

# An Introduction to R for Beginners

Sasha Hafner\*

August 17, 2019

---

\*[sasha.hafner@eng.au.dk](mailto:sasha.hafner@eng.au.dk)

# Contents

<b>1 Orientation</b>	<b>6</b>
<b>2 Introduction to R</b>	<b>6</b>
2.1 R overview and history . . . . .	6
2.2 Finding and installing R . . . . .	7
2.3 Using R . . . . .	7
2.4 Getting started with the RStudio IDE . . . . .	9
2.5 Commands, assignment, and objects . . . . .	11
<b>3 Operators and functions</b>	<b>15</b>
3.1 Operators . . . . .	15
3.2 Functions . . . . .	17
<b>4 Code, data, and file organization</b>	<b>23</b>
4.1 Writing good code . . . . .	23
4.2 Data organization . . . . .	24
4.3 File organization and workflow . . . . .	25
<b>5 Getting help</b>	<b>29</b>
5.1 Help in general . . . . .	29
5.2 Help on functions . . . . .	29
5.3 Finding new functions . . . . .	31
<b>6 Working with add-on packages</b>	<b>34</b>
<b>7 The tidyverse packages</b>	<b>37</b>
<b>8 Data types and data objects in R</b>	<b>38</b>
8.1 Data types . . . . .	38
8.2 Overview of R data structures . . . . .	42
8.3 Vectors . . . . .	46
8.4 Matrices, arrays, and lists . . . . .	48

<b>9 Data frames, data import, and data export</b>	<b>52</b>
9.1 Your working directory . . . . .	52
9.2 Reading text files . . . . .	53
9.3 Reading spreadsheet files . . . . .	59
9.4 Creating data frames manually . . . . .	59
9.5 Working with data frames . . . . .	61
9.6 Writing data to files . . . . .	70
<b>10 Working with vectors</b>	<b>72</b>
10.1 Vector arithmetic and vectorized functions . . . . .	72
10.2 Working with character data . . . . .	74
<b>11 Factors</b>	<b>79</b>
<b>12 Dates and times</b>	<b>83</b>
12.1 The lubridate package . . . . .	84
12.2 Dates and times in base R . . . . .	87
<b>13 Exploratory data analysis</b>	<b>96</b>
13.1 Summary statistics . . . . .	96
13.2 Counts and contingency tables . . . . .	98
13.3 Histograms and other summary plots . . . . .	98
13.4 Normal quantile and cumulative probability plots . . . . .	106
<b>14 Basic data manipulation</b>	<b>110</b>
14.1 Indexing and subsetting . . . . .	110
14.2 Sorting data and locating observations . . . . .	127
14.3 Combining data frames and matrices . . . . .	135
<b>15 More on data manipulation: aggregating and summarizing data</b>	<b>140</b>
15.1 Counts . . . . .	140
15.2 Data aggregation and grouped operations . . . . .	145
15.3 The dplyr package and the magrittr operator . . . . .	159
15.4 Reshaping data . . . . .	165

<b>16 Introduction to base graphics</b>	<b>172</b>
16.1 Introduction to the <code>plot</code> function . . . . .	172
16.2 Adding data to plots . . . . .	178
16.3 Exporting graphics . . . . .	187
<b>17 ggplot2 graphics</b>	<b>189</b>
17.1 Components of ggplot2 graphics . . . . .	189
17.2 An example . . . . .	190
17.3 Exporting ggplot2 graphics . . . . .	195
<b>18 t tests</b>	<b>196</b>
18.1 One-sample . . . . .	196
18.2 Two-sample . . . . .	196
<b>19 Linear regression</b>	<b>201</b>
19.1 The <code>lm</code> function, model formulas, and statistical output . . . . .	201
19.2 Linear regression . . . . .	201
<b>20 Analysis of variance and analysis of covariance</b>	<b>241</b>
20.1 Analysis of variance (ANOVA) . . . . .	241
20.2 Analysis of covariance (ANCOVA) . . . . .	256
<b>21 Generalized linear models</b>	<b>266</b>
21.1 Introduction to <code>glm</code> . . . . .	266
21.2 Binary responses . . . . .	266
21.3 Count data . . . . .	276
<b>22 Random and mixed-effects models</b>	<b>280</b>
22.1 Introduction to the <code>lme4</code> package . . . . .	280
22.2 Random effects models . . . . .	281
22.3 Mixed effects models . . . . .	287
<b>23 Nonlinear regression</b>	<b>312</b>
23.1 A little bit on optimization . . . . .	312

23.2 The <code>nls</code> function . . . . .	314
23.3 The <code>nls.lm()</code> function . . . . .	317
23.4 Other functions . . . . .	317
<b>24 Basic R programming</b>	<b>318</b>
24.1 Loops and grouping . . . . .	318
24.2 Conditional statements . . . . .	321
24.3 Writing simple functions . . . . .	324
<b>25 Base graphics, part II</b>	<b>334</b>
25.1 Arranging multiple plots per page . . . . .	334
25.2 More on the <code>plot</code> function: arguments and values . . . . .	346
<b>26 Dynamic documents</b>	<b>348</b>
<b>27 Common mistakes</b>	<b>351</b>
<b>28 Where to go next</b>	<b>356</b>
<b>29 References</b>	<b>358</b>

# 1 Orientation

The objective of this course is to introduce participants to the use of R for data manipulation and analysis. It is intended for individuals with little or no prior experience in R. The topics that are covered are those that I think are the most important for getting started with R. By the end of the course, you should be able to complete all steps required for data analysis and visualization using R, including the use of some relatively sophisticated methods.

R can be used for a very wide range of approaches for analysis and graphics. In addition to the functions available in the “base” packages that are installed with R (you have installed these if you followed the instructions for getting ready for the course), more than 14 000 contributed packages are available, each with its own suite of functions. There are entire books written on some of these packages. By no means should this course be taken as a comprehensive introduction to R! However, the skills you learn can be used for many different tasks, and should provide a foundation for exploration of more advanced topics. In both the book and course, I try to focus on fundamental concepts—if you continue working with R, these topics will come up again and again.

R is often thought of as a tool for statistical computing, and therefore you might expect this course to focus primarily on statistical models. It will not. This is so for a few reasons.

1. Statistical functions are relatively easy to use in R. I will introduce you to some of the most common and most useful, and they will provide a good base on which to build. You may find that they are all you need.
2. When it comes to data analysis, visualization is at least as important as is building a statistical model. Visualization should be an early step in any analysis, and therefore is a major part of the material presented here.
3. Using R efficiently and effectively requires an understanding of the basics of R. Learning these basics takes some effort, but the effort is a worthwhile investment if you expect to spend any amount of time on data analysis in the future.
4. Simply getting data ready for analysis (including identifying and fixing problems) usually takes more effort than actually fitting a statistical model. The tools for getting data ready are powerful and flexible in R, and therefore they are a major part of the course.

# 2 Introduction to R

## 2.1 R overview and history

R is a programming language and a software system for computations and graphics. According to the R FAQ<sup>1</sup>, “[i]t consists of a language plus a run-time environment with graphics, a debugger, access to certain system functions, and the ability to run programs stored in script files.” R was originally developed in 1992 by Ross Ihaka and Robert Gentleman at the University of Auckland in New Zealand. The R language is a “dialect” of the S language<sup>2</sup>, which was developed (mainly) by John Chambers at Bell Laboratories. This software is currently maintained by the R Core Team, which consists of more than a dozen people, and includes Ihaka, Gentleman, and Chambers.

Many other people have contributed code to R since it was first released. R is open source; the source code for R is available under the GNU General Public License, meaning that users can modify, copy,

<sup>1</sup> <http://cran.r-project.org/doc/FAQ/R-FAQ.html#R-Basics>

<sup>2</sup> The S language is also used in the commercial software S-PLUS, which is very similar to R.

and redistribute the software or derivatives, as long as the modified source code is made available. The software is regularly updated, but changes are usually not major.

## 2.2 Finding and installing R

The R Core Team maintains a network of servers that contains installation files and documentation on R, called the Comprehensive R Archive Network, or CRAN. You can access it through <http://cran.r-project.org/>, or a Google search for CRAN R. R is available for Windows, Mac, and Unix-like operating systems. Installation files and instructions can be downloaded from the CRAN site by selecting one of the download links at the top. Although the graphical user interfaces (GUIs) and their menus differ across systems (if present at all), the R commands do not. R has been extended by users, and thousands of add-on packages (referred to just as “packages” by R users), which are modules of R functions and possibly data, usually related to a particular purpose, are available for free online.

## 2.3 Using R

There are two basic ways to use R on your machine: *interactively* through a graphical user interface (GUI) or shell, where R evaluates your code and returns results as you work, or by writing, saving, and then running R *script files*. Note that even if you use a GUI, R is not like typical “selection-and-response” software that you might be used to. Instead, you have to compose expressions (or commands) that R interprets<sup>3</sup>. This approach may make R more difficult to learn than a typical Windows or Mac program, but it comes with many advantages, including flexibility, efficiency, and repeatability.

It is possible use R as installed from CRAN. I don’t recommend this, but so you have an understanding of all the options, I’ll describe it here. If you are working in Windows, the R GUI you installed from CRAN will look something like the screenshot below (Fig. 1). The left window is the R console, where you enter commands.<sup>4</sup> The middle window is a simple “script editor”—more on that below.

The R GUI for Mac is a bit nicer—an older example is shown below (Fig. 2). Again, the left window is the console, where you enter commands, and the middle window is a script editor.

And for Unix-like operating systems, there is no R GUI, but instead you access R through a command-line shell, such as Bash—accessed through the purple window in the screenshot shown below (Fig. 3)<sup>5</sup>. In this case, you should definitely use a stand-alone text editor, like GVIM shown to the left in the screenshot below, to work with your scripts. (My favorite way to work in R is using Neovim with the Nvim-R plugin.)

But I don’t recommend these options! Instead, new users should work with the integrated development environment (IDE) called RStudio, produced by the company of the same name<sup>6</sup>. The RStudio IDE is available for Windows, Mac OS X, and Linux operating systems. It generally makes learning R easier and using R more efficient. It is now much more than a script editor, and includes tools for building packages and writing dynamic reports, among others. RStudio (the company) is a major presence in the R world. Its chief scientist Hadley Wickham has developed some of the most

<sup>3</sup> To get around writing code altogether there are some icon-driven programs that interface with R, e.g., R Commander (<http://socserv.mcmaster.ca/jfox/Misc/Rcmdr/>). But, I recommend you stick to writing R commands yourself—otherwise you will miss out on some of the major advantages of R.

<sup>4</sup> I’ll use the term “console” in this book to refer to refer to this window or a shell that is used to run R.

<sup>5</sup> You can also run R through a command-line shell in Windows or Mac OS X operating systems as well.

