0.2473
## speciesPSME         0.225      0.331    0.68   0.4997
## logD                1.398      0.496    2.82   0.0067 **
## logH                0.161      0.526    0.31   0.7606
## speciesPIPO:logD    1.503      0.650    2.31   0.0246 *
## speciesPSME:logD    1.723      0.718    2.40   0.0200 *
## speciesPIPO:logH   -1.524      0.674   -2.26   0.0278 *
## speciesPSME:logH   -2.089      0.790   -2.64   0.0107 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.216 on 54 degrees of freedom
## Multiple R-squared: 0.863,Adjusted R-squared: 0.842
## F-statistic: 42.4 on 8 and 54 DF,  p-value: <2e-16
```

Notice that the species effect is still fitted first; `R` will do this when using the "*" form of the formula. From the summary, the logD (log diameter) coefficient is significant. In the way this is set up this represents the slope for the base species, PIMO. The terms like `speciesPIPO:logD` are also significant which means that the slope of logD for PIPO, for example, is different from that for the base species, PIMO, etc.

An ANOVA whether adding these slope terms (in order) improves the model.

```
anova(fit)
```

```
## Analysis of Variance Table
##
## Response: logLA
##              Df Sum Sq Mean Sq F value  Pr(>F)
## species       2   0.54    0.27    5.84 0.00507 **
## logD          1  14.20   14.20  305.64 < 2e-16 ***
## logH          1   0.58    0.58   12.47 0.00085 ***
## species:logD  2   0.06    0.03    0.61 0.54494
## species:logH  2   0.38    0.19    4.04 0.02322 *
## Residuals    54   2.51    0.05
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Rather bizarrely, the ANOVA suggests that adding separate slopes on logD by species does not improve the model, but then adding separate slopes on logH by species *after this* does. These kinds of things can happen, each test uses an estimate of residual variance, based on the model fitted at the time. So the tests for the coefficients in the summary above use a residual variance from the full model. The (sequential)

ANOVA is using a different residual variance when it tests the `species:logD` term. We could investigate the remove one term at a time version.

```
drop1(fit, test = "F")

## Single term deletions
##
## Model:
## logLA ~ species * logD + species * logH
##               Df Sum of Sq  RSS  AIC F value Pr(>F)
## <none>                     2.51 -185
## species:logD  2     0.338 2.85 -181    3.64  0.033 *
## species:logH  2     0.375 2.88 -180    4.04  0.023 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Note that `R` will not remove main effects when interactions involving them are present. This ANOVA shows removing either slope interaction is detrimental to the model, just - the p-values are less than 5% but not by much.

Above all use a bit of common sense in this situation.

## 5.5 Logistic Regression

So far we have looked at modelling a continuous response on one or more factor variables (ANOVA) and one or more continuous variables (regression) and on combinations. And, in fact, we have been assuming that conditional on the independent variables, the response in Normally distributed; and we transformed leaf area, for example, using a log to make this assumption better.

In some cases, there is not an obvious way to do this, and in any case better methods have been developed. Examples, of such situations are when the response is a count or when it is binary, that is, has only two outcomes. In this section, we will consider binary outcomes using a method called logistic regression or binomial regression.

The dataset A.11 contains information on the survival status of passengers of the Titanic, including ages, gender and passenger class. Many ages are missing so we will remove here. (This is perhaps not the best approach!)

```
titanic <- read.table("titanic.txt", header = TRUE)
titanic <- na.omit(titanic)
titanic$Survived <- factor(ifelse(titanic$Survived == 1, "yes", "no"))
summary(titanic)

##                              Name     PClass          Age              Sex
##  Carlsson, Mr Frans Olof      :  2   1st:226   Min.   : 0.17   female:288
##  Connolly, Miss Kate          :  2   2nd:212   1st Qu.:21.00   male  :468
##  Kelly, Mr James              :  2   3rd:318   Median :28.00
##  Abbing, Mr Anthony           :  1             Mean   :30.40
##  Abbott, Master Eugene Joseph:  1             3rd Qu.:39.00
##  Abbott, Mr Rossmore Edward  :  1             Max.   :71.00
##  (Other)                     :747
##  Survived
##  no :443
##  yes:313
```

The idea here is to model whether the passenger survived on Age, Sex and PClass. Linear models are not appropriate here, since a binary variable cannot easily be approximated by a Normal distribution. The solution is logistic regression.

In logistic regression we model the probability of the "1" response (in this case the probability of survival). Since probabilities are between 0 and 1, we use a logistic transform of the linear predictor, where the linear predictor is of the form we would use in linear models above. If $\eta$ is the linear predictor and $Y$ is the binary response, the logistic model takes the form;

$$P(Y = 1) = \frac{1}{1 + e^{-\eta}} \tag{5.2}$$

These models are fit in `R` using the function `glm`. It has a similar interface to `lm` with a couple of additional features. To fit a logistic regression to the (modified) titanic data we use;

```
fit <- glm(Survived ~ Age + Sex + PClass, data = titanic, family = binomial)
summary(fit)

##
## Call:
## glm(formula = Survived ~ Age + Sex + PClass, family = binomial,
##     data = titanic)
##
## Deviance Residuals:
##    Min      1Q  Median      3Q     Max
## -2.723  -0.707  -0.392   0.649   2.529
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  3.75966    0.39757    9.46  < 2e-16 ***
## Age         -0.03918    0.00762   -5.14  2.7e-07 ***
## Sexmale     -2.63136    0.20151  -13.06  < 2e-16 ***
## PClass2nd   -1.29196    0.26008   -4.97  6.8e-07 ***
## PClass3rd   -2.52142    0.27666   -9.11  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 1025.57  on 755  degrees of freedom
## Residual deviance:  695.14  on 751  degrees of freedom
## AIC: 705.1
##
## Number of Fisher Scoring iterations: 5
```

The text `family=binomial` specifies logistic regression (aka binomial regression) here. The summary shows that all terms are significant. Interpreting the coefficients has to be done with care. For binary factors (such as Sex) they can be interpreted as a log-odds ratio, but this is beyond the scope of this text. At the very least the signs of the coefficients tell us about how the factor/variable affects the probability of survival. All are negative which means an increase in age decreases the chance of survival, being male decreases survival, being in 2nd or 3rd class decrease survival with 3rd being worse than 2nd.

### 5.5.1 Tabulated data

Sometimes we do not have the data for individuals, but counts are aggregated upto the level of the various factors. For example, suppose instead of age we had only the adult status of passengers.

```r
titanic$AgeGrp <- factor(ifelse(titanic$Age > 18, "Adult", "Child"))
```

We can then count the numbers of survivors and non-survivors,

```r
titanic2 <- aggregate(cbind(Survived == "yes", Survived == "no") ~ AgeGrp +
    Sex + PClass, data = titanic, sum)
print(titanic2)
```

```
##    AgeGrp    Sex PClass Survived == "yes" Survived == "no"
## 1   Adult female    1st                87                4
## 2   Child female    1st                 9                1
## 3   Adult   male    1st                37               81
## 4   Child   male    1st                 6                1
## 5   Adult female    2nd                58                9
## 6   Child female    2nd                17                1
## 7   Adult   male    2nd                10               99
## 8   Child   male    2nd                11                7
## 9   Adult female    3rd                27               36
## 10  Child female    3rd                19               20
## 11  Adult   male    3rd                25              157
## 12  Child   male    3rd                 7               27
```

```r
### Tidy up the names
names(titanic2)[4:5] <- c("survivors", "nonsurvivors")
```

The binomial glm has a special version that will allow us to model data in this form.

```r
fit <- glm(cbind(survivors, nonsurvivors) ~ AgeGrp + Sex + PClass, data = titanic2,
    family = binomial)
summary(fit)
```

```
##
## Call:
## glm(formula = cbind(survivors, nonsurvivors) ~ AgeGrp + Sex +
##     PClass, family = binomial, data = titanic2)
##
## Deviance Residuals:
##    Min      1Q  Median      3Q     Max
## -3.134  -1.294   0.826   2.039   2.997
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)    2.068      0.220    9.42  < 2e-16 ***
## AgeGrpChild    0.848      0.252    3.37  0.00077 ***
## Sexmale       -2.566      0.198  -12.96  < 2e-16 ***
## PClass2nd     -0.877      0.235   -3.73  0.00019 ***
## PClass3rd     -2.043      0.243   -8.42  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 361.947  on 11  degrees of freedom
## Residual deviance:  48.528  on  7  degrees of freedom
## AIC: 101
##
## Number of Fisher Scoring iterations: 5
```

The conclusions are much the same as before here.

## 5.6 Generalized Linear Models

In the previous section we used the `glm` function to fit a logistic (binomial) regression. The `glm` function name stands for *Generalized Linear Model*. A GLM model generalizes the usual linear model in two ways; (i) the response need not be related to the independent factors/variables linearly, and; (ii) the response can have distributions other than Normal.

**Note:** The is also a term *General Linear Model* which is not the same, although some statistics packages use GLM to mean general linear model, and something else for generalized linear model.

Again, an in depth discussion of GLMs is beyond the scope of this text. However, the basic idea is this. Suppose you have response variable $y$ and one or more independent variables that can be formed into a linear predictor in the same way as in linear models, which we call $\eta$. Then $y$ is modeled as some distribution with mean $\mu$, which itself is related to the independent variables, through the linear predictor and a *link*-function, $g$;

$$g(\mu) = \eta \tag{5.3}$$

In practice, the distribution for $y$ and the link-function are chosen depending on the problem. Logistic regression is an example of a GLM, with distribution binomial and link-function that below,

$$\log\left(\frac{\mu}{1-\mu}\right) = \eta$$

Another common GLM is where the distribution is Poisson and the link-function is log. This is also called Poisson regression, and it is often used where the response is count of items. Returning to the Age and Memory dataset, the response is a count of words recalled; perhaps Poisson regression is appropriate. Remember that when we fit this as a linear model, we found that the interaction was highly significant.

```
fit <- glm(Words ~ Age * Process, data = memory, family = poisson)
summary(fit)

##
## Call:
## glm(formula = Words ~ Age * Process, family = poisson, data = memory)
##
## Deviance Residuals:
##    Min      1Q  Median      3Q     Max
## -2.290  -0.492  -0.199   0.562   2.377
##
## Coefficients:
##                             Estimate Std. Error z value Pr(>|z|)
## (Intercept)                   2.3979     0.0953   25.15   <2e-16 ***
## AgeYounger                    0.2967     0.1259    2.36   0.0184 *
## ProcessCounting              -0.4520     0.1529   -2.96   0.0031 **
## ProcessImagery                0.1974     0.1287    1.53   0.1250
## ProcessIntentional            0.0870     0.1320    0.66   0.5098
## ProcessRhyming               -0.4664     0.1536   -3.04   0.0024 **
## AgeYounger:ProcessCounting   -0.3708     0.2133   -1.74   0.0822 .
## AgeYounger:ProcessImagery    -0.0241     0.1703   -0.14   0.8875
## AgeYounger:ProcessIntentional   0.1785     0.1714    1.04   0.2976
## AgeYounger:ProcessRhyming      -0.2001     0.2086   -0.96   0.3373
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for poisson family taken to be 1)
##
##     Null deviance: 227.503  on 99  degrees of freedom
## Residual deviance:  60.994  on 90  degrees of freedom
## AIC: 501.3
##
## Number of Fisher Scoring iterations: 4
```

```
anova(fit, test = "LRT")
```

```
## Analysis of Deviance Table
##
## Model: poisson, link: log
##
## Response: Words
##
## Terms added sequentially (first to last)
##
##
##             Df Deviance Resid. Df Resid. Dev Pr(>Chi)
## NULL                           99      227.5
## Age          1     20.8        98      206.7  5.2e-06 ***
## Process      4    137.5        94       69.3  < 2e-16 ***
## Age:Process  4      8.3        90       61.0    0.082 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Using the GLM (Poisson regression) the interaction is not significant. However, the default link for Poisson regression a log. This means that the mean $\mu$ in this case depends on the factors Age and Process in a multiplicative fashion, whereas in the linear model (ANOVA) it was additive without the interaction.

Note that in the `anova` function here we use something called a *Likelihood Ratio Test* (LRT). The usual F-tests may not be appropriate when the distribution is not Normal.

There are diagnostic plots for GLMs just like linear models, but these are beyond the scope here.

## 5.7 Random Effects?

# Chapter 6

# Summarizing, tabulating and merging data

## 6.1 Summarizing dataframes

There are a few useful functions to print general summaries of a dataframe, to see which variables are included, what types of data they contain, and so on.

The most basic function is `summary`, which works on many types of objects.

Let's look at the output for the `allom` dataset.

```
summary(allom)

##  species      diameter         height         leafarea
##  PIMO:19   Min.   : 4.83   Min.   : 3.57   Min.   :  2.6
##  PIPO:22   1st Qu.:21.59   1st Qu.:21.26   1st Qu.: 28.6
##  PSME:22   Median :34.80   Median :28.40   Median : 86.4
##            Mean   :35.56   Mean   :26.01   Mean   :113.4
##            3rd Qu.:51.44   3rd Qu.:33.93   3rd Qu.:157.5
##            Max.   :73.66   Max.   :44.99   Max.   :417.2
##    branchmass          logH             logD            logLA
##  Min.   :   1.8   Min.   :0.553   Min.   :0.684   Min.   :0.421
##  1st Qu.:  16.9   1st Qu.:1.328   1st Qu.:1.334   1st Qu.:1.453
##  Median :  72.0   Median :1.453   Median :1.542   Median :1.936
##  Mean   : 145.0   Mean   :1.344   Mean   :1.464   Mean   :1.805
##  3rd Qu.: 162.7   3rd Qu.:1.530   3rd Qu.:1.711   3rd Qu.:2.197
##  Max.   :1182.4   Max.   :1.653   Max.   :1.867   Max.   :2.620
```

For each factor variable, the levels are printed (the `species` variable, levels `PIMO`, `PIPO` and `PSME`. For all numeric variables, the minimum, first quantile, median, mean, third quantile, and the max. values are shown.

To simply see what types of variables your dataframe contains (or, for objects other than dataframes, to summarize sort of object you have), use the `str` function (short for 'structure').

```
str(allom)

## 'data.frame': 63 obs. of  8 variables:
##  $ species   : Factor w/ 3 levels "PIMO","PIPO",..: 3 3 3 3 3 3 3 3 3 3 ...
##  $ diameter  : num  54.6 34.8 24.9 28.7 34.8 ...
##  $ height    : num  27 27.4 21.2 25 30 ...
##  $ leafarea  : num  338.49 122.16 3.96 86.35 63.35 ...
##  $ branchmass: num  410.25 83.65 3.51 73.13 62.39 ...
##  $ logH      : num  1.43 1.44 1.33 1.4 1.48 ...
##  $ logD      : num  1.74 1.54 1.4 1.46 1.54 ...
##  $ logLA     : num  2.53 2.087 0.598 1.936 1.802 ...
```

Finally, two very useful functions from the `Hmisc` package. This package is installed in `R` by default, so you don't have to install it, but you do have to load it (once per session) (see Section 1.1.3).

```
library(Hmisc)
```

The `describe` function is much like `summary` (not shown; try this yourself).

The `contents` function is similar to `str`, but does a very nice job of summarizing the `factor` variables in your dataframe, and prints the number of missing variables, number of rows, and so on.

```
# read data
pupae <- read.csv("pupae.csv")

# Make sure CO2_treatment is a factor (it will be read as a number)
pupae$CO2_treatment <- as.factor(pupae$CO2_treatment)

# Show contents:
contents(pupae)

##
## Data frame:pupae 84 observations and 5 variables    Maximum # NAs:6
##
##                Levels Storage NAs
## T_treatment         2 integer   0
## CO2_treatment       2 integer   0
## Gender                integer   6
## PupalWeight            double   0
## Frass                  double   1
##
## +-------------+----------------+
## |Variable     |Levels          |
## +-------------+----------------+
## |T_treatment  |ambient,elevated|
## +-------------+----------------+
## |CO2_treatment|280,400         |
## +-------------+----------------+
```

Here, `storage` refers to the internal storage type of the variable: note that the factor variables are stored as 'integer' (this is correct, see section 2.1), and other numbers as 'double' (this refers to the precision of the number).

## 6.2 Making tables of averages

### 6.2.1 `tapply()`

Often, you want to summarize a variable by the levels of another variable. For example, in the `rain` data (see Section A.3), the `Rain` variable gives daily values, but we might want to calculate annual sums:

```
# Read data
rain <- read.csv("Rain.csv")

# Annual rain totals.
with(rain, tapply(Rain, Year, FUN = sum))

##   1996   1997   1998   1999   2000   2001   2002   2003   2004   2005
##  717.2  640.4  905.4 1021.3  693.5  791.5  645.9  691.8  709.5  678.2
```

The `tapply` function applies a function to a variable (`Rain`), that is split into chunks depending on another variable (`Year`).

We can also use the `tapply` function on more than one variable at a time. Consider these examples on the `pupae` data.

```
# Read data
pupae <- read.csv("pupae.csv")

# Average pupal weight by CO2 and T treatment:
with(pupae, tapply(PupalWeight, list(CO2_treatment, T_treatment), FUN = mean))

##     ambient elevated
## 280   0.290   0.3049
## 400   0.342   0.2990


# Further split the averages, by gender of the pupae.
with(pupae, tapply(PupalWeight, list(CO2_treatment, T_treatment, Gender), FUN = mean))

## , , 0
##
##     ambient elevated
## 280  0.2516   0.2700
## 400  0.3040   0.2687
##
## , , 1
##
##     ambient elevated
## 280  0.3406   0.3386
## 400  0.3568   0.3693
```

As the examples show, the `tapply` function produces summary tables by one or more factors. The resulting object is either a vector (when using one factor, or a matrix (as the examples using the pupae data).

The limitation of `tapply` is that you can only summarize one variable at a time.

### 6.2.2 `aggregate()`

Similar results can be obtained using the `aggregate` function, with the difference that this function produces a `dataframe`, which is often more desired. Consider these two examples,

```
# Average pupal weight by CO2 and T treatment:
with(pupae, aggregate(PupalWeight, list(CO2_treatment, T_treatment), FUN = mean))

##   Group.1  Group.2      x
## 1     280  ambient 0.2900
## 2     400  ambient 0.3420
## 3     280 elevated 0.3049
## 4     400 elevated 0.2990

# Further split the averages, by gender of the pupae.
with(pupae, aggregate(PupalWeight, list(CO2_treatment, T_treatment, Gender),
    FUN = mean))

##   Group.1  Group.2 Group.3      x
## 1     280  ambient       0 0.2516
## 2     400  ambient       0 0.3040
## 3     280 elevated       0 0.2700
## 4     400 elevated       0 0.2687
## 5     280  ambient       1 0.3406
## 6     400  ambient       1 0.3568
## 7     280 elevated       1 0.3386
## 8     400 elevated       1 0.3693
```

To finish things off, let's tidy up the result of the last aggregate result, by adding column names.

```
pupweight_agg <- with(pupae, aggregate(PupalWeight, list(CO2_treatment, T_treatment,
    Gender), FUN = mean))
names(pupweight_agg) <- c("CO2_treatment", "T_treatment", "Gender", "PupalWeightMean")

# Display the dataframe
pupweight_agg

##   CO2_treatment T_treatment Gender PupalWeightMean
## 1           280     ambient      0          0.2516
## 2           400     ambient      0          0.3040
## 3           280    elevated      0          0.2700
## 4           400    elevated      0          0.2687
## 5           280     ambient      1          0.3406
## 6           400     ambient      1          0.3568
## 7           280    elevated      1          0.3386
## 8           400    elevated      1          0.3693
```

### Advanced aggregate example

Let's look at a more advanced example using weather data collected at the Hawkesbury Forest Experiment in 2008 (see section A.9). The data given are in half-hourly timesteps, it is a reasonable request to provde data as daily averages (as for temperature), daily sums (as for precipitation and radiation), or daily minima/maxima (again, temperature).

The following code produces a daily weather dataset, and Figure 6.1.

```
# Read data
hfemet <- read.csv("HFEmet2008.csv")

# This is a sizeable dataset:
nrow(hfemet)

## [1] 17568


# so let's only look at the first few rows:
head(hfemet)
```

```
##         DateTime  Tair AirPress   RH     VPD PAR Rain wind winddirection
## 1 1/1/2008 0:00 16.54    101.8 93.3 0.12656   0    0    0         152.6
## 2 1/1/2008 0:30 16.45    101.8 93.9 0.11457   0    0    0         166.7
## 3 1/1/2008 1:00 16.44    101.7 94.2 0.10887   0    0    0         180.2
## 4 1/1/2008 1:30 16.41    101.7 94.5 0.10304   0    0    0         180.2
## 5 1/1/2008 2:00 16.38    101.7 94.7 0.09910   0    0    0         180.2
## 6 1/1/2008 2:30 16.23    101.7 94.7 0.09816   0    0    0         180.2


# Read DateTime as a POSIXct class
hfemet$DateTime <- as.POSIXct(as.character(hfemet$DateTime), format = "%m/%d/%Y %H:%M",
    tz = "GMT")
# Do NOT forget the tz='GMT' bit.

# Add a new variable 'Date', a daily date variable
hfemet$Date <- as.Date(hfemet$DateTime)

# First aggregate some of the variables into daily means:
hfemetAgg <- aggregate(hfemet[, c("PAR", "VPD", "Tair")], by = list(hfemet$Date),
    FUN = mean)
# (look at head(hfemetMeans) to check this went OK!)

# Rename columns to something more meaningful:
names(hfemetAgg) <- c("Date", "PARmean", "VPDmean", "Tairmean")

# Now get daily total PAR and Rainfall, add those onto the above dataset:
hfetmp <- aggregate(hfemet[, c("PAR", "Rain")], by = list(hfemet$Date), FUN = sum)

# PAR is now in funny units, convert to mol m-2 day-1:
hfetmp$PAR <- hfetmp$PAR * 10^-6 * 30 * 60

# Rename variables:
names(hfetmp) <- c("Date", "PARtot", "Raintot")

# We now have two dataframes, let's stick them together using cbind() Note
# that we can delete the first column of hfetmp (it has Date, we already
# have that)
hfemetAgg <- cbind(hfemetAgg, hfetmp[, -1])

# To finish things off, let's make a plot of daily total rainfall:
# (type='h' makes a sort of narrow bar plot)
with(hfemetAgg, plot(Date, Raintot, type = "h", ylab = expression(Rain ~ (mm ~
    day^-1))))
```

## 6.3 ave()

We saw with tapply how we can make simple tables of averages (or totals, or whatever) of some variable by the levels of one or more factor variables. The result of tapply is typically a vector with the length equal to the number of levels of the factor you summarized by (see examples in section 6.2.1).

what if you want the result of this sort of summary to be the length of the original vector? Let's look at an example where this might come in handy.

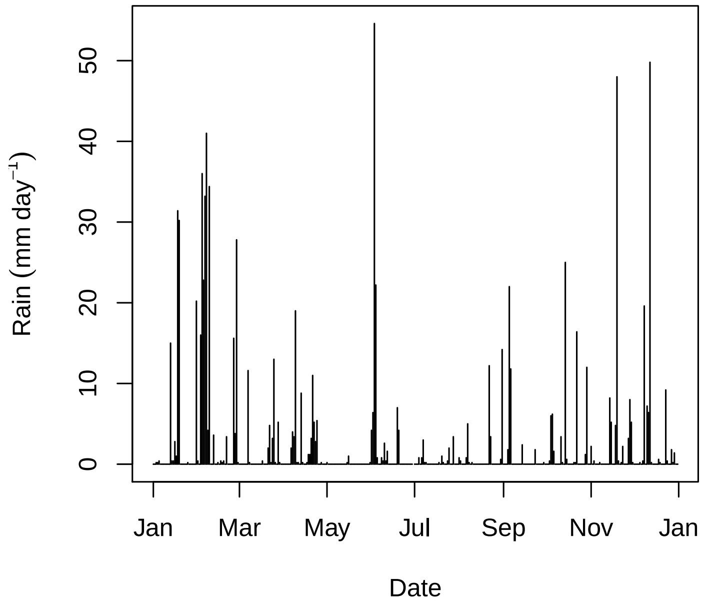


Figure 6.1: Daily rainfall at the HFE in 2008

Consider the `allometry` dataset, which includes tree height for three species. Suppose you want to add a new variable 'MaxHeight', that is the maximum tree height observed per species. We can use `ave` to achieve this:

```
# Read data
allom <- read.csv("Allometry.csv")

# Maximum tree height by species:
allom$MaxHeight <- ave(allom$height, allom$species, FUN = max)

# Look at first few rows (or just type allom to see whole dataset)
head(allom, 10)
```

```
##    species diameter height leafarea branchmass MaxHeight
## 1     PSME    54.61  27.04  338.486    410.246      33.3
## 2     PSME    34.80  27.42  122.158     83.650      33.3
## 3     PSME    24.89  21.23    3.958      3.513      33.3
## 4     PSME    28.70  24.96   86.351     73.130      33.3
## 5     PSME    34.80  29.99   63.351     62.390      33.3
## 6     PSME    37.85  28.07   61.373     53.866      33.3
## 7     PSME    22.61  23.43   32.078     22.225      33.3
## 8     PSME    39.37  32.45  147.271    119.389      33.3
## 9     PSME    39.88  30.10  141.787     95.585      33.3
## 10    PSME    26.16  25.85   45.020     45.331      33.3
```

Note that you can use any function in place of `max`, as long as that function can take a vector as an argument.