<sup>6</sup> Information and download available here: <http://rstudio.org/>.

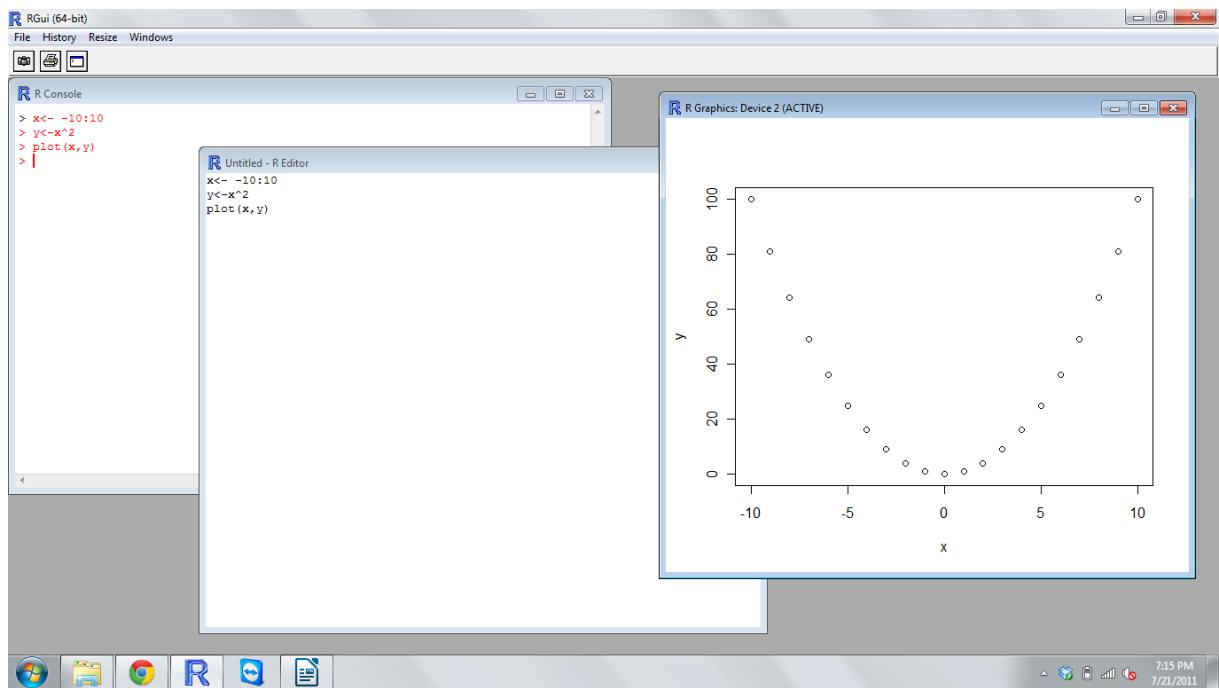


Figure 1: An R GUI in Windows 7.

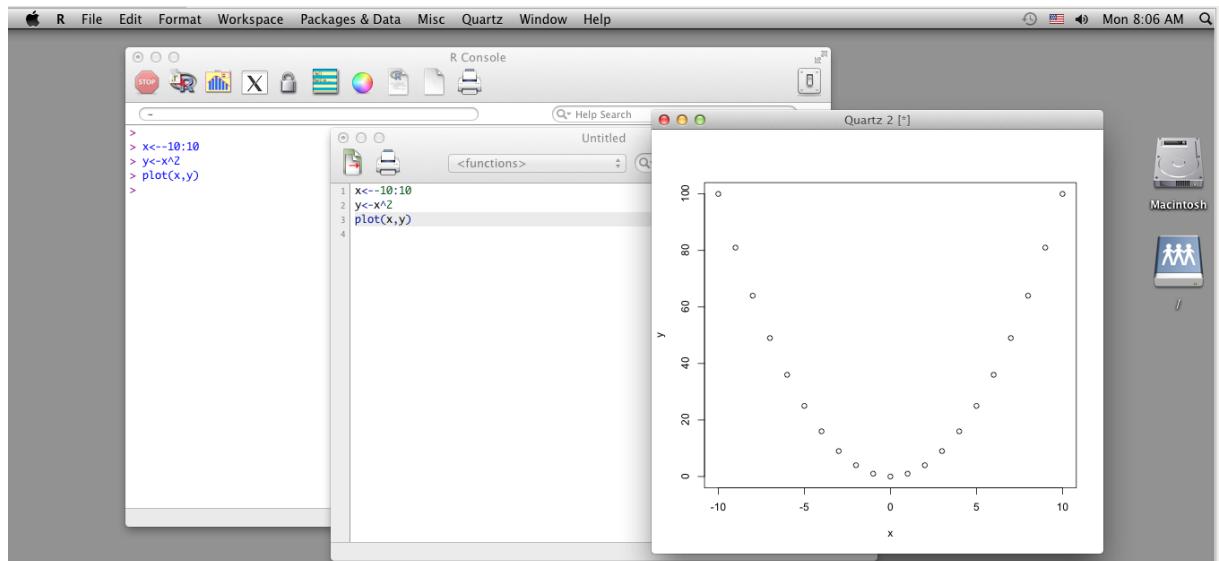


Figure 2: An R GUI in Mac OS X.

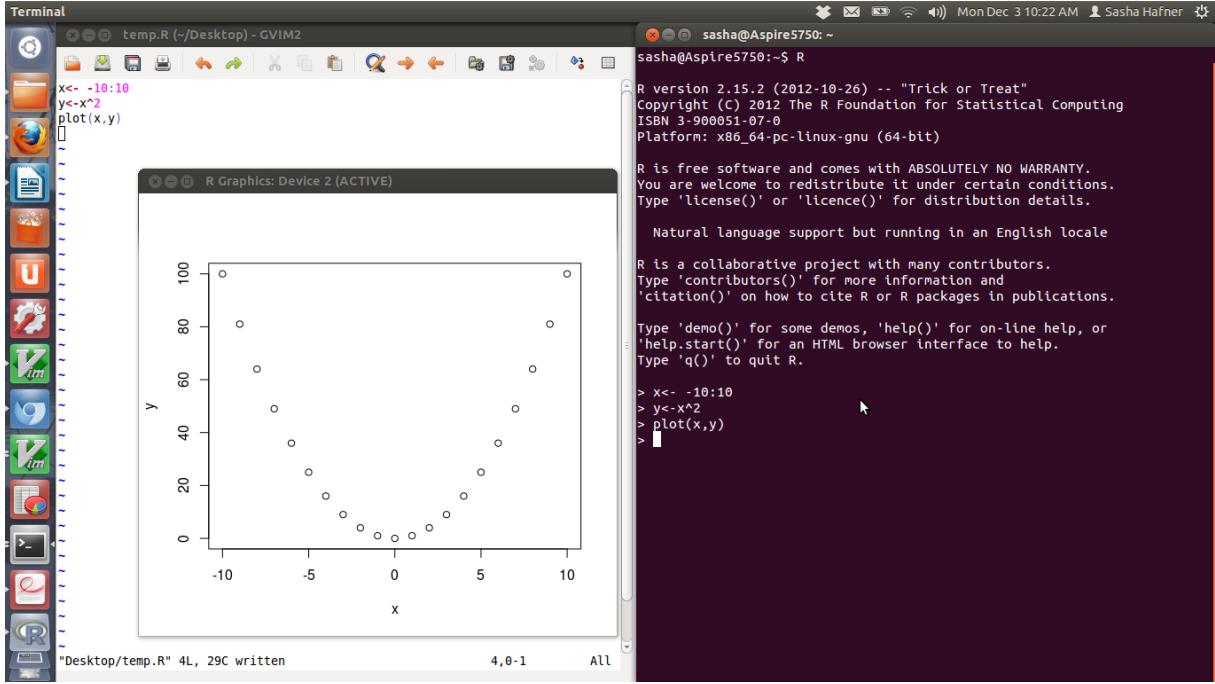


Figure 3: Running R through a command-line shell in Ubuntu Linux.

popular and useful add-on packages for R—some of which are described below. An example of the RStudio IDE (v. 1.0.136) is shown in Fig 4.

If you do not like the RStudio IDE (I still use something else), there are some useful (in some cases free) text editors or even IDEs available that can be set up with R syntax highlighting and other features. For a simple option for Windows, Notepad++ is a good choice. It is a general purpose text editor, and includes syntax highlighting for R, and the ability to send code directly to R with the NppToR plugin<sup>7</sup>. Several other options are available for Windows and other operating systems—see [http://www.sciviews.org/\\_rgui/projects/Editors.html](http://www.sciviews.org/_rgui/projects/Editors.html) for more information. For Linux, I recommend Neovim with the Nvim-R plugin—see [http://www.vim.org/scripts/script.php?script\\_id=2628](http://www.vim.org/scripts/script.php?script_id=2628) for details.

## 2.4 Getting started with the RStudio IDE

The most up-to-date information on the RStudio IDE is found online. There you can find a cheat sheet (<https://www.rstudio.com/resources/cheatsheets/>) with many of the important details. But I'll give an introduction here.

With R, you should almost always work with scripts. R script files (or scripts) are just text files that contain the same types of R commands that you can submit to a GUI or shell. Because everything you do in R can be recorded as a text command, script files provide an accurate way to record exactly what steps you carry out for a particular analysis. Once written, scripts can be run in R using an R GUI or a shell. All the code covered in this book will work if directly typed into the

<sup>7</sup> Notepad++ can be downloaded here: <http://notepad-plus-plus.org/download>. NppToR is available here: <http://sourceforge.net/projects/npttor/>. And, you can find instructions on using NppToR here: <http://jekyll.math.byuh.edu/other/howto/notepadpp/using.shtml>.