## 6.4 Merging datasets

In many problems, you do not have one simple dataset that contain all the measurements you are interested in, like most of the example datasets in this tutorial. Suppose you have two datasets that you would like to combine, or `merge`. This is straightforward in `R`, but there are some pitfalls.

Let's start with a common tricky problem when you need to combine two datasets that have different number of rows.For this type of problem, we can use the `merge` function.

```
# Two dataframes
data1 <- data.frame(unit = c("x", "x", "x", "y", "z", "z"), Time = c(1, 2, 3,
    1, 1, 2))
data2 <- data.frame(unit = c("y", "z", "x"), height = c(3.4, 5.6, 1.2))

# Look at the dataframes
data1

##   unit Time
## 1    x    1
## 2    x    2
## 3    x    3
## 4    y    1
## 5    z    1
## 6    z    2

data2

##   unit height
## 1    y    3.4
## 2    z    5.6
## 3    x    1.2


# Merge dataframes:
combdata <- merge(data1, data2, by = "unit")

# Combined data
combdata

##   unit Time height
## 1    x    1    1.2
## 2    x    2    1.2
## 3    x    3    1.2
## 4    y    1    3.4
## 5    z    1    5.6
## 6    z    2    5.6
```

Sometimes, the variable you are merging with has a different name in either dataframe. In that case, you can either rename the variable before merging, or use the following option:

```
merge(data1, data2, by.x="unit", by.y="item")
```

Where `data1` has a variable called 'unit', and `data2` has a variable called 'item'.

Other times you need to merge two dataframes where you have multiple key variables. Consider this example where two dataframes have measurements on the same units at some of the the same times, but measured different variables:

```
# Two dataframes
data1 <- data.frame(unit = c("x", "x", "x", "y", "y", "y", "z", "z", "z"), Time = c(1,
    2, 3, 1, 2, 3, 1, 2, 3), Weight = c(3.1, 5.2, 6.9, 2.2, 5.1, 7.5, 3.5, 6.1,
    8))
data2 <- data.frame(unit = c("x", "x", "y", "y", "z", "z"), Time = c(1, 2, 2,
    3, 1, 3), Height = c(12.1, 24.4, 18, 30.8, 10.4, 32.9))

# Look at the dataframes
data1

##   unit Time Weight
## 1    x    1    3.1
## 2    x    2    5.2
## 3    x    3    6.9
## 4    y    1    2.2
## 5    y    2    5.1
## 6    y    3    7.5
## 7    z    1    3.5
## 8    z    2    6.1
## 9    z    3    8.0

data2

##   unit Time Height
## 1    x    1   12.1
## 2    x    2   24.4
## 3    y    2   18.0
## 4    y    3   30.8
## 5    z    1   10.4
## 6    z    3   32.9


# Merge dataframes:
combdata <- merge(data1, data2, by = c("unit", "Time"))

# By default, only those times appear in the dataset that have
# measurements for both Weight (data1) and Height (data2)
combdata

##   unit Time Weight Height
## 1    x    1    3.1   12.1
## 2    x    2    5.2   24.4
## 3    y    2    5.1   18.0
## 4    y    3    7.5   30.8
## 5    z    1    3.5   10.4
## 6    z    3    8.0   32.9


# To include all data, use this command. This produces missing values for
# some times:
merge(data1, data2, by = c("unit", "Time"), all = TRUE)

##   unit Time Weight Height
## 1    x    1    3.1   12.1
## 2    x    2    5.2   24.4
## 3    x    3    6.9     NA
## 4    y    1    2.2     NA
## 5    y    2    5.1   18.0
## 6    y    3    7.5   30.8
## 7    z    1    3.5   10.4
## 8    z    2    6.1     NA
## 9    z    3    8.0   32.9
# Compare this result with 'combdata' above!
```

### Merging multiple datasets

Consider the cereal dataset (section A.6), which gives measurements of all sorts of contents of cereals. Suppose the measurements for 'protein', 'vitamins' and 'sugars' were all produced by different labs, and each lab sends you a separate dataset. To make things worse, some measurements for sugars and vitamins are missing, because samples were lost in those labs.

How to put things together?

```
# Read the three datasets given to you from the three different labs:
cereal1 <- read.csv("cereal1.csv")
cereal2 <- read.csv("cereal2.csv")
cereal3 <- read.csv("cereal3.csv")

# Look at the datasets:
cereal1
##                   Cereal.name protein
## 1              Frosted_Flakes       1
## 2                  Product_19       3
## 3               Count_Chocula       1
## 4                  Wheat_Chex       3
## 5                  Honey-comb       1
## 6  Shredded_Wheat_spoon_size       3
## 7         Mueslix_Crispy_Blend       3
## 8            Grape_Nuts_Flakes       3
## 9     Strawberry_Fruit_Wheats       2
## 10                   Cheerios       6
cereal2
##                   cerealbrand vitamins
## 1                  Product_19      100
## 2               Count_Chocula       25
## 3                  Wheat_Chex       25
## 4                  Honey-comb       25
## 5 Shredded_Wheat_spoon_size        0
## 6      Mueslix_Crispy_Blend       25
## 7         Grape_Nuts_Flakes       25
## 8                   Cheerios       25
cereal3
##                  cerealname sugars
## 1           Frosted_Flakes     11
## 2               Product_19      3
## 3    Mueslix_Crispy_Blend     13
## 4        Grape_Nuts_Flakes      5
## 5 Strawberry_Fruit_Wheats      5
## 6                 Cheerios      1
```

```
# Note that the number of rows is different between the datasets, and even
# the index name ('Cereal.name') differs between the datasets.

# To merge them all together, use merge() twice, like this.
cerealdata <- merge(cereal1, cereal2, by.x = "Cereal.name", by.y = "cerealbrand",
    all.x = TRUE)
# NOTE: all.x=TRUE specifies to keep all rows in cereal1 that do not exist
# in cereal2.

# Then merge again:
cerealdata <- merge(cerealdata, cereal3, by.x = "Cereal.name", by.y = "cerealname",
    all.x = TRUE)

# And double check the final result
cerealdata

##                  Cereal.name protein vitamins sugars
## 1                   Cheerios       6       25      1
## 2              Count_Chocula       1       25     NA
## 3             Frosted_Flakes       1       NA     11
## 4          Grape_Nuts_Flakes       3       25      5
## 5                 Honey-comb       1       25     NA
## 6        Mueslix_Crispy_Blend       3       25     13
## 7                 Product_19       3      100      3
## 8  Shredded_Wheat_spoon_size       3        0     NA
## 9    Strawberry_Fruit_Wheats       2       NA      5
## 10                Wheat_Chex       3       25     NA

# Note that missing values (NA) have been inserted where some data was not
# available.
```

### Merging multiple datasets

Consider the cereal dataset (section A.6), which gives measurements of all sorts of contents of cereals. Suppose the measurements for 'protein', 'vitamins' and 'sugars' were all produced by different labs, and each lab sends you a separate dataset. To make things worse, some measurements for sugars and vitamins are missing, because samples were lost in those labs.

How to put things together?

```
# Read the three datasets given to you from the three different labs:
cereal1 <- read.csv("cereal1.csv")
cereal2 <- read.csv("cereal2.csv")
cereal3 <- read.csv("cereal3.csv")

# Look at the datasets:
cereal1

##                  Cereal.name protein
## 1             Frosted_Flakes       1
## 2                 Product_19       3
## 3              Count_Chocula       1
## 4                 Wheat_Chex       3
## 5                 Honey-comb       1
## 6  Shredded_Wheat_spoon_size       3
## 7       Mueslix_Crispy_Blend       3
## 8          Grape_Nuts_Flakes       3
## 9    Strawberry_Fruit_Wheats       2
## 10                  Cheerios       6
```

```
cereal2
```

```
##                cerealbrand vitamins
## 1               Product_19      100
## 2            Count_Chocula       25
## 3               Wheat_Chex       25
## 4               Honey-comb       25
## 5 Shredded_Wheat_spoon_size        0
## 6      Mueslix_Crispy_Blend       25
## 7         Grape_Nuts_Flakes       25
## 8                 Cheerios       25
```

```
cereal3
```

```
##               cerealname sugars
## 1          Frosted_Flakes     11
## 2              Product_19      3
## 3    Mueslix_Crispy_Blend     13
## 4       Grape_Nuts_Flakes      5
## 5 Strawberry_Fruit_Wheats      5
## 6                Cheerios      1
```

```
# Note that the number of rows is different between the datasets, and even
# the index name ('Cereal.name') differs between the datasets.

# To merge them all together, use merge() twice, like this.
cerealdata <- merge(cereal1, cereal2, by.x = "Cereal.name", by.y = "cerealbrand",
    all.x = TRUE)
# NOTE: all.x=TRUE specifies to keep all rows in cereal1 that do not exist
# in cereal2.

# Then merge again:
cerealdata <- merge(cerealdata, cereal3, by.x = "Cereal.name", by.y = "cerealname",
    all.x = TRUE)

# And double check the final result
cerealdata
```

```
##                  Cereal.name protein vitamins sugars
## 1                   Cheerios       6       25      1
## 2              Count_Chocula       1       25     NA
## 3             Frosted_Flakes       1       NA     11
## 4          Grape_Nuts_Flakes       3       25      5
## 5                 Honey-comb       1       25     NA
## 6       Mueslix_Crispy_Blend       3       25     13
## 7                 Product_19       3      100      3
## 8  Shredded_Wheat_spoon_size       3        0     NA
## 9    Strawberry_Fruit_Wheats       2       NA      5
## 10                Wheat_Chex       3       25     NA
```

```
# Note that missing values (NA) have been inserted where some data was not
# available.
```

## 6.5 Exercises

### 6.5.1 Summarizing the cereal data

*1.* Read the cereal data, and produce quick summaries using `str`, `summary` and `contents` (the latter is in the `Hmisc` package). Interpret the results.

*2.* Find the average sodium, fiber and carbohydrates by Manufacturer.

*3.* Add a new variable 'SodiumClass', which is 'high' when sodium ¿ 150 and 'low' otherwise. Make sure the new variable is a factor. Find the average sugar content for 'low' and 'high' sodium.

*4.* Find the maximum sugar content by Manufacturer AND SodiumClass. Inspect the result and figure out why there are missing values.

*5.* If you used `tapply` for the previous question, repeat it with `aggregate`. Or vice versa. Compare the results.

### 6.5.2 More summarizing exercises

*1.* Using the 'Age and memory' dataset (see section A.10 on how to read this dataset!), find out the mean and maximum number of words recalled by 'Older' and 'Younger' age class.

*2.* Using the HFE weather dataset (section A.9), find the mean air temperature by month. To do this, first add the month variable as shown in section 2.5.2.

### 6.5.3 Weight loss

This exercise takes together many skills from chapters 1,2 and 3.

Consider a dataset of sequential measurements of a person's weight while on a diet (the 'weightloss' dataset, see Section A.4). Finish the following steps:

*1.* Read the dataset (`weightloss.csv`), and convert the 'Date' variable to a proper `Date` class. See Section 2.5.1 for converting the date, and note the example in Section 2.5.1!.

*2.* Add a new variable to the dataset, with his weight in kilograms (kg) (1 kg = 2.204 pounds).

*3.* Produce a line plot that shows his weight (in kg) versus time.

*4.* The problem with the plot you just produced is that all measurements are connected by a line, although we would like to have line breaks for the days where the weight was not measured. To do this, construct a dataframe based on the `weightloss` dataset that has daily values. Hints:

- Make an entirely new dataframe, with a Date variable, ranging from the first to last days in the weightloss dataset.
- Using `merge`, paste the Weight data onto this new dataframe. Use the new dataframe to make the plot.

*5.* Based on the new dataframe you just produced, graph the daily change in weight versus time. Hint: use the `diff` function.

# Chapter 7

# Functions, lists and loops

## 7.1 Writing simple functions

We have already used many built-in functions throughout this tutorial, but you can become very efficient at complex data tasks when you write your own simple functions. Writing your own functions can help with tasks that are carried out many times, which would result in a lot of code.

Let's write a function for the standard error of the mean, a function that is not built-in in `R`.

```r
SEmean <- function(x) {
    se <- sd(x)/sqrt(length(x))
    return(se)
}
```

Here, the `function` SEmean takes one 'argument' called `x` (i.e., input), which is a numeric vector. The standard error for the mean is calculated in the first line, and stored in an object called `se`, which is then returned as output. We can now use the function on a numeric vector like this:

```r
# A numeric vector
unifvec <- runif(10, 1, 2)

# The sample mean
mean(unifvec)

## [1] 1.489


# Standard error for the mean
SEmean(unifvec)

## [1] 0.08693
```

## 7.2 Working with lists

Sofar, we have worked a lot with vectors, with are basicaly strings of numbers or bits of text. In a vector, each element has to be of the same data type. Lists are a more general and powerful type of vector, where each element of the `list` can be anything at all. This way, lists are a very powerful type of object to store a lot of information that may be in different formats.

Lists are at somewhat daunting for the beginning R user, which is why most introductory texts and tutorials skip them altogether. However, with some practice, lists can be mastered from the start. Mastering a few basic skills with lists can really help increase your efficiency in dealing with more complex data analysis tasks.

Let's look at constructing a list that contains a numeric vector, a character vector, and a dataframe:

```
mylist <- list(a = 1:10, txt = c("hello", "world"), dfr = data.frame(x = c(2,
    3, 4), y = c(5, 6, 7)))
```

### 7.2.1 Breaking up dataframes

For more advanced analyses, it is often necessary to repeat a particular analysis many times, for example sections of a dataframe.

Using the `allom` data for example, we might want to split the dataframe into three dataframes: one per species. The resulting list will then have three components. There are a number of advantages to this approach over the more obvious choice of splitting the dataset into three files, and reading these separately. The most important advantage is when you have a large number of levels, which would result in many files.

Let's look at an example on how to construct a list of dataframes from the allom dataset, one per species:

```
# Read allom data, make sure 'species' is a factor:
allom <- read.csv("Allometry.csv")
is.factor(allom$species)

## [1] TRUE


# The levels of the factor variable 'species'
levels(allom$species)

## [1] "PIMO" "PIPO" "PSME"


# Now use 'split' to construct a list:
allomsp <- split(allom, allom$species)

# The length of the list should be 3, with the names equal to the original
# factor levels:
length(allomsp)

## [1] 3

names(allomsp)

## [1] "PIMO" "PIPO" "PSME"
```

Let's look at a more advanced example using the `hydro` data. The data contains water levels of a hydrodam in Tasmania, from 2005 to 2011 (see section A.2).

```
# Read hydro data, and convert Date to a proper date class.
hydro <- read.csv("hydro.csv")
hydro$Date <- as.Date(as.character(hydro$Date), format = "%d/%m/%Y")

# Extract 'year' from Date: (Note that 'year' is coded as years since
# 1900)
hydro$year <- 1900 + as.POSIXlt(hydro$Date)$year
```

```
# Look at the Date range:
range(hydro$Date)

## [1] "2005-08-08" "2011-08-08"


# Let's get rid of the first and last years (2005 and 2011) since they are
# incomplete
hydro <- subset(hydro, !year %in% c(2005, 2011))

# Now split the dataframe by year. This results in a list, where every
# element contains the data for one year:
hydrosp <- split(hydro, hydro$year)

# Properties of this list:
length(hydrosp)

## [1] 5

names(hydrosp)

## [1] "2006" "2007" "2008" "2009" "2010"
```

### 7.2.2 Applying functions to lists

There are two basic tools that we use to apply functions to each element of a list: `sapply` and `lapply`. We use `sapply` when the result of the function is a single value (typically, a single number). We use `lapply` for everything else.

First let's look at some simple examples:

```
# Let's make a simple list with only numeric vectors (of varying length)
numlis <- list(x = c(10, 9, 1, 13, 9), y = c(2.1, 0.1, -19), z = c(1000, 2000,
    3000, 4000))

# For the numeric list, let's get the mean for every element, and count
# the length of the three vectors:
sapply(numlis, mean)

##      x      y      z
##    8.4   -5.6 2500.0

sapply(numlis, length)

## x y z
## 5 3 4


# It is straightforward to define your own functions within sapply, for
# more flexibility: Here, 'x' refers to an element of the list, to which
# the function is applied.
sapply(numlis, function(x) any(duplicated(x)))

##     x     y     z
##  TRUE FALSE FALSE


# Recall that the 'strsplit' (string split) function usually returns a
# list of values. Consider the following example, where the data provider
# has included the units in the measurements of fish lengths.  How do we
# extract the number bits?
fishlength <- c("120 mm", "240 mm", "159 mm", "201 mm")

# Here is one solution, using strsplit
strsplit(fishlength, " ")

## [[1]]
## [1] "120" "mm"
##
## [[2]]
## [1] "240" "mm"
##
## [[3]]
## [1] "159" "mm"
##
## [[4]]
## [1] "201" "mm"


# We see that strsplit returns a list, let's use sapply to extract only
# the first element (the number)
splitlen <- strsplit(fishlength, " ")
sapply(splitlen, function(x) x[1])

## [1] "120" "240" "159" "201"


# Now all you need to do is use 'as.numeric' to convert these bits of text
# to numbers.
```

The main purpose of splitting dataframes into lists, as we have done above, is so that we can save time with analyses that have to be repeated many times. In the following examples, you must have already produced the objects `hydrosp` and `allomsp` (from examples in the previous section).Both those objects are *lists of dataframes*, that is, each element of the list is a dataframe in itself. Let's look at a few examples with `sapply` first.

```
# How many observations per species in the allom dataset?
sapply(allomsp, nrow)

## PIMO PIPO PSME
##   19   22   22

# Here, we applied the 'nrow' function to each separate dataframe.  (note
# that there are easier ways to find the number of observations per
# species!, this is just illustrating sapply.)

# Things get more interesting when you define your own functions on the
# fly:
sapply(allomsp, function(x) range(x$diameter))

##       PIMO  PIPO  PSME
## [1,]  6.48  4.83  5.33
## [2,] 73.66 70.61 69.85

# Here, we define a function that takes 'x' as an argument: sapply will
# apply this function to each element of the list, one at a time. In this
# case, we get a matrix with ranges of the diameter per species.
# How about the correlation of two variables, separate by species:
sapply(allomsp, function(x) cor(x$diameter, x$height))

##   PIMO   PIPO   PSME
## 0.9140 0.8595 0.8783


# For hydro, find the Date of the minimum water level, for each year (use
# hydrosp from above).
sapply(hydrosp, function(x) x$Date[which.min(x$storage)], simplify = FALSE)

## $`2006`
## [1] "2006-07-03"
##
## $`2007`
## [1] "2007-04-30"
##
## $`2008`
## [1] "2008-06-23"
##
## $`2009`
## [1] "2009-05-11"
##
## $`2010`
## [1] "2010-04-12"

# Here we have to set 'simplify=FALSE' to make sure the Dates are printed
# correctly.
```

Now things get even more interesting, let's use `lapply` to analyze our datasets:

```
# Get a summary of the allom dataset by species:
lapply(hydrosp, summary)

## $`2006`
##       Date               storage         year
##  Min.   :2006-01-02   Min.   :411   Min.   :2006
##  1st Qu.:2006-04-01   1st Qu.:450   1st Qu.:2006
##  Median :2006-06-29   Median :493   Median :2006
##  Mean   :2006-06-29   Mean   :514   Mean   :2006
##  3rd Qu.:2006-09-26   3rd Qu.:554   3rd Qu.:2006
##  Max.   :2006-12-25   Max.   :744   Max.   :2006
##
## $`2007`
##       Date               storage         year
##  Min.   :2007-01-01   Min.   :137   Min.   :2007
##  1st Qu.:2007-04-02   1st Qu.:223   1st Qu.:2007
##  Median :2007-07-02   Median :319   Median :2007
##  Mean   :2007-07-02   Mean   :363   Mean   :2007
##  3rd Qu.:2007-10-01   3rd Qu.:477   3rd Qu.:2007
##  Max.   :2007-12-31   Max.   :683   Max.   :2007
##
## $`2008`
##       Date               storage         year
##  Min.   :2008-01-07   Min.   :202   Min.   :2008
##  1st Qu.:2008-04-05   1st Qu.:268   1st Qu.:2008
##  Median :2008-07-03   Median :345   Median :2008
##  Mean   :2008-07-03   Mean   :390   Mean   :2008
##  3rd Qu.:2008-09-30   3rd Qu.:552   3rd Qu.:2008
##  Max.   :2008-12-29   Max.   :637   Max.   :2008
##
## $`2009`
##       Date                storage         year
##  Min.   :2009-01-05   Min.   :398   Min.   :2009
##  1st Qu.:2009-04-04   1st Qu.:467   1st Qu.:2009
##  Median :2009-07-02   Median :526   Median :2009
##  Mean   :2009-07-02   Mean   :568   Mean   :2009
##  3rd Qu.:2009-09-29   3rd Qu.:678   3rd Qu.:2009
##  Max.   :2009-12-28   Max.   :791   Max.   :2009
##
## $`2010`
##       Date                storage         year
##  Min.   :2010-01-04   Min.   :251   Min.   :2010
##  1st Qu.:2010-04-03   1st Qu.:316   1st Qu.:2010
##  Median :2010-07-01   Median :368   Median :2010
##  Mean   :2010-07-01   Mean   :428   Mean   :2010
##  3rd Qu.:2010-09-28   3rd Qu.:578   3rd Qu.:2010
##  Max.   :2010-12-27   Max.   :649   Max.   :2010


# More...
```

Let's do a simple linear regression of log(leafarea) on log(diameter) for the `allom` dataset, by species:

```
# Run the linear regression on each element of the list, store in a new
# object:
lmresults <- lapply(allomsp, function(x) lm(log10(leafarea) ~ log10(diameter),
    data = x))

# Now, lmresults is itself a list (where each element is an 'lm' object)
# We can extract the coefficients like this:
sapply(lmresults, coef)

##                    PIMO    PIPO    PSME
## (Intercept)     -0.357 -0.7368 -0.3136
## log10(diameter)  1.541  1.6428  1.4841

# This shows the intercept and slope by species.  Also look at
# lapply(lmresults, summary) yourself (not shown).
```

## 7.3 Loops

Loops can be useful when we need to repeat certain analyses many times, and it is difficult to achieve this with `lapply` or `sapply`. To understand how a `for` loop works, look at this example:

```
for (i in 1:5) {
    message(i)
}

## 1

## 2
## 3

## 4

## 5
```

Here, the bit of code between  is executed five times, and the object `i` has the values 1,2,3,4 and 5, in that order. In stead of just printing `i` as we have done above, we can also index a vector with this object:

```
# make a vector
myvec <- round(runif(5), 1)

for (i in 1:length(myvec)) {
    print(myvec[i])
}
## [1] 0.7
## [1] 0.5
## [1] 0.3
## [1] 0.9
## [1] 0.3
```

Note that this is only a toy example: the same result can be achieved by simply typing `myvec`.

Now let's look at a very useful application of a `for` loop: producing multiple plots in a `pdf`, using the `allomsp` object we created earlier.