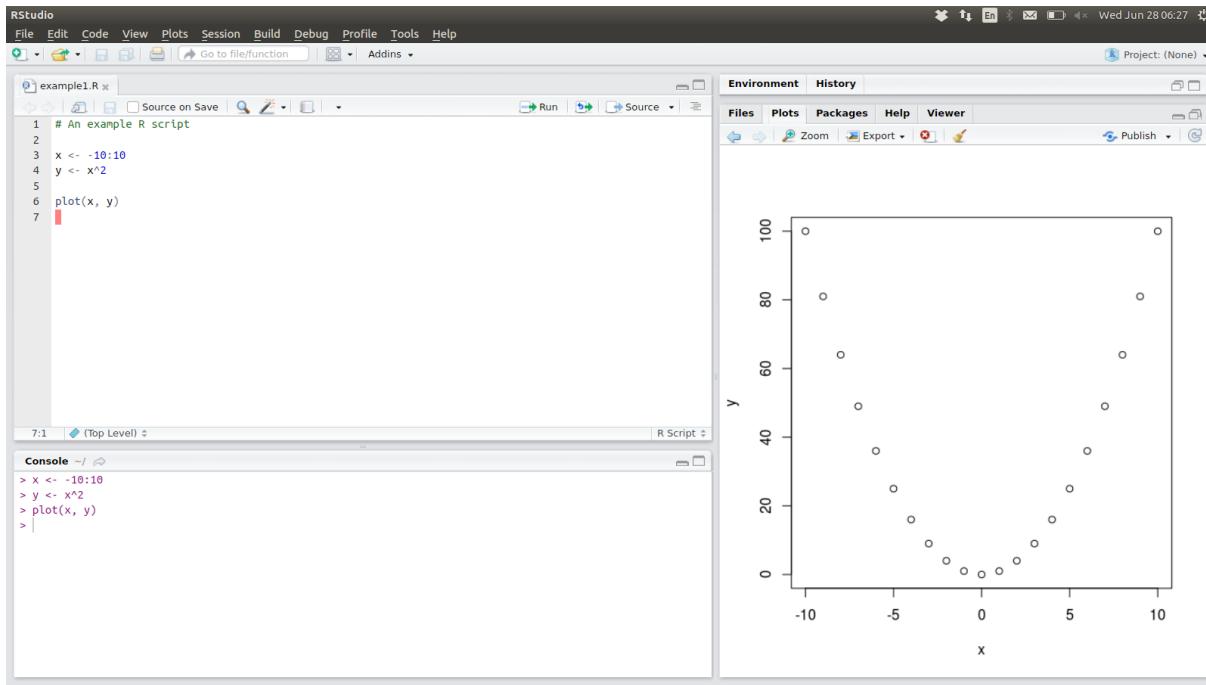


Figure 4: The RStudio IDE.

GUI, or it can be saved in a script file which can then be submitted to R<sup>8</sup>.

In RStudio, the lower left window is the console by default. You can click within it and interact directly with R. But working only interactively with R usually doesn't make sense. If your code is worth writing, it is probably worth saving. An advantage of R over software that requires numerous mouse clicks to run an analysis is that you can save an exact record of your analysis—in a script file. You can then repeat the same analysis in the future, perhaps with an updated data set, or you can modify your script to add new results or change some of the steps.

In RStudio, the upper left window is, by default, a script editor. It is where you will create, write, or modify scripts. To open a new script file, hit **Ctrl + shift + N**. On the first line, create a comment with the comment character **#**, e.g.,

```
# Notes from class 6 August
# S. Hafner
```

Then be sure to save your script with **Ctrl + s**. Select a logical location for saving the script! For example, how about a sub-directory in the directory for this course named “scripts”? Lastly, select the **Session** menu at the top, then **Set working directory**, and **To source file location**. The last step simply tells R where to look for files that are to be read in, and where to write files. It is good practice to always set the working directory to the location of your script or source file. You can now use this script for the remainder of the day or week (or perhaps you will want to create a new script for each section).

In general, you can run a script in R in the R GUI console or a command-line shell that is running R with the command `source("filename.R")`. Or, you can enter the command `R CMD BATCH`

<sup>8</sup> There is at least one difference between scripts and the GUI: with scripts, the results are not automatically printed—to manually print to the output file, use the function `print`.

filename.R<sup>9</sup> directly in a command-line shell (e.g., a Windows Command Prompt console). But in RStudio (and the similar editors mentioned above), there are keyboard shortcuts for running all or some of the code in a script. Some of the most useful are listed below.

Ctrl + Enter	Run current line or selection
Ctrl + Shift + Enter	Run all lines
Ctrl + Alt + B	Run from beginning to current line
Ctrl + Alt + E	Run from current line to end

## 2.5 Commands, assignment, and objects

In order to learn R, you need to learn the R language, and really, not much more. The R language is a mix of functional and object-oriented style [3]. For our purposes here, this means a few things. First, most (technically all) of the operations carried out in R are done by calling up functions, e.g., `sqrt(10)` will calculate the square root of 10. And the values that are stored in symbolic variables (e.g., you might store the value 10 in a variable called `x`) are not changed unless there is some kind of *assignment* involved—more on assignment in a bit. Both of these traits are characteristics of functional programming languages.

The variables you create and work with in R are called *objects*. Objects are really anything that can be assigned to a symbolic variable; data structures (e.g., a matrix) and functions (e.g., `sqrt`) are examples of objects. Data structures and other objects in R have what are called classes—really just a description of the type of object, such as “numeric vector” for a collection of numbers<sup>10</sup>. Many functions in R operate differently on different types of objects, and these functions are called *generic*. These traits are characteristics of object-oriented languages.

Advantages of the approach used for the R language should become clear as we begin to work with R. In general, it means less work for the user than languages without these characteristics.

The instructions you give R are called commands. The basic approach to using R interactively is to type a command and hit **Enter**, which causes R to evaluate what you typed, and generally to print the result. For example, to calculate  $10 - 6$ , enter `10 - 6` in your console

```
> 10 - 6
```

and you’ll get this result:

```
[1] 4
```

In your console, the `>` character is the prompt character, which indicates that R is ready to accept input. The result that R returns, `4`, has as `[1]` at the left side of the line. This number simply indicates the position of the adjacent element in the output—this will make more sense later when the output has more elements. Here, it isn’t needed.

For multi-line commands, an R console will display a “continuation symbol” `+` instead of the prompt `>`. So, for example, if we entered the same command as above, but hit **Enter** before typing the `6`, this is what we would see:

<sup>9</sup> To execute R scripts in batch mode, your operating systems needs to know where to find the R executable, and you may need to manually add the file location to the Environment variable.

<sup>10</sup> The class of numeric vectors is actually “`numeric`”.

```
> 10 -
+ 6
[1] 4
```

The continuation character indicates that R is waiting for something else—that you haven’t supplied a complete command. The way commands are displayed in this book looks a little different from what you’ll see in a console<sup>11</sup>

## 10-6

```
# [1] 4
```

First, the prompt character > is left out, to make it easy to copy and paste code directly into R<sup>12</sup>. The continuation symbol is also omitted. And, all the output lines are preceded with #, which is a comment character in R<sup>13</sup>, in order to facilitate copying and pasting.

For the above command, the result is printed to the screen and lost—there is no assignment involved<sup>14</sup>. In order to do anything other than the simplest analyses, you must be able to store and recall data. In R, you can assign the results of command to symbolic variables (as in other computer languages) using the assignment operator<sup>15</sup> <- . When a command is used for assignment, the result is no longer printed to the GUI console<sup>16</sup>. To see what is contained within a symbolic variable, type its name, and R will print the contents<sup>17</sup>.

```
x <- 10
x
```

```
# [1] 10
```

In the above command, R actually did two things: it created an object called x, and it gave x the value 10. If x had already existed, its original value would have been lost. So the assignment operator can be used to do two things: create objects and change the data stored in objects. In RStudio the assignment operator can be inserted with the keyboard shortcut Alt + -. Use it!

Note that the above command is very different from:

```
x < -10
# [1] FALSE
```

In the second case, putting a space between the two characters that make up the assignment operator causes R to interpret the command as “is x less than -10?”. However, spaces usually do not matter in R, as long as they do not separate a single operator or an object name. This, for example, is fine:

<sup>11</sup> The display format used in this book is actually just the default approach used for the `knitr` package, which I used to create this book (along with a LATEX compiler).

<sup>12</sup> Although you should avoid doing this wherever possible.

<sup>13</sup> More on this below.

<sup>14</sup> You might call this command an expression, to distinguish it from an assignment, but be aware that this distinction is not consistently used in the literature on R. Note that you can actually recall the last value printed to the console with `.Last.value`.

<sup>15</sup> The `assign` function is an alternative, but is seldom needed. One use is when the name for a variable is stored within another variable.

<sup>16</sup> Unless you surround the entire command in parentheses.

<sup>17</sup> Really, R runs the print method for the appropriate type of object. You may or may not see the entire contents.

```
x <- 10          + 1
```

The equal sign (`=`) can also be used as an assignment operator. However, in other cases the equal sign means something different (such as with column names when setting up a data frame), and this use is discouraged<sup>18</sup>.

You can recall a previous command in an R console by hitting the up arrow on your keyboard (as with Command Prompt and other command-line shells). This becomes handy when you are debugging code.

When you give R an assignment, such as the one above, the object referred to as `x` is stored in your *workspace*. You can see what is currently stored in the workspace by using the `ls` function.

```
ls()  
  
# [1] "x"
```

To remove objects from your workspace, use `rm`.

```
rm(x)  
x  
  
# Error in eval(expr, envir, enclos): object 'x' not found
```

You can combine `ls` with `rm` to remove all objects that you've added to your workspace.

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

But you can do the same thing by restarting R.

In general, R evaluates expressions from right to left, so, for example, if you want to assign the same value to several symbolic variables, you can use the following syntax.

```
x <- y <- z <- 1.0
```

There is also a right-pointing assignment operator, `->`, and a global assignment operator, `<<-`, but there is seldom a good reason to use these.

R is a case-sensitive language. This is true for almost everything in R, including symbolic variable names and function names<sup>19</sup>.

```
x <- 1+1  
x  
  
# [1] 2
```

<sup>18</sup> But some serious R users prefer it, e.g., Spector (2008) [21].

<sup>19</sup> It is not true for file names and file paths.

```
X
```

```
# Error: object 'X' not found
```

In R, commands can be separated by moving onto a new line (i.e., hitting the **Enter** key) or by typing a semicolon (;). Using a new line is the better approach—it makes for more readable code. As mentioned above, if a command is not completed in one line (by design or error), the typical R console prompt > is replaced with a +. Pay attention to what happens when you type the next example into your console.

```
x <-  
1 + 1
```

If you ever find that you are stuck in a bad command in the console, just hit the **Esc** key to get back to the regular prompt in Windows or Mac OSX, or **Ctrl+c** in Linux.

Commands in R operate on objects, where anything that can be assigned to a symbolic variable is an object. Objects include vectors, matrices, data frames, and functions. R objects that are used to hold data (most of the types of objects) are also referred to as data structures or data objects. We'll cover common data structures in sections 8 and 9.

In R, you use functions and operators to manipulate data structures<sup>20</sup>. In the examples above, we used only two operators: the assignment operator ( <- ) and the plus sign for addition (+). Many other operators will be discussed in the next section.

R also has a few built in constants, including  $\pi$  (**pi**) and **letters**.

```
pi
```

```
# [1] 3.141593
```

```
letters
```

```
# [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p"  
# [17] "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

When you try to close the R or Rstudio, it will ask you if you want to “save workspace image?”. This refers to the workspace that you have created—i.e., all the objects that you have loaded or created<sup>21</sup>. This can be handy for saving results that took a lot of (processor) time to create, but it is best to not rely on a saved workspace. Instead, you should save the commands that created all the objects in your workspace in a script file. In RStudio, you can see all commands that you have submitted since installing the software in the History window in the upper right. This could be handy if you lose some code that you meant to save in a script.

---

<sup>20</sup> Operators are actually implemented as functions as well in R, but generally you don't need to think about this.

<sup>21</sup> You can save all objects in your workspace or specific objects without closing R: see **save** and **save.image**.

## 3 Operators and functions

### 3.1 Operators

In R, you can carry out operations on objects using operators and functions. There are several operators that can be used in R, and the most of the common ones are summarized below.

#### *Arithmetic*

+	addition
-	subtraction
*	multiplication
/	division
$\wedge$	exponentiation
$\%/\%$	integer division
$\% \%$	modulo (remainder)

#### *Relational*

a == b	is a equal to b? (do not confuse with = )
a != b	is a not equal to b? (the ! symbol can be used to negate any expression)
a < b	is a less than b?
a > b	is a greater than b?
a <= b	is a less than or equal to b?
a >= b	is a greater than or equal to b?

#### *Logical*

!	not
&	and
	or
&&	sequential and
	sequential or
isTRUE	check whether logical value is true

#### *Indexing*

\$	part of a data frame or list
[]	part of a data frame, array, list
[[]]	part of a list
@	part of an S4 object

#### *Grouping commands*

{}

#### *Others*

:	sequences or combining factors
#	commenting (anything to the right is ignored)
;	alternative for separating commands
()	model formula specification
	order of operations, function arguments
%>%	magrittr operator-forward pipe

Using most of these operators is pretty straightforward. For example, to have R evaluate this mathematical expression:  $10^{3.5} + 87 \cdot 3.2 - \frac{1000}{18}$  we could use the following command.

```
10^3.5 + 87*3.2 - 1000/18
```

```
# [1] 3385.122
```

One character you should use a lot in your script files in the comment character #.

```
# This bit of text is a comment. Anything that comes after "#" is ignored by R.
```

Relational operators are generally used in between two values or variables to create a relational expression. For example, to see if **a**

```
a <- 50
```

is greater or equal to 100, use the following command.

```
a >= 100
```

```
# [1] FALSE
```

And logical operators can be used to combine or negate relational expressions. So, for example, to see if either condition is true: **a** is greater than or equal to 100 or equal to 50, use the vertical bar (i.e., pipe) character | as a logical “or”.

```
a >= 100 | a == 50
```

```
# [1] TRUE
```

This command will return TRUE if either relational expression is TRUE. Conversely, the ampersand & is used to create a compound expression with two simultaneous conditions<sup>22</sup>.

```
a > 0 & a < 100
```

```
# [1] TRUE
```

The order of mathematical operations in R (order of precedence) follow the standard mathematical order of operations, with some additional operators mixed in: indexing is higher than arithmetic and relational operators are lower.<sup>23</sup>

Parentheses can be used to control the order of operations, as in other programming languages. So,

```
7 - 2 * 4
```

```
# [1] -1
```

is different from:

---

<sup>22</sup> The functions **any** and **all** can be used to combine multiple relational expressions.

<sup>23</sup> You can find more details in section 10 of the R Language Definition [18].

```
(7 - 2) * 4
```

```
# [1] 20
```

## 3.2 Functions

Functions are very important in R: John Chambers (the primary author of S) writes that “[t]he central computation in R is a function call” [3, p. 111]. Functions require *arguments*, which are the objects that the function should act upon and other instructions for the function. For example, the function `sqrt` will calculate the square root of numeric data submitted to it.

```
sqrt(10)
```

```
# [1] 3.162278
```

In this case the number 10 is the only argument in the command. Arguments are always given in parentheses after the function name in a command that calls a function. Functions are written to accept certain arguments. In R there are “formal arguments”, which are the arguments within the function code, and “actual arguments”, which are the values or objects that you submit to functions. In this book I’ll generally just refer to “arguments”, and I think the distinction will be clear. While some functions are flexible in terms of the types and number of arguments they can accept, in general it is important to know what arguments are expected by a given function. There are two ways to see what arguments a given function expects: with the `args` function or by reading the help file.

```
args(sqrt)
```

```
# function (x)
# NULL
```

And you can bring up the help page for any function by typing `?`  followed by the function name. We’ll discuss help files more later in Section 5.

```
?sqrt
```

The actual arguments in R commands can also be named, i.e., by typing the name of the argument, an equal sign, and the argument value. Arguments specified in this way are also called tagged, or in the “tag = value” form. Both of the function calls given below accomplish the same thing using named arguments.

```
a <- 10
sqrt(x = a)
```

```
# [1] 3.162278
```

```
sqrt(x = 10)
```

```
# [1] 3.162278
```

With named arguments, R recognizes the argument name (keyword) (`x` above) and assigns the given actual argument (10 or `a` above) to the correct formal argument. When using named arguments, the order of the arguments doesn't matter. The alternative, which we've used above, is positional matching, where R does the matching based on the position of actual arguments. In this simple example, there is only one argument, so there is no way to get positional matching wrong.

```
sqrt(10)
```

```
# [1] 3.162278
```

In addition to `sqrt`, the typical mathematical functions are also available in R. A partial list is given below<sup>24</sup>. These functions all behave similarly to `sqrt`.

Function	Description
<code>log(x)</code>	natural log of <code>x</code>
<code>log10(x)</code>	base 10 log of <code>x</code>
<code>exp(x)</code>	$e^x$
<code>sin(x)</code>	sine of <code>x</code>
<code>cos(x)</code>	cosine of <code>x</code>
<code>tan(x)</code>	tangent of <code>x</code>
<code>sqrt(x)</code>	square root of <code>x</code>
<code>abs(x)</code>	absolute value of <code>x</code>

Let's take a look at a different function: `round`, which is used for rounding numeric data<sup>25</sup>. It has two arguments, the order of which can be found in the help file or with the `args` function.

```
args(round)
```

```
# function (x, digits = 0)
# NULL
```

Sometimes, as in this example, the names of the arguments alone provide enough information to learn how to use a new function. When this isn't the case, check the help file. So with `round`, to round  $\pi$  to four decimal places, we can use positional matching as in the following commands<sup>26</sup>.

```
pi
# [1] 3.141593
round(pi, 4)
# [1] 3.1416
```

<sup>24</sup> Check out <http://www.stats4stem.org/r-math-functions.html> for a longer list.

<sup>25</sup> One of five available functions. See `?round` for more information.

<sup>26</sup> `pi` is one of a few constants in R's base packages. See `?Constants` for more information.

This expression is equivalent to the two following commands, which use named arguments.

```
round(x = pi, digits = 4)
```

```
# [1] 3.1416
```

```
round(digits = 4, x = pi)
```

```
# [1] 3.1416
```

Notice how the `digits` argument seems to have a value associated with it in the result from `args(round)`? This is a default value that will be used if the user doesn't supply an actual argument. Default values can be found using the `args` function or in the help page.

Let's look at one more example. The `rnorm` function generates (pseudo-)random numbers from a normally distributed random variable. Let's take a look at the arguments.

```
args(rnorm)
```

```
# function (n, mean = 0, sd = 1)
# NULL
```

So there are three arguments, and two of them have default values.

```
x <- rnorm(1000)
mean(x)
```

```
# [1] 0.04046163
```

```
sd(x)
```

```
# [1] 1.00195
```

```
y <- rnorm(1000, mean = 25, sd = 4)
mean(y)
```

```
# [1] 25.03673
```

```
sd(y)
```

```
# [1] 3.87833
```

It usually makes sense to use positional arguments for only the first few arguments in a function. After that, named arguments are easier to keep track of.

Some functions include a special formal argument represented by an ellipsis .... This means the function can accept additional arguments, often ones that are passed onto another function that is called by the function. Any arguments listed in the help file or from args after ... can only be specified using naming.

Any time you want to call up a function, you must include parentheses after it, even if you are not specifying any arguments. If you don't, R will return the function code<sup>27</sup>.

It is not necessary to use explicit numeric values as function arguments. Symbolic variable names which represent appropriate objects can be used. It is also possible to use functions as actual arguments within functions. R will evaluate such expressions from the inside outward. I'll refer to this type of code as *nested*<sup>28</sup>. In fact, any expression can be used as an actual argument in a function call. This quality makes R very flexible, efficient, and easy (for code writing). For example, we could collapse the following four commands,

```
x <- pi/2 + 0.2
a <- sin(x)
b <- round(a, 2)
b

# [1] 0.98
```

into just one that does the same thing:

```
round(sin(pi/2 + 0.2), 2)

# [1] 0.98
```

To understand this expression, you need to read it from the inside out, e.g., “take  $\pi$ , divide by 2, add 0.2, take the *sine* of the result, and then round this result to two significant digits”. There is no explicit limit to the degree of nesting that can be used, but it is often easiest to assign intermediate results to symbolic variables, as in the example above. R evaluates nested expressions based on the values that functions return or the data represented by symbolic variables. For example, if a function expects character data for a particular argument, then you can use a call to a function that returns character data (e.g., `paste`) in place of explicit character data.

If nested expressions are confusing to you, you are not alone. There is now an alternative, the forward pipe operator `%>%` from the `magrittr` package. It is analogous to a “pipe” operator in other languages; it passes the object to its left to the function on the right as the first operator.