This bit of code produces a `pdf` in your current working directory. Can't find it? Look at `getwd()` for the current working directory.

```
# Open a pdf to send the plots to:
pdf("Allom plot by species.pdf", onefile = TRUE)
for (i in 1:3) {
    with(allomsp[[i]], plot(diameter, leafarea, pch = 15, xlim = c(0, 80), ylim = c(0,
        450), main = levels(allom$species)[i]))
}
# Close the pdf (important.)
dev.off()
## pdf
##   2
```

Here, we create three plots (`i` goes from 1 to 3), every time using a different element of the list `allomsp`. First, `i` will have the value 1, so that we end up using the dataframe `allomsp[[1]]`, the first element of the list. And so on. Take a look at the resulting pdf to understand what is going on here.

## 7.4 Exercises

## 7.5 Writing functions

*1.* You learned in Chapter 2 that you can take subset of a string using the `substr` function. First, using that function to extract the first 2 characters of a bit of text. Then, write a function called `firstTwoChars` that extracts the first two characters of any bit of text.

*2.* The function `readline` can be used to ask for data to be typed in. First, figure out how to use `readline` by reading the corresponding help file. Then, construct a function called

*3. Advanced.* Recall the functions `head` and `tail`. Write a function called `middle` that shows a few rows around (approx.) the 'middle' of the dataset. *Hint:* use `nrow`, and `print`.

### 7.5.1 Multiple plots

### 7.5.2 Expert: Central limit theorem

The 'central limit theorem' (CLT) forms the backbone of inferential statistics. This theorem states (informally) that if you draw samples (of $n$ units) from a population, the mean of these samples follows a normal distribution. This is true regardless of the underlying distribution you sample from.

In this exercise, you will apply a simple simulation study to test the CLT, and to make histograms and quantile-quantile plots.

*1.* Draw 200 samples of size 10 from a uniform distribution. Use the `runif` function to sample from the uniform distribution, and the `replicate` function to repeat this many times.

*2.* Compute the sample mean for each of the 200 samples in *1.*. Use `apply` or `colMeans` to calculate column-wise means of a matrix (note: `replicate` will return a matrix, if used correctly).

*3.* Draw a histogram of the 200 sample means, using `hist`. Also draw a normal quantile-quantile plot, using `qqnorm`.

*4.* On the histogram, add a normal curve using the `dnorm` function. Note: to do this, plot the histogram with the argument `freq=FALSE`, so that the histogram draws the probability density, not the frequency.

*5.* Write a function that does all of the above, and call it `PlotCLT`.

# Chapter 8

# Multivariate Statistics

## 8.1 Using multivariate statistics: when and why?

In many studies, several types of data are collected on the same unit. Analysing these in combination may reveal patterns that are indicative of structure in these data, which may be more informative than looking at each individual variable. You may want to do some exploratory data analysis with these data to see whether certain individuals can be classified into groups based on their relative similarities. This would be useful, for example, for identifying soil types to be used as blocking factors in an experiment or groups of individuals that expressed similar traits in response to an experimental manipulation. You may also want to determine whether two or more variables provide essentially the same information, which will allow you to more efficiently focus your efforts on one of these. Or you may have specific hypotheses that you are interested in testing and are worried that, by performing these tests on multiple response variables, you may be inflating error rates associated with rejecting a correct null hypothesis (Type I error).

These are examples of when you might want to use multivariate statistical analysis. You should think carefully about using multivariate statistics when the number of units being studied is on the same order as the number of variables being measured. A rule of thumb is that there should be five independent units for every variable being measured. You can proceed with the analysis if you have less than this; in fact, you can even proceed with the analysis when you have fewer observations than you have variables. However, you should think carefully about replication when deciding whether it would be worthwhile to measure additional response variables, especially when testing hypotheses.

There are several resources for performing multivariate statistical analysis in R and a summary of these can be found at the Environmetrics CRAN Task View (`http://cran.r-project.org/web/views/Environmetrics.html`) under the 'Ordination', 'Dissimilarity coefficients', and 'Cluster analysis' headings. A few tools are in the 'stats' package, which comes with your R distribution, but environmental scientists tend to mainly use the **'vegan'** and/or **'ade4'** packages. There is a lot of overlap between these packages in terms of the types of analyses that can be done, but the structure of the output is very different. They both contain a function named cca(), so be careful if you have both packages loaded at the same time since the most recently loaded version of cca() will be called. Many other packages used by ecologists and evolutionary biologists rely on these packages.

The Spatial CRAN Task View (`http://cran.r-project.org/web/views/Spatial.html`) will also be useful for more complex analysis of univariate and multivariate data with explicit spatial structure.

## 8.2 Exploring your data: unconstrained ordination

Unconstrained ordination approaches are very useful for simplifying multivariate data for visualisation of patterns. These should always be the first step in the analysis of multivariate data, even when you are interested in testing specific hypotheses regarding the potential indicators and drivers of data structure. *The R Book* by Michael Crawley includes a chapter on these methods that is accessible to environmental scientists.

### 8.2.1 Principal Components Analysis (PCA)

Several variants are commonly used in the environmental sciences but are ultimately based on PCA. PCA is ultimately transformation of continuous multivariate data into new variables that maximise the amount of variance explained in the data, with each subsequent variable orthogonal (think perpendicular in two dimensions) to the previous and explaining decreasing amounts of variation. The new variables are linear combinations of the original variables and the contribution of each of the original variables to the new variables is indicated by their loadings.

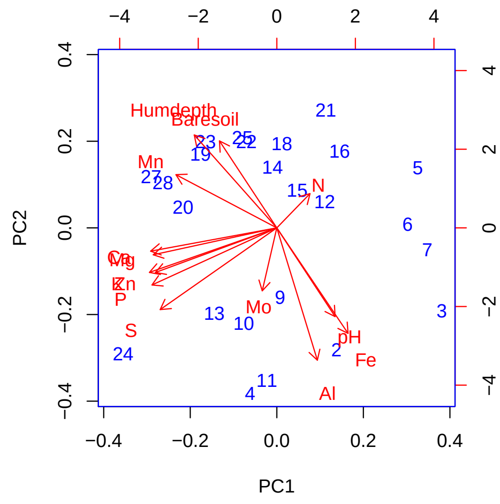
Performing these is simple using `prcomp()`. We'll use 'varechem' data from the 'vegan' package, which contains observations of edaphic variables associated with 24 sites grazed by reindeer.

```
library(vegan)
data(varechem)  # read data into your workspace
str(varechem)

## 'data.frame': 24 obs. of  14 variables:
##  $ N       : num  19.8 13.4 20.2 20.6 23.8 22.8 26.6 24.2 29.8 28.1 ...
##  $ P       : num  42.1 39.1 67.7 60.8 54.5 40.9 36.7 31 73.5 40.5 ...
##  $ K       : num  140 167 207 234 181 ...
##  $ Ca      : num  519 357 973 834 777 ...
##  $ Mg      : num  90 70.7 209.1 127.2 125.8 ...
##  $ S       : num  32.3 35.2 58.1 40.7 39.5 40.8 33.8 27.1 42.5 60.2 ...
##  $ Al      : num  39 88.1 138 15.4 24.2 ...
##  $ Fe      : num  40.9 39 35.4 4.4 3 ...
##  $ Mn      : num  58.1 52.4 32.1 132 50.1 ...
##  $ Zn      : num  4.5 5.4 16.8 10.7 6.6 9.1 7.4 5.2 9.3 9.1 ...
##  $ Mo      : num  0.3 0.3 0.8 0.2 0.3 0.4 0.3 0.3 0.3 0.5 ...
##  $ Baresoil: num  43.9 23.6 21.2 18.7 46 40.5 23 29.8 17.6 29.9 ...
##  $ Humdepth: num  2.2 2.2 2 2.9 3 3.8 2.8 2 3 2.2 ...
##  $ pH      : num  2.7 2.8 3 2.8 2.7 2.7 2.8 2.8 2.8 2.8 ...
```

The values associated with the different gradients are very different, and their variances have to be standardised otherwise variables with large absolute variances will have greater weight. This is done with the 'scale' argument (frustratingly, scale=F is the default).

```
var(varechem$pH)  # variance in soil pH

## [1] 0.0458

var(varechem$Fe)  # variance in iron concentration

## [1] 3654

chem.pca <- prcomp(varechem, scale = T)
summary(chem.pca)

## Importance of components:
##                           PC1   PC2   PC3    PC4    PC5    PC6    PC7
## Standard deviation     2.278 1.787 1.298 1.0339 0.9033 0.8401 0.6606
## Proportion of Variance 0.371 0.228 0.120 0.0764 0.0583 0.0504 0.0312
## Cumulative Proportion  0.371 0.599 0.719 0.7956 0.8539 0.9043 0.9355
##                           PC8    PC9   PC10    PC11    PC12   PC13    PC14
## Standard deviation     0.6073 0.4132 0.3867 0.29199 0.26430 0.1872 0.15367
## Proportion of Variance 0.0263 0.0122 0.0107 0.00609 0.00499 0.0025 0.00169
## Cumulative Proportion  0.9618 0.9740 0.9847 0.99082 0.99581 0.9983 1.00000
```

```
biplot(chem.pca)
```



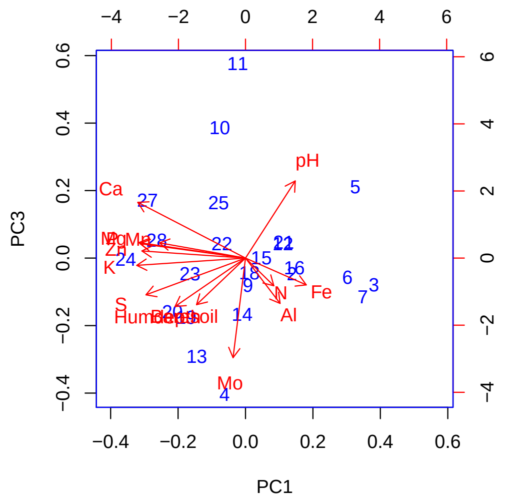
The plot shows how the samples (indicated by number) are separated based on the first two principal components, the new variables resulting from the transformations of the original variables based on their ability to account for variation in the multivariate data. According to the summary table, these components explain 37 and 23 % of the variance in these data. The samples are labelled according to the row names of the input dataframe or matrix. The arrows, or vectors, indicate the loadings associated with the original variables. From this, we can see that the vectors for calcium and magnesium overlap entirely and, therefore, are positively correlated along both of the first two principal components. Because they are almost horizontal, they loadings associated with the first axis are much greater than that for the second axis. Sulfur and nitrogen are negatively correlated along both axes: their vectors are running in opposite directions. The direction of the arrow along each axis indicates samples in which the values for that variable are high; from this we see that sample 24 is relatively high in calcium, magnesium, zinc, potassium, phosphorus, and sulfur while samples 3, 5, 6, and 7 are low in these elements.

```
varechem[c("5", "6", "7", "3", "24"), ]
```

```
##        N    P     K    Ca    Mg    S    Al    Fe   Mn   Zn  Mo Baresoil
## 5   33.1 22.7  43.6 240.3  25.7 14.9  39.0   8.4 26.8  8.4 0.2      8.1
## 6   19.1 26.4  61.1 259.1  37.0 21.4 155.1  81.4 20.6  4.0 0.6      5.8
## 7   30.5 24.6  78.7 188.5  55.5 25.3 294.9 123.8 10.1  3.0 0.4     18.6
## 3   31.1 32.3  73.7 219.0  52.5 25.5 304.6 204.4 14.2  2.6 0.5      3.6
## 24  20.2 67.7 207.1 973.3 209.1 58.1 138.0  35.4 32.1 16.8 0.8     21.2
##    Humdepth  pH
## 5       1.0 3.1
## 6       1.9 3.0
## 7       1.7 3.1
## 3       1.8 3.3
## 24      2.0 3.0
```

The third component explains 12 % of variation, and we may be interested to see how samples and variables are distributed along this axis. To do this, we use the 'choice' argument.

```
biplot(chem.pca, choice = c(1, 3))
```



The first principal component is still plotted on the x-axis, but now the third component is plotted on the y-axis. We can see that calcium and magnesium are separated along the third axis (variation in calcium is explained by this axis, but not magnesium, which has loading close to zero). We also see that nitrogen and sulfur are positively correlated along the third axis (both are positioned on the negative side of this axis). The sample and variable scores for the first component are the same as before.