```
library(magrittr)

pi %>%
  divide_by(2) %>%
  add(0.2) %>%
  sin %>%
  round(2)

# [1] 0.98
```

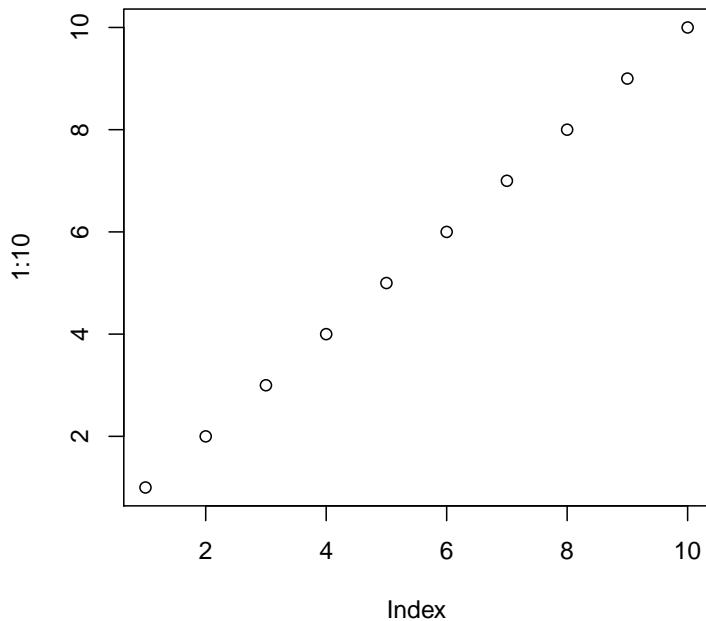
<sup>27</sup> Which can be useful if you want to quickly see how the function works. For actually working with the code for existing functions, the best bet is to download the package source code with `download.packages` with the `type` argument set to "source".

<sup>28</sup> This quality, called referential transparency, is part of the functional programming approach.

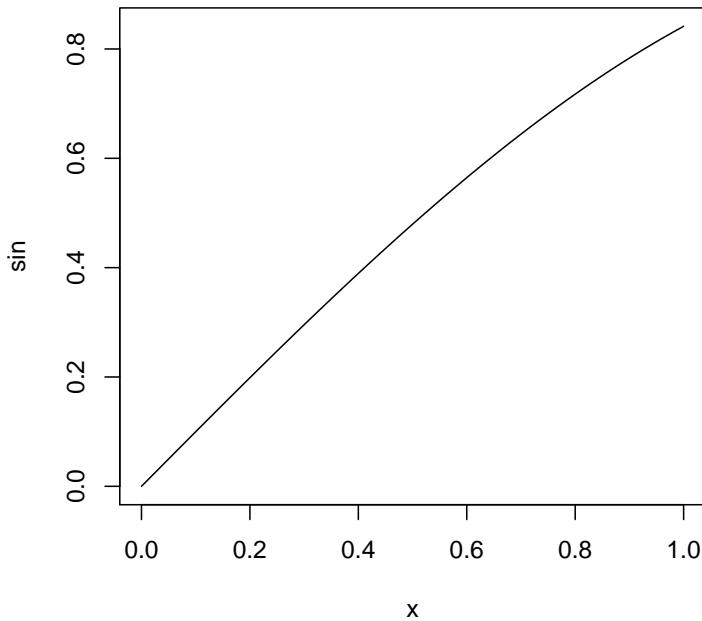
If the magrittr approach makes more sense to you, please feel free to use it. Most of remainder of this book uses the base R approach of nested expressions.

Functions in R often require that the actual arguments be of a certain type, e.g., `mean` cannot operate on character data. You can find information on the requirements for particular arguments within the help files. But, as mentioned above, many functions in R do accept multiple types of data and are called generic functions. What they do depends on the class of data submitted. For example, the `plot` function does something different for numeric data than it does for the output from a linear model. Another example: the `summary` function returns a different kind of summary for vectors, data frames, and linear model output. So `plot` and `summary` are generic functions.

```
plot(1:10)
```



```
plot(sin)
```



To see all the individual functions that are actually called up by a generic function (these functions are called *methods*), use `methods`.

```
methods(plot)
```

```
# [1] plot.acf*          plot.data.frame*    plot.decomposed.ts*
# [4] plot.default        plot.dendrogram*   plot.density*
# [7] plot.ecdf           plot.factor*       plot.formula*
# [10] plot.function       plot.hclust*       plot.histogram*
# [13] plot.HoltWinters*  plot.isoreg*       plot.lm*
# [16] plot.medpolish*    plot.mlm*         plot.ppr*
# [19] plot.prcomp*       plot.princomp*    plot.profile.nls*
# [22] plot.raster*       plot.spec*        plot.stepfun
# [25] plot.stl*          plot.table*       plot.ts
# [28] plot.tskernel*     plot.TukeyHSD*
# see '?methods' for accessing help and source code
```

```
methods(summary)
```

```
# [1] summary.aov          summary.aovlist*
# [3] summary.aspell*       summary.check_packages_in_dir*
# [5] summary.connection    summary.data.frame
# [7] summary.Date          summary.default
# [9] summary.ecdf*         summary.factor
# [11] summary.glm           summary.infl*
# [13] summary.lm            summary.loess*
# [15] summary.manova        summary.matrix
```

```

# [17] summary.mlm*           summary.nls*
# [19] summary.packageStatus*  summary.PDF_Dictionary*
# [21] summary.PDF_Stream*    summary.POSIXct
# [23] summary.POSIXlt        summary.ppr*
# [25] summary.prcomp*        summary.princomp*
# [27] summary.proc_time      summary.srcfile
# [29] summary.srcref         summary.stepfun
# [31] summary.stl*           summary.table
# [33] summary.tukeysmooth*
# see '?methods' for accessing help and source code

```

In general, the class of the first argument is what determines which method will be used. So if you were to call up the `summary` function with output from the `lm` function, the method that would be used is `summary.lm`.

There are actually two different sets of generic functions, called S3 and S4. It is probably not important to know the difference unless you are writing a package<sup>29</sup>. With S4 generic functions, use the `showMethods` function to view the available methods.

For tasks that you repeat, but which have no associated function in R, or if you don't like the functions that are available, you can write your own functions.

This introduction should give you an idea of the importance of functions in R, and how to use them, but it does not give any indication of the number and diversity of functions present in the “base” installation of R. Moreover, R is modular, and there are many more functions available in add-on packages that you can download from CRAN (as described in section 6). However, as described in the next section (Section 5), it is relatively easy to find functions for a particular task.

## 4 Code, data, and file organization

### 4.1 Writing good code

Good coding practice makes it easier for you and others to understand the code you've written. Consistently using a good approach also makes it easier to write code, since you do not have to think about small things like the best approach for nesting conditional expressions, but instead just follow the rule. In general I recommend following the guidelines compiled by Google (<https://google.github.io/styleguide/Rguide.xml>). To summarize my recommendations:

- Include comments in your scripts
- Use spaces between object names, operators, and constants
- Indent code in order to see clearly what belongs where
- Stick with lowercase object and column names
- Use blank lines to separate blocks of code
- Where there is variation in syntax within the R community (there is in several areas), try to avoid mixing different approaches

---

<sup>29</sup> One difference is that the method used with S4 generic functions can depend on more than just the first argument.

In the code chunk below, you can see examples of most of these points.

```
# Read in data
g <- read.csv("../data/us_gdp.csv")

# Create subsets for plot
gpre <- subset(g, year < 1929)
gdep <- subset(g, year > 1928 & year < 1940)
gpost <- subset(g, year > 1939)

# Blank plot (note indenting)
plot(gdp.real/1E6 ~ year, data = g, type = "n", xlab = "Year",
      ylab = "US GDP (trillion $)", las = 1)