Most of the variation is accounted for in the first two principal components and ¿ 90 % is accounted for by the first six components. Any patterns observed associated with subsequent components will not contain much information.

To get a better idea of what happens in the analysis, let's look more closely at the resulting object.

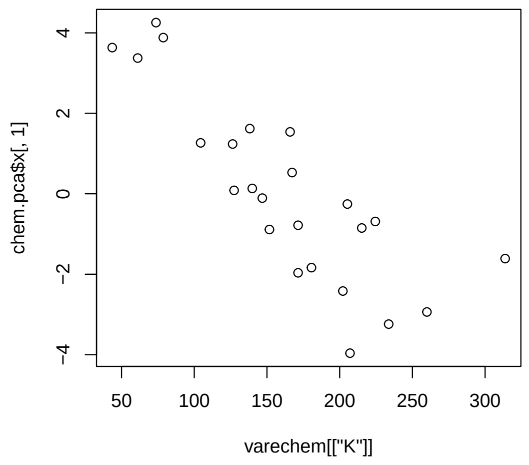
```
str(chem.pca)

## List of 5
##  $ sdev    : num [1:14] 2.278 1.787 1.298 1.034 0.903 ...
##  $ rotation: num [1:14, 1:14] 0.0946 -0.3559 -0.3632 -0.3598 -0.3523 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:14] "N" "P" "K" "Ca" ...
##   .. ..$ : chr [1:14] "PC1" "PC2" "PC3" "PC4" ...
##  $ center  : Named num [1:14] 22.4 45.1 162.9 569.7 87.5 ...
##   ..- attr(*, "names")= chr [1:14] "N" "P" "K" "Ca" ...
##  $ scale   : Named num [1:14] 5.53 14.95 64.84 243.58 41.01 ...
##   ..- attr(*, "names")= chr [1:14] "N" "P" "K" "Ca" ...
##  $ x       : num [1:24, 1:14] 0.132 0.528 -3.963 -3.241 -1.837 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:24] "18" "15" "24" "27" ...
##   .. ..$ : chr [1:14] "PC1" "PC2" "PC3" "PC4" ...
##  - attr(*, "class")= chr "prcomp"
```

Means and standard deviations of the original variables are stored in `chem.pca[['center']]` and `chem.pca[['scale']]`, respectively. Keep in mind that PCA generates *linear* combinations and that the analysis is sensitive to deviations from normality, as are linear models. 'sdev' contains the standard deviations associated with the components, an estimation of the variance explained by each component (although the proportion of variance explained is calculated from eigenvalues, these standard deviations squared).

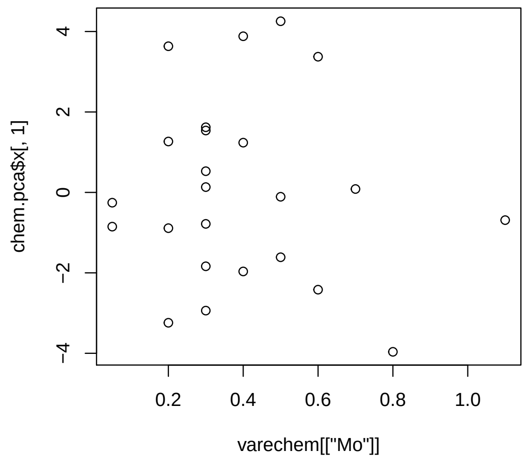
The loadings along each axis associated with the individual variables are stored in chem.pca[['rotation']]. 'Rotations' is a more intuitive way to think about these as that is what happens in the analysis: the axes associated with the variables are rotated until the greatest amount of remaining variation is accounted for. The variables with large rotations, positive or negative, have variance that is partitioned along that axis. This will become clearer in a moment.

The scores associated with each of the samples are stored in chem.pca[['x']]. These scores are used to plot the data associated with the samples, with the measurements of the original variables, in the ordination. Plotting these scores against the raw data, we can see how variation in each of the original variables is partitioned along the principal components, and how this relates to the values in chem.pca[['rotations']]. Potassium, with a negative rotation on the first axis, is negatively correlated with the scores along the first principal component. On the other hand, molybdenum, which has a loading of close to zero on the first axis, is not correlated with the scores along the first principal component.

```r
plot(varechem[["K"]], chem.pca$x[, 1])
```



```r
plot(varechem[["Mo"]], chem.pca$x[, 1])
```



Therefore the spread of the samples along the first axis is determined more by variation in potassium than in molybdenum, as seen from the biplot.

### 8.2.2 Correspondence analysis (CA)

PCA uses euclidean distances, geometrical distances in multidimensional space, to estimate the divergence between samples. This is appropriate for continuous variables exhibiting normal distributions, like the above example, but not for data that resemble frequencies or counts. CA uses a different approach to estimate divergence among samples, based on calculation of [chi-square] distances as for contingency tables. This property makes CA a better option than PCA for analysing tables of species counts in different environments.

CA is performed using cca() in 'vegan' by providing only the response matrix as an argument to the function. We will use the plant community data collected from the same locations as the soil data analysed above; these data are stored in 'varespec' and consist of observations relating to the percent cover associated with 44 plant species.

```
data(varespec)  # read data into your workspace
str(varespec)

## 'data.frame': 24 obs. of  44 variables:
##  $ Cal.vul: num  0.55 0.67 0.1 0 0 ...
##  $ Emp.nig: num  11.13 0.17 1.55 15.13 12.68 ...
##  $ Led.pal: num  0 0 0 2.42 0 0 1.55 0 0.35 0.07 ...
##  $ Vac.myr: num  0 0.35 0 5.92 0 ...
##  $ Vac.vit: num  17.8 12.1 13.5 16 23.7 ...
##  $ Pin.syl: num  0.07 0.12 0.25 0 0.03 0.12 0.1 0.1 0.05 0.12 ...
##  $ Des.fle: num  0 0 0 3.7 0 0.02 0.78 0 0.4 0 ...
##  $ Bet.pub: num  0 0 0 0 0 0 0.02 0 0 0 ...
##  $ Vac.uli: num  1.6 0 0 1.12 0 0 2 0 0.2 0 ...
##  $ Dip.mon: num  2.07 0 0 0 0 0 0 0 0 0.07 ...
##  $ Dic.sp : num  0 0.33 23.43 0 0 ...
##  $ Dic.fus: num  1.62 10.92 0 3.63 3.42 ...
##  $ Dic.pol: num  0 0.02 1.68 0 0.02 0.02 0 0.23 0.2 0 ...
##  $ Hyl.spl: num  0 0 0 6.7 0 0 0 0 9.97 0 ...
##  $ Ple.sch: num  4.67 37.75 32.92 58.07 19.42 ...
##  $ Pol.pil: num  0.02 0.02 0 0 0.02 0.02 0 0 0 0 ...
##  $ Pol.jun: num  0.13 0.23 0.23 0 2.12 1.58 0 0.02 0.08 0.02 ...
##  $ Pol.com: num  0 0 0 0.13 0 0.18 0 0 0 0 ...
##  $ Poh.nut: num  0.13 0.03 0.32 0.02 0.17 0.07 0.1 0.13 0.07 0.03 ...
##  $ Pti.cil: num  0.12 0.02 0.03 0.08 1.8 0.27 0.03 0.1 0.03 0.25 ...
##  $ Bar.lyc: num  0 0 0 0.08 0.02 0.02 0 0 0 0.07 ...
##  $ Cla.arb: num  21.73 12.05 3.58 1.42 9.08 ...
##  $ Cla.ran: num  21.47 8.13 5.52 7.63 9.22 ...
##  $ Cla.ste: num  3.5 0.18 0.07 2.55 0.05 ...
##  $ Cla.unc: num  0.3 2.65 8.93 0.15 0.73 0.25 2.38 0.82 0.05 0.95 ...
##  $ Cla.coc: num  0.18 0.13 0 0 0.08 0.1 0.17 0.15 0.02 0.17 ...
##  $ Cla.cor: num  0.23 0.18 0.2 0.38 1.42 0.25 0.13 0.05 0.03 0.05 ...
##  $ Cla.gra: num  0.25 0.23 0.48 0.12 0.5 0.18 0.18 0.22 0.07 0.23 ...
##  $ Cla.fim: num  0.25 0.25 0 0.1 0.17 0.1 0.2 0.22 0.1 0.18 ...
##  $ Cla.cri: num  0.23 1.23 0.07 0.03 1.78 0.12 0.2 0.17 0.02 0.57 ...
##  $ Cla.chl: num  0 0 0.1 0 0.05 0.05 0.02 0 0 0.02 ...
##  $ Cla.bot: num  0 0 0.02 0.02 0.05 0.02 0 0 0.02 0.07 ...
##  $ Cla.ama: num  0.08 0 0 0 0 0 0 0 0 0 ...
##  $ Cla.sp : num  0.02 0 0 0.02 0 0 0.02 0.02 0 0.07 ...
##  $ Cet.eri: num  0.02 0.15 0.78 0 0 0 0.02 0.18 0 0.18 ...
##  $ Cet.isl: num  0 0.03 0.12 0 0 0 0 0.08 0.02 0.02 ...
##  $ Cet.niv: num  0.12 0 0 0 0.02 0.02 0 0 0 0 ...
##  $ Nep.arc: num  0.02 0 0 0 0 0 0 0 0 0 ...
##  $ Ste.sp : num  0.62 0.85 0.03 0 1.58 0.28 0 0.03 0.02 0.03 ...
##  $ Pel.aph: num  0.02 0 0 0.07 0.33 0 0 0 0 0.02 ...
##  $ Ich.eri: num  0 0 0 0 0 0 0 0.07 0 0 ...
##  $ Cla.cer: num  0 0 0 0 0 0 0 0 0 0 ...
##  $ Cla.def: num  0.25 1 0.33 0.15 1.97 0.37 0.15 0.67 0.08 0.47 ...
##  $ Cla.phy: num  0 0 0 0 0 0 0 0 0 0 ...

spec.ca <- cca(varespec)

## Error:  argument "sitenv" is missing, with no default

plot(spec.ca)

## Error:  object 'spec.ca' not found
```

The plot is interpreted in the same way as the PCA biplot: the numbers represent the scores of the samples on the first two correspondence axes and the labels in red represent the loadings associated with the species. The plot does not include arrows, as these are reserved for another purpose that we will see later. Note that we did not use biplot() to produce the plot; this generic function does not work since the output is stored in a completely different way than for prcomp(). You could use str(spec.ca) to observe this, and what you would see is difficult to intepret without an understanding of the mathematics. Calling summary() on the object returns the information that is needed to interpret the analysis.

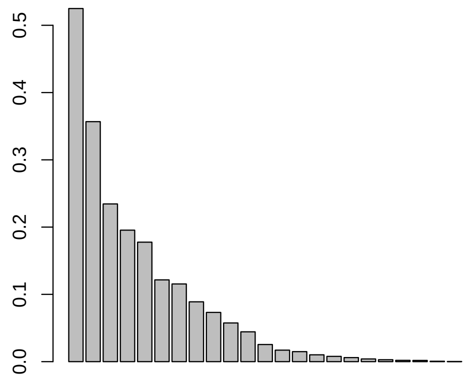
```
summary(spec.ca)

## Error:  object 'spec.ca' not found
```

Inertia is analogous to variance, and the value at the top of the output provides an indication of how much inertia is present in the data. The importance of the components is depicted in a similar way as the PCA output, with the exception that eigenvalues are given instead of standard deviations. This is followed by a table of 'species' scores and 'site' scores, equivalent to the 'rotations' and 'x' tables in the PCA example, respectively. (Keep in mind that should the input table be transposed, putting species in rows and sites in columns, the output will still label the loadings associated with columns as 'species' and rows as 'sites') By default, summary() returns values for six axes; this can be changed with the 'axes' argument (e.g., 'axes=12').

There was no scaling of the variables in our example but this may be desirable if, for example, the species vary greatly in total abundance along the whole gradient. decostand() in 'vegan' will transform the input matrix into a standarised from with the desired characteristics. For example, decostand(varespec,method='total') would provide a dataframe of the same dimensions as 'varespec' containing proportional relative abundances of each species per sample.

In 'ade4', CA is performed using dudi.coa(). It is worth performing an exercise with this function, just to demonstrate the similarities in the outcomes between the two approaches, but also the different ways in which these outputs are stored. 'ade4' has some functionality that is not present in 'vegan', particularly for some more advanced multivariate analyses, and an introduction to the output is useful.

```
library(ade4)
spec.ca.dudi <- dudi.coa(varespec)

## Select the number of axes:
```



```
## Error:  missing value where TRUE/FALSE needed
spec.ca.dudi
## Error:  object 'spec.ca.dudi' not found
```

Before performing calculations, a scree plot pops up and you are asked to indicate how many axes should be retained. This decision is not terribly important, but you should try to select those axes that appear to be explaining a large amount of variation.

The resulting object contains the transformed data (under 'tab'), species and site scores ('co' and 'li', respectively), eigenvalues ('eig'), and additional output. There is no convenient summary function. 'ade4' includes some flexible plotting functions that can be useful for exploring the outcomes of the analysis. However, these can also be frustrating due to large amount of information conveyed in a small space, particularly with large numbers of labels. We won't continue with these, but feel free to play around with scatter() and score() and worked examples can be found associated with the help pages.

### 8.2.3 Principal coordinates analysis (PCoA)

Just as euclidean distances are not appropriate for estimating the divergence between samples when analysing frequency data, some data types require an alternative to [chi-squared] distances. PCoA allows for the analysis of these data using distance matrices estimated from various multidimensional scaling indices. Many common metrics for estimating community dissimilarity are available in the vegdist() function (see the help page for these indices and their formulae). The distance matrix is used as the input for wcmdscale() in 'vegan', which performs the PCoA.

```
spec.pco <- wcmdscale(vegdist(varespec, method = "bray"), eig = T)
spec.pco$eig  # returns the eigenvalues
##  [1]  1.7552165  1.1334455  0.4429018  0.3698054  0.2453532  0.1960921
##  [7]  0.1751131  0.1284467  0.0971594  0.0759601  0.0637178  0.0583225
## [13]  0.0394934  0.0172699  0.0051011 -0.0004131 -0.0064654 -0.0133147
## [19] -0.0253944 -0.0375105 -0.0480069 -0.0537146 -0.0741390
spec.pco$points  # returns the site scores
##        [,1]      [,2]       [,3]      [,4]      [,5]       [,6]      [,7]
## 18  0.09459  0.159146  0.0744008 -0.202466  0.034230  0.0253456  0.011558
## 15 -0.31249  0.100328 -0.0622434  0.110845  0.030630  0.0007016  0.074153
## 24 -0.35107 -0.059541 -0.0380794  0.095061  0.211986 -0.0877436  0.154572
## 27 -0.32915 -0.170193  0.2316237  0.019111 -0.012922  0.0535773 -0.053461
## 23 -0.19259 -0.014592 -0.0056794 -0.209718  0.091026  0.0981331  0.022078
## 19 -0.06795 -0.145017 -0.0856457  0.002431 -0.029481  0.1282456  0.081429
## 22 -0.35121  0.003005 -0.1122296 -0.009057 -0.114983  0.0247217 -0.146074
## 16 -0.20889  0.131619 -0.1865496 -0.002034 -0.120909  0.1518848 -0.153954
## 28 -0.42973 -0.149930  0.2975932  0.274160 -0.139324 -0.0508562  0.001080
## 13  0.17363  0.248951 -0.0008768 -0.015750 -0.165087  0.0054470  0.090798
## 14 -0.16222  0.255851 -0.2160916 -0.105610 -0.147569 -0.2675697  0.019865
## 20 -0.16610  0.047986 -0.0767266 -0.117409  0.125610  0.0588493  0.002803
## 25 -0.37239 -0.010924 -0.1351577  0.030514  0.118374 -0.0589902 -0.056056
## 7   0.18856  0.360008  0.1579204 -0.003207  0.099925 -0.0469250 -0.127513
## 5   0.23883  0.431855  0.1648571  0.107794  0.076522 -0.0122703 -0.023288
## 6   0.22647  0.250582  0.1057272 -0.053213  0.062590 -0.0409204  0.003508
## 3   0.41035 -0.052963 -0.0425556  0.132421  0.006001  0.0163437 -0.075315
## 4   0.33600  0.115685 -0.0835108  0.090195 -0.155829  0.0684135  0.169621
## 2   0.39231 -0.232181  0.0035008  0.077937  0.010363 -0.0097599 -0.100127
## 9   0.25649 -0.409724 -0.0649106  0.007958  0.003604 -0.1423856  0.011832
## 12  0.23075 -0.260847 -0.0765246 -0.048866  0.038716  0.0340969 -0.011804
## 10  0.28972 -0.357493 -0.0609137 -0.002073  0.035312 -0.0706377 -0.058612
## 11  0.13456  0.007129 -0.0610821  0.130811  0.054989  0.1352968  0.097502
## 21 -0.02848 -0.248737  0.2731539 -0.309833 -0.113774 -0.0129984  0.065405
##          [,8]     [,9]     [,10]      [,11]     [,12]     [,13]      [,14]
## 18  0.101565 -0.05456 -0.023864 -0.0133300  0.067470 -0.039596 -0.0432256
## 15  0.095274  0.01267  0.006291 -0.1057067  0.042984  0.015689  0.0331262
## 24 -0.062795 -0.02904  0.075521  0.0752233  0.018429 -0.058365 -0.0065259
## 27  0.114519 -0.06482 -0.027803 -0.0456848  0.027151 -0.045824  0.0103189
## 23  0.079423  0.10367 -0.033770  0.0003276  0.020639  0.084708  0.0029154
## 19 -0.114146  0.01588 -0.116029 -0.0356104 -0.049739  0.010004 -0.0030087
## 22 -0.099934  0.02474  0.130014 -0.0698433  0.052666 -0.025181 -0.0167812
## 16  0.007859  0.01663  0.014156  0.0828421 -0.019284 -0.025941  0.0192208
## 28  0.022142 -0.01207 -0.019375  0.0267316 -0.060004  0.038989 -0.0148663
## 13 -0.013603 -0.14648  0.007016  0.0700743  0.090258  0.065987  0.0296657
## 14  0.059243  0.02178 -0.023737 -0.0080657 -0.061231  0.005549 -0.0005235
## 20  0.054677 -0.08760  0.054950  0.0258178 -0.144774  0.021385 -0.0034507
## 25 -0.096523  0.01593 -0.139610  0.0463788  0.067749 -0.002909 -0.0091388
## 7  -0.055966 -0.02744 -0.019480 -0.0498719 -0.018168  0.018971 -0.0557864
## 5   0.056047  0.16123  0.025938  0.0695783  0.007028 -0.022922  0.0297547
## 6  -0.134006 -0.06903 -0.002122 -0.0760607 -0.032059 -0.011774  0.0525005
## 3  -0.007784  0.01679  0.007804  0.0044064 -0.014967  0.042882 -0.0140399
## 4   0.022051  0.04529 -0.034476 -0.0251692 -0.020251 -0.080721 -0.0382481
## 2  -0.006987 -0.04565 -0.036999  0.0558521  0.008228 -0.013611  0.0066018
## 9   0.018173  0.03748  0.053506 -0.0098314  0.036486  0.057123 -0.0270397
## 12  0.074217 -0.05494  0.024583  0.0315927 -0.005487 -0.028260 -0.0083938
## 10  0.038488  0.02222 -0.025193 -0.0504730 -0.011653 -0.036910  0.0542932
## 11 -0.041105  0.02125  0.068948 -0.0354729  0.007850  0.045023 -0.0049891
## 21 -0.110829  0.07606  0.033731  0.0362948 -0.009323 -0.014297  0.0076206
##         [,15]
## 18  0.0040447
## 15 -0.0032202
## 24 -0.0064304
## 27  0.0102052
## 23 -0.0125873
## 19  0.0070799
## 22  0.0007868
## 16 -0.0088713
## 28 -0.0017036
## 13 -0.0045670
## 14  0.0063988
## 20 -0.0052213
## 25  0.0109008
## 7  -0.0197666
## 5   0.0090260
## 6   0.0105923
## 3   0.0374336
## 4  -0.0117468
## 2  -0.0305628
## 9  -0.0048871
## 12  0.0315931
## 10 -0.0181522
## 11 -0.0034051
## 21  0.0030606
```

Standardisation can be performed prior to the analysis using decostand() on the input dataframe or by providing a vector of weights using the 'w' argument in wcmdscale().

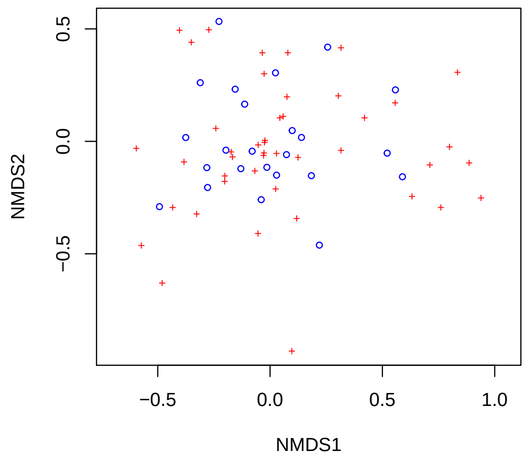
### 8.2.4 Non-metric multidimensional scaling (NMDS)

NMDS maximises the differences between samples on two or three dimensions, which can be particularly useful for visual representation of the dissimilarities between samples. metaMDS() in 'vegan' performs NMDS on either a table of community data or a distance matrix calculated from this table, as above.

```
spec.nmds <- metaMDS(varespec)

## Square root transformation
## Wisconsin double standardization
## Run 0 stress 0.1843
## Run 1 stress 0.2116
## Run 2 stress 0.1858
## Run 3 stress 0.214
## Run 4 stress 0.1962
## Run 5 stress 0.2097
## Run 6 stress 0.2066
## Run 7 stress 0.1858
## Run 8 stress 0.1843
## ... New best solution
## ... procrustes: rmse 0.0002684  max resid 0.0008183
## *** Solution reached

scores(spec.nmds)

##       NMDS1    NMDS2
## 18 -0.12999 -0.12150
## 15 -0.01419 -0.11561
## 24  0.25634  0.41895
## 27  0.58927 -0.15753
## 23  0.13987  0.01738
## 19  0.09861  0.04787
## 22  0.18395 -0.15272
## 16  0.02895 -0.15035
## 28  0.52133 -0.05250
## 13 -0.07963 -0.04390
## 14 -0.03936 -0.25947
## 20  0.07309 -0.05908
## 25  0.21947 -0.46146
## 7  -0.27816 -0.20516
## 5  -0.49234 -0.29044
## 6  -0.19644 -0.03925
## 3  -0.28142 -0.11682
## 4  -0.37527  0.01689
## 2  -0.22731  0.53310
## 9  -0.11305  0.16534
## 12  0.02417  0.30464
## 10 -0.15525  0.23190
## 11 -0.31061  0.26078
## 21  0.55800  0.22894
```

The procedure is iterative, and technical data are output to the screen as it runs. '*** Solution reached' should appear at the end of this output; if not, redo the analysis and increase the value associated with the 'trymax' argument (the default is 20). The stress in the final run is 0.182, which is acceptable; stress values greater than 0.3 are unsatisfactory suggesting that the dissimilarities are not effectively captured by these two dimensions. Increase the number of dimensions using the 'k' argument (the default is 2). Since the purpose of this approach is visualise the spread of the data in reduced dimensions, it makes sense to plot the data.

```
spec.nmds.plot <- plot(spec.nmds)
```



No labels are included in the plot, but the information needed to generate a plot for publication can be found using scores(spec.nmds.plot). The reason why we created an new object containing the plot is that we can use identify() to label the points that we click on. Use identify(spec.nmds.plot,'sites') to select the circles and identify(spec.nmds.plot,'species') to select the red crosses. By the way, ordiplot() and identify() can be used to identify points on a plot for other types of analyses in 'vegan'.

## 8.3 Factors driving data structure: constrained ordination

Up to this point, you have used ordination to look for structure in multivariate data. Once that structure is observed, you may have ideas about what factors are driving that structure and would like to test the hypotheses that these factors are actually important. To do this is to apply a constraint to the data, partitioning variation in the data to linear combinations of these specific factors (as above), and then comparing this partitioned variation to any remaining variation in the data.

### 8.3.1 Constrained ordination in an observational framework

In ecology, it is common to have collected data on species abundances in a variety of environments and to try to explain variation in those species' distributions due to characteristics of their environments. There two general approaches to do so and which one to use depends largely on the length of the environmental gradient under study.