# Plot three series
lines(gdp.real/1E6 ~ year, data = gpre, col = "blue")
lines(gdp.real/1E6 ~ year, data = gdep, col = "black")
lines(gdp.real/1E6 ~ year, data = gpost, col = "red")
```

RStudio can help a bit with good syntax. If you use the keyboard shortcut for the assignment symbol `<-` (Alt + -) it will always have spaces around it. And if you use autocomplete for argument names, they too will have spaces. Finally, RStudio will automatically indent in a usually appropriate way when a command extends across multiple lines.

## 4.2 Data organization

In order to make it easy to work with your data in R, you should try to follow these rules when creating data files:

1. Header rows are only present at the top of the file
2. Each column contains a single variable
3. Each row contains a single observation
4. Each file (or worksheet) contains a single block of data

This is probably best shown by example. See the files `silage_comp_original.xlsx` and `silage_comp_restruct.xlsx` for an example. Half of the original file is shown below in Fig. 5. This file violates rules 1, 2, 3 (although it is not clear in Fig. 5, there is another set of block of data to the right), and 4. This structure is pretty easy to understand and a person could interpret it without much trouble. But it would be very difficult to read the data into R and work with them.

The restructured file, the contents of which are shown in Fig. 6, in contrast, would be easy to work with. It follows all of the rules listed above. The only feature that is perhaps a bit odd is the use of multiple header rows. This turns out to be a convenient approach, however. The first two rows provide information for understanding the data, including units and more details on the analytes. These “extra” headers are simply skipped when reading that data into R.

Sometimes new R users have an inclination to avoid repetition in data files, and so find the value in column B in Fig. 6 to be inappropriate. Perhaps this has to do with a focus on data entry efficiency. If you have this perspective, try to get over it.

	A	B	C	D	E	F	G	H	I	J	K
1	Initial samples (time = 0 d)										
2	CONTROL										
3	Sample #	REP	WSC (% of DM)	NH3-N (% of DM)	VFAS (% of DM)	% LAC	%ACE	%1,2 PROP	%PROP	%ETOH	%BUT
4	782	1	6.8538168327	0.027563496279	0	0.02635782	0.10953209	0	0.18430902	0	0
5	783	2	4.8184928126	0.021054348548	0	0.03128378	0.08514032	0	0.10960043	0	0
6	784	3	6.4547007269	0.030083861071	0.02531126984	0.05508907	0.09859174	0	0.11831045	0	0
7	785	4	6.0182442559	0.031985419751	0	0.03177679	0.1161146	0	0.11535586	0	0
8											
9	LP										
10	Sample #	REP	WSC (% of DM)	NH3-N (% of DM)	VFAS (% of DM)	% LAC	%ACE	%1,2 PROP	%PROP	%ETOH	%BUT
11	786	1	6.0149035626	0.026111100824	0.0223549376	0.05464205	0.1040861	0	0.13176641	0	0
12	787	2	6.1707269391	0.020955252073	0	0.02842632	0.08901928	0	0.13375278	0	0
13	788	3	6.0421891922	0.024968908673	0.01764531219	0.05091475	0.13863687	0	0.14247514	0	0
14	789	4	4.3164107859	0.021975191514	0	0.00696246	0.0830558	0	0.11771189	0	0
15											
16	PS										
17	Sample #	REP	WSC (% of DM)	NH3-N (% of DM)	VFAS (% of DM)	% LAC	%ACE	%1,2 PROP	%PROP	%ETOH	%BUT
18	790	1	4.3121608914	0.02376575074	0	0.02583482	0.15956919	0	0.11794924	0	0
19	791	2	6.8534712874	0.024979879909	0.01764661214	0.03731931	0.10413928	0	0.13168528	0	0
20	792	3	6.641286909	0.034482802209	0	0.03147261	0.09311995	0	0.1360752	0	0
21	793	4	5.1577041535	0.022042234507	0	0.02399491	0.11490901	0	0.13771272	0	0
22											
23	LP+PS										
24	Sample #	REP	WSC (% of DM)	NH3-N (% of DM)	VFAS (% of DM)	% LAC	%ACE	%1,2 PROP	%PROP	%ETOH	%BUT
25	794	1	5.5594442367	0.024309250671	0	0.03582544	0.06233151	0	0.12500716	0	0
26	795	2	6.5071930344	0.02839076182	0	0.03152608	0.07504792	0	0.13199742	0	0
27	796	3	4.6476545885	0.022590253341	0	0.02908559	0.0968207	0	0.12526258	0	0
28	797	4	3.7946128363	0.02367665723	0	0	0.16355376	0	0.12866802	0	0
29											
30											

Figure 5: An example of a poor data structure. Data are on composition of silage (fermented animal feed) from a factorial experiment.

Data that don't follow these rules can often be restructured (reshaped) using R. We will discuss the reshape2 and tidyr packages for doing this. The original silage data (Fig. 5), however, could not be sorted out in R (at least not without a lot of work) and has to be corrected manually. Avoid this type of organization!

### 4.3 File organization and workflow

It is very easy to make a mess of file organization, and not too much harder to do a good job. In my experience, the reason most of us make a mess from the start is because we want to get to an answer quickly. But usually the small amount of time required for to set up a logical structure from the start pays off, even in the short term. In general, you should try to organize your files (data files, scripts, and output) at the start, or at least near the start—it may make sense to start working with a single script file and then divide it up after getting a better idea of what you need to do.

Windows and Mac may make file organization difficult by automatically hiding the extension of some files (e.g., ".csv" and ".xlsx" are generally not shown, and these files may appear the same). It is important to change this behavior so you can see all file extensions. Otherwise, how can you be sure whether you are opening the output file you think you are opening, or some other? You can find steps for Windows operating systems here: <http://windows.microsoft.com/en-us/windows/show-hide-file-name-extensions#show-hide-file-name-extensions=windows-7>, <http://ccm.net/faq/28650-windows-8-1-display-file-extensions>, and <http://www.laptopmag.com/articles/>

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1							VFAS (% of DM)							
2	Treatment	Time (d)	Sample #	REP	WSC (% of DM)	NH3-N (% of DM)	% SUC	% LAC	%ACE	%1,2 PROP	%PROP	%ETOH	%BUT	
3	trt	time	samp	rep	wsc	nh3	suc	lac	ace	prop12	prop	etho	but	
4	CONTROL	0	782	1	6.854	0.028	0.000	0.026	0.110	0.000	0.184	0.000	0	
5	CONTROL	0	783	2	4.818	0.021	0.000	0.031	0.085	0.000	0.110	0.000	0	
6	CONTROL	0	784	3	6.455	0.030	0.025	0.055	0.099	0.000	0.118	0.000	0	
7	CONTROL	0	785	4	6.018	0.032	0.000	0.032	0.116	0.000	0.115	0.000	0	
8	LP	0	786	1	6.015	0.026	0.022	0.055	0.104	0.000	0.132	0.000	0	
9	LP	0	787	2	6.171	0.021	0.000	0.028	0.089	0.000	0.134	0.000	0	
10	LP	0	788	3	6.042	0.025	0.018	0.051	0.139	0.000	0.142	0.000	0	
11	LP	0	789	4	4.316	0.022	0.000	0.007	0.083	0.000	0.118	0.000	0	
12	PS	0	790	1	4.312	0.024	0.000	0.026	0.160	0.000	0.118	0.000	0	
13	PS	0	791	2	6.853	0.025	0.018	0.037	0.104	0.000	0.132	0.000	0	
14	PS	0	792	3	6.641	0.034	0.000	0.031	0.093	0.000	0.136	0.000	0	
15	PS	0	793	4	5.158	0.022	0.000	0.024	0.115	0.000	0.138	0.000	0	
16	LP+PS	0	794	1	5.559	0.024	0.000	0.036	0.062	0.000	0.125	0.000	0	
17	LP+PS	0	795	2	6.507	0.028	0.000	0.032	0.075	0.000	0.132	0.000	0	
18	LP+PS	0	796	3	4.648	0.023	0.000	0.029	0.097	0.000	0.125	0.000	0	
19	LP+PS	0	797	4	3.795	0.024	0.000	0.000	0.164	0.000	0.129	0.000	0	
20	CONTROL	119	798	1	0.884	0.088	0.154	4.575	0.959	0.000	0.000	0.707	0	
21	CONTROL	119	799	2	0.686	0.068	0.150	4.337	0.958	0.000	0.000	1.108	0.073	
22	CONTROL	119	800	3	0.922	0.091	0.135	4.350	1.242	0.159	0.000	0.722	0	
23	CONTROL	119	801	4	0.843	0.055	0.141	4.440	0.872	0.000	0.000	0.798	0	
24	LP	119	802	1	0.521	0.091	0.121	4.201	0.810	0.089	0.000	0.846	0.153	
25	LP	119	803	2	0.674	0.079	0.138	4.456	1.124	0.000	0.000	0.932	0	
26	LP	119	804	3	0.586	0.069	0.147	4.487	1.620	0.000	0.000	1.174	0.061	
27	LP	119	805	4	0.641	0.090	0.137	4.658	1.220	0.172	0.554	0.000	0.067	
28	PS	119	806	1	2.103	0.077	0.098	4.560	1.014	0.000	0.000	0.235	0	
29	PS	119	807	2	1.033	0.027	0.041	2.553	1.234	0.163	0.000	0.072	0	
30	PS	119	808	3	1.515	0.073	0.169	4.661	1.372	0.235	0.000	0.455	0	
31	PS	119	809	4	1.964	0.080	0.060	3.776	1.001	0.000	0.000	0.161	0	
32	LP+PS	119	810	1	0.800	0.069	0.053	5.735	0.999	0.000	0.000	0.208	0	
33	LP+PS	119	811	2	1.086	0.040	0.047	5.920	1.105	0.000	0.000	0.212	0	
34	LP+PS	119	812	3	0.656	0.040	0.060	5.848	1.080	0.000	0.000	0.411	0	
35	LP+PS	119	813	4	0.892	0.087	0.074	6.213	1.108	0.000	0.000	0.413	0	
36														

Figure 6: A better way to structure a file containing the data shown in Fig. 5.

[windows-10-settings-to-change](http://www.idownloadblog.com/2014/10/29/how-to-show-or-hide-filename-extensions-in-os-x-yosemite/). And for Mac OS X, here: <http://www.idownloadblog.com/2014/10/29/how-to-show-or-hide-filename-extensions-in-os-x-yosemite/>.

Let's start by discussing folder or directory organization. Each project (for example, one chapter in your PhD dissertation), should have a separate folder containing all project files. The best way to organize data and script files within such a folder is up for debate, but it can be helpful to select and generally use a relatively consistent approach. The structure shown below is one option that I like.

```
main directory: project x
    sub-directory: analysis
        sub-sub-directories:
            data
            scripts
            intermediate
            figs
            output
    other sub-directories
    ...
```

Here, the "data" directory only contains original data, which is not edited if possible, unless you are the source of the data. If it is necessary to edit data files produced by someone else, the originals can be moved to an "original" sub-directory, and the changes documented in a log file or in a notes worksheet within the new (modified) spreadsheet. The "scripts" directory contains all R scripts (see below), and "figs" and "output" contain graphical and other output. The "intermediate" directory is used for intermediate data sets that need to be saved.

You should focus on automating every step in your data analysis workflow, e.g., try to avoid any data processing in Excel. Why? It is inefficient, it is not documented in a script (Do you remember exactly what you did? Are you sure?), and it cannot be automatically repeated when input data are updated. Instead, think of original data as read-only, and do the required cleaning and other steps using R. Output should be thought of as disposable—it can always be recreated by running your scripts. You can find a related discussion in this helpful introduction to R project management by Vince Buffalo: <https://swcarpentry.github.io/r-novice-gapminder/02-project-intro/>.

So what goes in the "scripts" folder? Again, there are many ways to organize your R code in scripts, but that's not an excuse for not organizing it! For simple analyses, you may find that a single script is sufficient. But if you find you are spending a lot of time looking through the script for certain sections, or if it takes time to run your complete script, consider splitting up your code into separate scripts. There is no reason to avoid very short scripts. Here is an approach that I like, based on a blog post by Rob J. Hyman: <http://robjhyndman.com/hyndsworth/workflow-in-r/>.

- functions.R Any and all functions that you've created for the analysis
- main.R This short script runs the complete analysis
- other\_stuff.R This and the following scripts carry out your analysis. Give these files descriptive names, not "other\_stuff"!
- ...

With this approach you can run your complete analysis with `source("main.R")`. Importantly, you can also see the relationship among your scripts by looking at main.R. For example, does one need

to be run before another? And lastly, with this approach, it is easy to find a certain line of code or identify errors, as compared with the use of a single script.

You may find it helpful to stick to a regular structure for the "other stuff" scripts. The following suggestion is mostly based on a Stack Overflow answer (<https://stackoverflow.com/questions/1429907/workflow-for-statistical-analysis-and-report-writing>) and a related blog post(<https://robjhyndman.com/hyndnsight/workflow-in-r/>).

- load.R For loading data
- packages.R For loading packages
- clean.R For cleaning data, fixing problems, merging data frames, etc.
- do.R For data analysis and generation of output

But for many projects, it will make sense to separate analysis code into multiple files, instead of just "do.R". And you may find it helpful to use designated scripts for making plots. If you load data in a designated script, you can simply call up data objects by name in following scripts (from your workspace), or save modified data frames to the "intermediate" directory.

It usually makes the most sense to set the working directory in R to your script location, i.e., "project x/analysis/scripts" in the above example. To refer to other directories, use relative paths, e.g. "../data" for data in the above example (this is the approach used in this book). This approach makes for less typing than absolute paths, and also allows you to copy and run your analysis on another machine (e.g., sharing it with someone else).

RStudio can help you with file organization as well, through the use of *projects*. When you create a project in RStudio<sup>30</sup> all files currently open are associated with the project, and a single directory is designated as the working directory. Once you open the project again, all your scripts will be open, and the working directory will be set to the correct location.

The topic of file organization and workflow is an active one in the R community, and new tools have been developed recently. If you spend a lot of time analyzing data, and find that the simple approach described above is still holding you back, or if you are simply interested in alternatives, check out the ProjectTemplate package.

---

<sup>30</sup> File, then New Project, and choose the appropriate option.

## 5 Getting help

### 5.1 Help in general

The sources that you learn to get started with R, such as this book, should continue to be useful as you continue to work with R. This book is not comprehensive, and there are several good introductory books on R that provide more information—some of them are listed at the end of this book. You can also find free detailed manuals on the CRAN website (<http://cran.r-project.org/manuals.html>, or else browse to CRAN (<http://cran.r-project.org/>) then select the “Manuals” link at the lower left). These manuals can also be accessed directly through R using the `RShowDoc` function<sup>31</sup>. Also, it helps to keep a copy of Short’s R Reference Card (Short 2005), which demonstrates the use of many common functions and operators in 4 pages (<http://cran.r-project.org/doc/contrib/Short-refcard.pdf>).

Globally, there are many R users<sup>32</sup> and they make for a very active online community, which you should take advantage of. There is a very large amount of information on R available online, but it is scattered among many different sites. Fortunately, R users have developed search engines for content on R.

For new R users, the list of Frequently Asked Questions on CRAN (<http://cran.r-project.org/faqs.html>) answers a lot of basic questions, beyond those on the R language. To search online material, a good first stop is the list of search engines given on the CRAN website (<http://cran.r-project.org/search.html>, or browse to CRAN and select the “Search” link on the upper left). The first one listed, “R site search” (<http://finzi.psych.upenn.edu/search.html>) works well. One of the best sources for discussion on R functions is the archives of mailing lists, which is the second search listed (<http://tolstoy.newcastle.edu.au/R/>). Another list of search engines can be found at <http://search.r-project.org/>. And Google itself (<http://www.google.com>) does a pretty good job too. To limit the results to R-related pages, adding “cran” seems to work well, or you can add site:r-project.org. You can access some of these search engines by submitting commands to R directly, using the `RSiteSearch` function. These methods are good for more than just finding new functions. You can also find a lot of answers and discussion on problems that users have in R.

If you’ve exhausted these options, you can ask for help, and send a message to one of the mailing lists for R users. There are a few large mailing lists for R users that are very active. If you are looking for help on R, the one you should sign up for is “R-help”. You can sign up for any of the lists here: <http://www.r-project.org/mail.html>. Before posting a question, be sure to search the mailing list archives, and check the posting guide (<http://www.r-project.org/posting-guide.html>). Individuals on the mailing list can provide helpful answers to even very specialized questions, but they are not shy about telling users to go back and read the posting guide or the manual if you ask for the thousandth time “How can I get my Excel data into R?”.

### 5.2 Help on functions

Each function in R has a help file associated with it that explains the syntax and usually includes examples. Help files are concisely written, and can take a little practice to get used to, but they usually provide all you need to know about a function. You can bring up a help file for any function by typing `?`  and then the function name. Or, use the search box in the lower right **Help** window in RStudio.

---

<sup>31</sup> For example, `RShowDoc("R-lang")`.

<sup>32</sup> Probably millions. See: <http://bigcomputing.blogspot.ca/2011/03/growth-of-r-users.html> and <http://r.789695.n4.nabble.com/OT-How-many-useRs-td917031.html>

```
?aov
```

The screenshot shows the R Help Viewer window. The title bar says "?aov". The menu bar includes "Files", "Plots", "Packages", "Help", and "Viewer". Below the menu is a toolbar with icons for back, forward, search, and help. A search bar contains "aov". The main content area has a header "R: Fit an Analysis of Variance Model" and a "Find in Topic" button. The topic page for "aov {stats}" is displayed under "R Documentation". The page content includes:

- Description**: Fit an analysis of variance model by a call to `lm` for each stratum.
- Usage**:

```
aov(formula, data = NULL, projections = FALSE, qr = TRUE,
     contrasts = NULL, ...)
```
- Arguments**:
  - `formula`: A formula specifying the model.
  - `data`: A data frame in which the variables specified in the formula will be found. If missing, the variables are searched for in the standard way.
  - `projections`: Logical flag: should the projections be returned?
  - `qr`: Logical flag: should the QR decomposition be returned?
  - `contrasts`: A list of contrasts to be used for some of the factors in the formula. These are not used for any Error term, and supplying contrasts for factors only in the Error term will give a warning.
  - `...`: Arguments to be passed to `lm`, such as `subset` or `na.action`. See 'Details' about weights.
- Details**:

...

This will bring up the help file for the `aov` function (used for analysis of variance, as described in Section 20). Once the help file is opened, you can search within it.

For operators (e.g., `*` or `[]`) or flow control terms (e.g., `if`), you have to use quotes.

```
?"*"  
?"["  
?"if"
```

For the most part, help files are self-explanatory. However, a few points are worth noting. The first section (*Description*) gives a description of the function or functions. The second section (*Usage*) shows how the functions are called up, i.e., the function name(s); the names of the arguments; their order and default values; and whether or not additional arguments are accepted (remember the ellipsis, `...`?). The *Arguments* section provides information on the arguments, including the allowed

object types for each argument. Not following these descriptions will probably result in an error message. For some functions, particularly generic ones, the *Usage* section can be a bit confusing. Start by looking at the “S3 method” or “Default S3 method”, or bring up the help file for the appropriate method if it exists. Methods for generic functions may or may not have their own help files—`summary.lm` (an S3 method), for example does, and you can view the help file with the following command.

```
?summary.lm
```

The *Value* section describes the object returned by the function. At the very bottom of help files there is an *Examples* section, which can have very useful demonstrations of how to use the function. You should be able to copy and run the exact commands given in this section. Help files include a *See Also* section, which lists related functions—this is a really good place to find new functions that carry out related operations.

The `?function_name` command is actually a shortcut to the `help` function. In some cases, such as viewing S4 methods for generic function, you’ll have to use it. So, for example, if you wanted to view the `plot` method for `raster` objects as defined in the `raster` package, you would use the following command.

```
help(plot, package = "raster")
```

### 5.3 Finding new functions

What if you aren’t sure what function you need for a particular task? How can you know what help file to open? There is a complete manual of all R functions in the base packages available through CRAN (<http://cran.r-project.org/manuals.html>), but at 3500 pages it’s not practical to read through, and anyway it just covers the based packages—not the 6000 user-contributed packages.

Fortunately, you can go far in R without using a large number of functions. And it is pretty easy to find new R functions. There are several different approaches that you can use, however.

You can search for a particular keyword within all the help files on your machine with `??keyword`<sup>33</sup>. For example,

```
??Kruskal
```

```
Help files with alias or concept or title matching
'Kruskal' using fuzzy matching:
```

```
MASS:::isoMDS      Kruskal's Non-metric Multidimensional
                   Scaling
stats:::kruskal.test   Kruskal-Wallis Rank Sum Test
```

```
Type '?PKG::FOO' to inspect entries 'PKG::FOO', or
'TYPE?PKG::FOO' for entries like 'PKG::FOO-TYPE'.
```

<sup>33</sup> This is a shortcut for the `help.search` function. To use search terms that contain a space, surround them with quotes.

In this case, you can leave out the `stats::` part of the command, which is the name of the package that contains the `kruskal.test` function, since the `stats` package is in the base packages and so is automatically loaded when R is started. If the package were not loaded, you would need the `package.name::` bit<sup>34</sup>.

```
?kruskal.test
```

The screenshot shows the R Help Viewer window. The title bar says "kruskal.test {stats}" and the right pane is titled "R Documentation". The main content area displays the documentation for the Kruskal-Wallis Rank Sum Test.

## Kruskal-Wallis Rank Sum Test

### Description

Performs a Kruskal-Wallis rank sum test.

### Usage

```
kruskal.test(x, ...)
## Default S3 method:
kruskal.test(x, g, ...)

## S3 method for class 'formula'
kruskal.test(formula, data, subset, na.action, ...)
```

### Arguments

- `x` a numeric vector of data values, or a list of numeric data vectors. Non-numeric elements of a list will be coerced, with a warning.
- `g` a vector or factor object giving the group for the corresponding elements of `x`. Ignored with a warning if `x` is a list.
- `formula` a formula of the form `response ~ group` where `response` gives the data values and `group` a vector or factor of the corresponding groups.
- `data` an optional matrix or data frame (or similar: see [model.frame](#)) containing the variables in the formula `formula`. By default the variables are taken from `environment(formula)`.

To search for objects (including functions) that include a particular text string, you can use the `apropos` function:

```
apropos("mean")

# [1] "colMeans"          ".colMeans"        "kmeans"           "mean"
# [5] "mean.Date"         "mean.default"    "mean.difftime"   "mean.POSIXct"
# [9] "mean.POSIXlt"      "rowMeans"        ".rowMeans"       "weighted.mean"
```

<sup>34</sup> But the package would need to be installed either way. Package installation is described in Section 6.

It will only search for functions that are in loaded packages or for other objects that are in your workspace. The `apropos` function with accept regular expressions<sup>35</sup>.

RStudio has a very useful autocomplete feature.<sup>36</sup> To use it, hit the **Tab** key after typing part of the name of a function or other object, and R will list possible matches (or fill in the remainder of the word if there is only one possible match). You can scroll through the listed options using the up and down arrows on your keyboard. While this feature isn't designed to be a search tool, it can be helpful when you can't quite remember the exact name of a function, but can remember the letters that it starts with.

The methods described above in Section 5.1 are also useful for finding new functions. Taken together, here is an outline of one possible approach you might use for finding new functions. Other orders, or only a few of the items, might be a good approach as well, but you can't go wrong by working through all of these steps until you've found the information you need.

1. Guess
2. `apropos` function (searches loaded packages only)
3. `?"keyword"` (searches installed packages only)
4. Read help files
5. R site search (<http://finzi.psych.upenn.edu/search.html>)
6. Mail archive search (<http://tolstoy.newcastle.edu.au/R/>)
7. Other R search (<http://cran.r-project.org/search.html> or <http://search.r-project.org/>)
8. Google search (<http://google.com>, maybe with site:r-project.org added)
9. Send a message to a mail list (<http://www.r-project.org/mail.html>)

---

<sup>35</sup> See the help file for `regex` for more information on regular expressions in R.

<sup>36</sup> Any interactive version of R (including the Windows and Mac GUIs) include an autocomplete feature.

## 6 Working with add-on packages

Not all the functionality of R is immediately available upon starting the software. R is modular, and functions and data sets are organized in “packages”, which may contain R code, compiled Fortran or C code, or data sets. Many functions (including all of those that we’ve worked with so far) come with the R “base packages”, so they are loaded and ready to go as soon as you open R<sup>37</sup>. While the base packages include many useful functions, for specialized procedures you may need content from add-on packages.

The CRAN website <http://cran.r-project.org/> currently lists more than 4000 add-on packages that contain functions and data that users have contributed—select “Packages” from the left to view a list. To use the functions in a non-base R package, you need to first install and then load the package. Packages can be installed via GUI menus, but it is probably quicker to use the command<sup>38</sup>:

```
install.packages("package_name")
```

where "package\_name" should be replaced with the actual name of the package you want to install, for example:

```
install.packages("dplyr")
```

You may be asked to select a “mirror” (server) to use to download the package. What this installation process actually does is to download the dplyr package and save a copy in a “library”, which is simply a directory (i.e., “folder”) that has packages<sup>39</sup>. Installation is a one-time process<sup>40</sup>. But, packages must be “loaded” into your search path each time you want to use them. This is very simple, e.g., to load the package `dplyr`, use the following command.

```
library(dplyr)
```

To avoid loading packages each time you use R, you can add the appropriate commands to a file named `Rprofile.site`<sup>41</sup> that will be executed every time you start R.

You can see what packages are loaded (i.e., in your search path) with the `search` function.