#### 8.3.1.1 Redundancy analysis (RDA)

RDA is an extension of PCA and, as such, is appropriate for estimating the importance of constraining variables along short environmental gradients.

(PCA is also done via the rda() function in 'vegan'; to perform this, the user provides only the response matrix).

...

To perform RDA in 'ade4', use dudi.pca() and pcaiv().

#### 8.3.1.2 Canonical correspondence analysis (CCA)

CCA is an extension of CA, but serves the same purpose as RDA in relation to PCA. CCA is performed using cca() in both 'vegan' and 'ade4' packages. It's use in 'vegan' is very similar to rda(), so we will not deal with it further here.

### 8.3.2 Constrained ordination in an experimental framework

For factorial experimental designs, it is possible to estimate the effects associated with the factors and their interactions on ... adonis ...

## 8.4 Identifying groups

### 8.4.1 Cluster analysis

... are there groups? ...

¡¡¿¿+ hclust() @

## 8.5 Extra exercises

– analyse subset of the data, how many principal components are in the output?

– scaling and plots

– general non-normality in the data, how does this affect the ordinations

– plotting nmds, points with different symbols

# Chapter 9

# When things go wrong

## 9.1 Understanding errors in R

*This section is rather short! We will update this section using common errors encountered during the R course*

### 9.1.1 Reading data

#### 9.1.1.1 Numeric data not read as numeric

Sometimes, you read in your dataframe only to find out that some variables were not read correctly. It is very common for some bad fields in a numeric variable to mess things up. Let's look at an example using the `fluxtower` dataset, and an easy way to solve it.

```
# Read the fluxtower dataset:
flux <- read.csv("fluxtower.csv")

# Since we did not receive error messages, or warnings, we assume all is
# well.  Double check the 'str' statement:
str(flux)

## 'data.frame': 244 obs. of  8 variables:
##  $ TIMESTAMP: Factor w/ 234 levels "23-2-09 22:00",..: 1 2 3 4 5 6 7 8 29 30 ...
##  $ FCO2     : num  -0.743 0.614 -1.858 0.354 0.335 ...
##  $ FH2O     : num  -0.0167 0.0113 -0.0412 -0.0004 0.0003 0.0036 0.0095 0.0113 0.0019 -0.01 ...
##  $ ustar    : num  0.0296 0.0402 0.0982 0.0979 0.0398 0.0627 0.0576 0.0638 0.0552 0.0823 ...
##  $ Tair     : num  4.13 3.54 3.28 2.75 2.69 ...
##  $ RH       : num  74.3 77.2 78.7 81.8 82.1 ...
##  $ Tsoil    : num  9.25 8.66 8.19 7.79 7.3 ...
##  $ Rain     : Factor w/ 2 levels "#DIV/0!","0": 2 2 2 2 2 2 2 2 2 2 ...


# Note that 'Rain' is read as a factor, although we expect it to be
# numeric!  Note also that the first level of the factor is '#DIV/0!',
# which is generated by Excel when a division by zero is attempted.

# Read the flux data again, treat '#DIV/0!' as a missing value, as well as
# 'NA' which also appears in the dataset (in several places)
flux <- read.csv("fluxtower.csv", na.strings = c("#DIV/0!", "NA"))
```

```
# Check again
str(flux)

## 'data.frame': 244 obs. of  8 variables:
##  $ TIMESTAMP: Factor w/ 234 levels "23-2-09 22:00",..: 1 2 3 4 5 6 7 8 29 30 ...
##  $ FCO2     : num  -0.743 0.614 -1.858 0.354 0.335 ...
##  $ FH2O     : num  -0.0167 0.0113 -0.0412 -0.0004 0.0003 0.0036 0.0095 0.0113 0.0019 -0.01 ...
##  $ ustar    : num  0.0296 0.0402 0.0982 0.0979 0.0398 0.0627 0.0576 0.0638 0.0552 0.0823 ...
##  $ Tair     : num  4.13 3.54 3.28 2.75 2.69 ...
##  $ RH       : num  74.3 77.2 78.7 81.8 82.1 ...
##  $ Tsoil    : num  9.25 8.66 8.19 7.79 7.3 ...
##  $ Rain     : int  0 0 0 0 0 0 0 0 0 0 ...

# Now all variables are numeric except 'TIMESTAMP', which we don't expect
# to be.
```

### 9.1.2 Trying to treat characters as numeric

```
# Suppose you have a vector, which contains one character (by mistake!)
x <- c(5, 2, 9, 1, "a")

# Assuming you have numbers only, you try to calculate the mean:
mean(x)

## Warning:  argument is not numeric or logical:  returning NA

## [1] NA


# This returns a missing value (NA).  Take note of that warning message:
# it says 'argument is not numeric...'

# So, always double check your data:
str(x)

##  chr [1:5] "5" "2" "9" "1" "a"

# Your data are character ('chr')
```

## 9.2 Advanced examples

# Appendix A

# Description of datasets

## A.1 Tree Allometry

This dataset contains measurements of tree dimensions and biomass. Data kindly provided by John Marshall, University of Idaho.

**Variables:**

- `species` - The tree species (PSME = Douglas fir, PIMO = Western white pine, PIPO = Ponderosa pine).
- `diameter` - Tree diameter at 1.3m above ground (*cm*).
- `height` - Tree height (*m*).
- `leafarea` - Total leaf area ($m^2$).
- `branchmass` - Total (oven-dry) mass of branches (*kg*).

## A.2 Hydro dam

This dataset describes the storage of the hydrodam on the Derwent river in Tasmania (Lake King William & Lake St. Clair), in equivalent of energy stored. Data were downloaded from `http://www.hydro.com.au/water/energy-data`.

**Variables:**

- `Date` - The date of the bi-weekly reading
- `storage` - Total water stored, in energy equivalent (*GWh*).

## A.3 Rain

This dataset contains ten years (1995-2006) of daily rainfall amounts as measured at the Richmond RAAF base.

**Variables:**

- `Year`

- `DOY` - Day of year (1-366)
- `Rain` - Daily rainfall (*mm*)

## A.4 Weight loss

This dataset contains measurements of a person's weight over a period of about 3 months while he was trying to lose weight (data obtained from `http://jeremy.zawodny.com/blog/archives/006851.html`). This is an example of an irregular timeseries dataset (intervals between measurements vary).

**Variables:**

- `Date` - The Date in typical Excel format : dd/mm/yy
- `Weight` - The person's weight (*lbs*)

## A.5 Pupae

This dataset is from an experiment where larvae were left to feed on *Eucalyptus* leaves, in a glasshouse that was controlled at two different levels of temperature and $CO_2$ concentration. After the larvae pupated (that is, turned into pupae), the body weight was measured, as well as the cumulative 'frass' (larvae excrement) over the entire time it took to pupate.

**Variables:**

- `Ttreatment` - Temperature treatments ('ambient' and 'elevated')
- `CO2treatment` - $CO_2$ treatment (280 or 400 ppm).
- `Gender` - The gender of the pupae : 0 (male), 1 (female)
- `PupalWeight` - Weight of the pupae (*g*)
- `Frass` - Frass produced (*g*)

## A.6 Cereals

This dataset summarizes 77 different brands of breakfast cereals, including calories, proteins, fats, and so on, and gives a 'rating' that indicates the overall nutritional value of the cereal. **Variables:**

- `Cereal name` - Name of the cereal (text)
- `Manufacturer` - One of : "A","G","K","N","P","Q","R"
- `Cold or Hot` - Either "C" (cold) or "H" (Hot).
- `calories`
- `protein`
- `fat`
- `sodium`
- `fiber`
- `carbo`
- `sugars`

- `potass`
- `vitamins`
- `rating` - Nutritional rating (function of the above 8 variables).

*NOTE:* also included are the files 'cereal1.csv', 'cereal2.csv' and 'cereal3.csv', small subsets of the cereal data used in section 6.4.

## A.7 Flux tower data

This dataset contains measurements of $CO_2$ and $H_2O$ fluxes (and related variables) over a pine forest in Quintos de Mora, Spain. The site is a mixture of *Pinus pinaster* and *Pinus pinea*, and was planted in the 1960's.

Data courtesy of Victor Resco de Dios. **Variables:**

- `TIMESTAMP` - Character vector with date and time
- `FCO2` - Canopy CO2 flux ($\mu$ mol m$^{-2}$ s$^{-1}$)
- `FH2O` - Canopy H2O flux (mmol m$^{-2}$ s$^{-1}$)
- `ustar` - Roughness length (m s$^{-1}$)
- `Tair` - Air temperature (degrees C)
- `RH` - Relative humidity (%)
- `Tsoil` - Soil temperature (degrees C)
- `Rain` - Rainfall (mm half hour$^{-1}$)

## A.8 Pulse Rates before and after Exercise

A dataset on pulse rates before and after exercise. Taken from the OzDASL library (Smyth, GK (2011). Australasian Data and Story Library (OzDASL). `http://www.statsci.org/data`.)

More information at `http://www.statsci.org/data/oz/ms212.html`. **Variables:**

- `Height` - Height (cm)
- `Weight` - Weight (kg)
- `Age` - Age (years)
- `Gender` - Sex (1 = male, 2 = female)
- `Smokes` - Regular smoker? (1 = yes, 2 = no)
- `Alcohol` - Regular drinker? (1 = yes, 2 = no)
- `Exercise` - Frequency of exercise (1 = high, 2 = moderate, 3 = low)
- `Ran` - Whether the student ran or sat between the first and second pulse measurements (1 = ran, 2 = sat)
- `Pulse1` - First pulse measurement (rate per minute)
- `Pulse2` - Second pulse measurement (rate per minute)
- `Year` - Year of class (93 - 98)

**Note:** This data is in a TAB seperated file, so to read it in use `read.table("ms212.txt", header=TRUE)`.

## A.9 Weather data at the HFE

Data for the weather station at the Hawkesbury Forest Experiment for the year 2008.

Courtesy of Craig Barton.

**Variables:**

- `DateTime` - Date Time (half-hourly steps)
- `Tair` - Air temperature (degrees C)
- `AirPress` - Air pressure (kPa)
- `RH` - Relative humidity (%)
- `VPD` - Vapour pressure deficit (kPa)
- `PAR` - Photosynthetically active radiation ($\mu$ mol m$^{-2}$ s$^{-1}$)
- `Rain` - Precipitation (mm)
- `wind` - Wind speed (m s$^{-1}$)
- `winddirection` - Wind direction (degrees)

## A.10 Age and Memory

A dataset on the number of words remembered from list, for various learning techniques, and in two age groups. Taken from the OzDASL library (Smyth, GK (2011). Australasian Data and Story Library (OzDASL). http://www.statsci.org/data.)

More information at http://www.statsci.org/data/general/eysenck.html.

**Variables:**

- `Age` - Younger or Older
- `Process` - The level of processing: Counting, Rhyming, Adjective, Imagery or Intentional
- `Words` - Number of words recalled

**Note:** This data is in a TAB seperated file, so to read it in use `read.table("eysenck.txt", header=TRUE)`.

## A.11 Passengers on the Titanic

Survival status of passengers on the Titanic, together with their names, age, sex and passenger class. Taken from the OzDASL library (Smyth, GK (2011). Australasian Data and Story Library (OzDASL). http://www.statsci.org/data.)

More information at http://www.statsci.org/data/general/titanic.html. **Variables:**

- Name Recorded name of passenger
- PClass Passenger class: 1st, 2nd or 3rd
- Age Age in years (many missing)

- Sex male or female
- Survived 1 = Yes, 0 = No

**Note:** This data is in a TAB seperated file, so to read it in use `read.table("titanic.txt", header=TRUE)`.

## A.12 Xylem vessel diameters

Measurements of diameters of xylem (wood) vessels on a single *Eucalyptus saligna* tree grown at the Hawkesbury Forest Experiment.

Data courtesy of Sebastian Pfautsch.

**Variables:**

- `position` - Either 'base' or 'apex' : the tree was sampled at stem base and near the top of the tree.
- `imagenr` - At the stem base, six images were analyzed (and all vessels measured in that image). At apex, three images.
- `vesselnr` - Sequential number

- `vesseldiam` - Diameter of individual vessels ($\mu m$).