```
search()
```

```
# [1] ".GlobalEnv"           "package:magrittr"   "package:knitr"  
# [4] "package:stats"        "package:graphics"  "package:grDevices"  
# [7] "package:utils"         "package:datasets"  "package:methods"  
# [10] "Autoloads"            "package:base"
```

But the `sessionInfo` function may be more useful:

<sup>37</sup> You can find a list of these packages here: <http://cran.r-project.org/doc/FAQ/R-FAQ.html>. Or, the command `getOption("defaultPackages")` will list them as well. This option is will be read from an environment variable `R_DEFAULT_PACKAGES` if it exists, and so provides one way to load additional packages automatically by default.

<sup>38</sup> If you leave out the first argument (named `pkgs`), R will display a list of packages to select from.

<sup>39</sup> For Windows, `install.packages` downloads and installs a binary file, while for UNIX-like operating systems, it downloads a source file, and then builds and installs the package with R CMD INSTALL. In Mac OS X, installation generally goes along with other UNIX-like operating systems. See [http://cran.r-project.org/doc/manuals/R-admin.html#Add\\_002don-packages](http://cran.r-project.org/doc/manuals/R-admin.html#Add_002don-packages) for more info.

<sup>40</sup> But it is a good idea to update your packages before installing new packages, with `update.packages`.

<sup>41</sup> See details in Section 11.

```

sessionInfo()

# R version 3.4.4 (2018-03-15)
# Platform: x86_64-pc-linux-gnu (64-bit)
# Running under: Ubuntu 18.04.2 LTS
#
# Matrix products: default
# BLAS: /usr/lib/x86_64-linux-gnu/openblas/libblas.so.3
# LAPACK: /usr/lib/x86_64-linux-gnu/libopenblas-r0.2.20.so
#
# locale:
# [1] LC_CTYPE=en_US.UTF-8          LC_NUMERIC=C
# [3] LC_TIME=en_US.UTF-8          LC_COLLATE=en_US.UTF-8
# [5] LC_MONETARY=en_US.UTF-8       LC_MESSAGES=en_US.UTF-8
# [7] LC_PAPER=en_US.UTF-8          LC_NAME=C
# [9] LC_ADDRESS=C                  LC_TELEPHONE=C
# [11] LC_MEASUREMENT=en_US.UTF-8    LC_IDENTIFICATION=C
#
# attached base packages:
# [1] stats      graphics   grDevices utils      datasets  methods
# [7] base
#
# other attached packages:
# [1] magrittr_1.5 knitr_1.19
#
# loaded via a namespace (and not attached):
# [1] compiler_3.4.4  tools_3.4.4    codetools_0.2-15
# [4] stringi_1.1.6  highr_0.6     digest_0.6.15
# [7] stringr_1.2.0  evaluate_0.10.1

```

Of the “base” packages listed above, the only one you might ever need to think about is `datasets`, which contains example data sets. You can view of list of these data sets with `library(help = datasets)`, and access individual data sets through “lazy loading” by simply typing their name. Some of the data sets that come with other packages need to be loaded first with a call to the `data` function.

In documentation, packages may be identified as packages by braces: `{somepackage}` means “somepackage” is a package. But in other cases, you may see “`package: somepackage`” or even “From `somepackage`”<sup>42</sup>.

You can find information on specific packages through CRAN, by browsing to <http://cran.r-project.org/> and selecting the “Packages” link on the lower left. Each package has a separate web page, which includes links to source code and a pdf manual. This manual is a good place to start when learning a new package. If any are available, the vignettes can be even better. You can download vignettes (and manuals) directly from the package webpage on CRAN, or access those that are automatically installed along with the rest of the package with the `vignette` function.

```
vignette("Raster", package = "raster")
```

Leave out the first argument to get a list of available vignettes.

---

<sup>42</sup>Here, for example: <https://www.rdocumentation.org/>.

Additionally, a `library(help = dplyr)` call will produce a summary of the package (in this case `dplyr`) that includes a list of the objects in it.

```
library(help = dplyr)
```

And, some packages include a help file, which can be accessed with `package?stats`.

```
package?stats
```

Usually, the above information is all you need to know in order to install and load add-on packages in R. If you encounter problems, the additional details given below may be helpful.

Some users encounter problems when trying to install or load packages in Windows 7 due to directory permission restrictions<sup>43</sup>. Depending on the type of user account you are using, you may not have write permission for the library where R wants to install packages. The best way around this particular problem is to allow R to create a personal library in a directory for which you have write permission. To do this, simply select “Yes” when R asks if it should create a personal library the first time you install a package.

If you didn’t do this, you have a few options. You can uninstall and reinstall R, and select “Yes” when R asks if you want to create a personal library the first time you install a package. For the other options, we need to cover a bit of background first.

You can see the libraries that R is working with through the `.libPaths` function.

```
.libPaths()  
  
# [1] "/home/sasha/R/x86_64-pc-linux-gnu-library/3.4"  
# [2] "/usr/local/lib/R/site-library"  
# [3] "/usr/lib/R/site-library"  
# [4] "/usr/lib/R/library"
```

When you install a package without specifying the `lib` argument, R will use the first library returned by `.libPaths`. The paths of these libraries are stored in two environment variables: `R_LIBS_USER` and `R_LIBS_SITE`. To add a new library via an environment variable in Windows 7, you can add an argument to the “Target” box of the shortcut window that you get when you right-click on the R shortcut and select “Properties”. Just append something like `R_LIBS_USER = C:/Users/Sasha/Documents/R/R_lib` to the file path of the executable that is listed there<sup>44</sup>.

Alternatively, you can specify a library each time you use `install.packages` and `library` functions with the `lib` and `lib.loc` arguments, respectively.

The same name for an object may be used in two or more packages. If just using the name of the function alone, you can only access one of these at a time, and R will let you know which one is active<sup>45</sup>.

To “unload” functions, use the `detach` function (the syntax is a bit tricky for this task):

<sup>43</sup> The CRAN “R for Windows FAQ” (<http://cran.r-project.org/bin/windows/base/rw-FAQ.html>) has information on this topic under section 4.2.

<sup>44</sup> For Linux, see the `Renviron` file in `/etc/R`. For Mac OS X, see the help file for `.libPaths` for more information.

<sup>45</sup> You can work around this constraint by specifying the package name like this: `package.name::function.name`.

```
detach(package:dplyr)
```

Since restarting R accomplishes the same thing, this command is not usually needed.

Installing packages may be easier than finding the appropriate package for a particular task. The information provided in Section 5 should be useful for actually finding packages that provide the tools you need. In addition, it can be helpful to take a look at the list of “task views” on CRAN (<http://cran.r-project.org/web/views/>). These are groups of packages organized by topic. For example, the “Spatial” task view (<http://cran.r-project.org/web/views/Spatial.html>) lists all the packages that are useful for analysis of spatial data, and includes a discussion of the list.

## 7 The tidyverse packages

The R language is currently evolving. Base R (everything you have access to when you install R as you did for this workshop, i.e., not the add-on packages) changes only slightly and slowly. But Hadley Wickham and others are working to change how R is used, and how easy it is to carry out complex operations, through the development of new, and now popular, packages. One of the most important of these is dplyr, which is designed for data manipulation. The dplyr package and 11 others that share some approaches to programming are collectively referred to as the tidyverse, or tidyverse packages. See the tidyverse website (<http://tidyverse.org/>) for more details. The most important of these packages are:

- dplyr, for data manipulation
- magrittr, which defines the forward pipe operator
- ggplot2, which can be used to make sophisticated plots without much effort
- lubridate, for working with dates and times
- tibble, which defines a new type of data frame
- readr, for reading in data from text files
- readxl, for reading data from Excel files
- tidyr, for reshaping data frames to make “tidy” data

These packages are great. They make complicated operations simple, and use a consistent and relatively simple syntax. But there are some complications. First, the syntax of these packages is not completely consistent with base R. A good example is in the use of unquoted column names. Second, the functions are not always consistent with other functions, generally because of differences between tibbles (the new data frame) and the base R data frame.

Unfortunately, these issues make it a bit difficult to both teach and learn R. These packages should be covered, but they do not completely replace base packages. In this book, I cover both base and tidyverse approaches in several areas.

## 8 Data types and data objects in R

### 8.1 Data types

Understanding the different types of data in R can be a bit confusing. Use of the `class`, `mode`, or even `typeof` functions to check data types can be helpful (or, in some cases, confusing). The following discussion will attempt to accurately describe what I think you need to know<sup>46</sup>.

The simplest data structure in R is a vector, which is a simply an ordered collection of elements. All of the data objects we have worked with so far are vectors. There are several kinds of vectors, which differ only in the type of data they contain—we'll use them to demonstrate the different types of data in R. There are four common types of data that can be held in vectors<sup>47</sup>: numeric, integer, character, and logical.

Numeric data<sup>48</sup>

```
x <- 10.2
x

# [1] 10.2
```

Integer data

```
y <- 1:5
y

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

Character data

```
name <- "Johnny Appleseed"
name

# [1] "Johnny Appleseed"
```

Any time character data are entered directly into a console, you must surround individual elements with quotes. Otherwise, R will look for an object.

```
name <- Johnny

# Error in eval(expr, envir, enclos): object 'Johnny' not found
```

<sup>46</sup> For more information, check out the R Lanauge Manual available on CRAN [18].

<sup>47</sup> There are many more types of data in R (you can find a list here in the R Language Manual [18]); these four are just the most important to be aware of.

<sup>48</sup> All non-integer numeric data in R are “double-precision”, and are stored with a precision of 53 bits. See the help files for `as.double` and `.Machine` for more information.

Either single or double quotes can be used in R<sup>49</sup>. When character data are read into R from a file, quotes are not necessary<sup>50</sup>.

Logical data contain only three values: TRUE, FALSE, or NA (NA indicates a missing value—more on this later)<sup>51</sup>.

```
a <- TRUE  
a  
  
# [1] TRUE
```

Note that there are no quotes around logical values (quotes would make them character data). R will return logical data for any relational expression submitted to it.

```
4 < 2  
  
# [1] FALSE  
  
or  
  
b <- 4 < 2  
b  
  
# [1] FALSE
```

There are several functions that can be used to identify data type. In most cases, `class` or `mode` is the best bet. The class (i.e., the value returned by `class`) of an R object is meant to be “the official public view” [3, p 141] and it determines how generic functions operate on an object<sup>52</sup>. This is the same “class” that is referred to above in the discussion on generic functions.

```
class(x)  
  
# [1] "numeric"  
  
class(y)  
  
# [1] "integer"
```

<sup>49</sup> Double quotes are recommended, and are used in this book. Quotes can also be used to quote object names, including functions for operators (if you want to confuse yourself, check out "\*" or "["). Backticks (`) can also be used. In general, the three types of quotes behave the same, but for quoting object names, backticks may be the only ones that work in some cases. Check out the help file for quotes ?`" and experiment for more information.

<sup>50</sup> Unless spaces are present in individual elements, although even here quotes can be avoided by specifying a separator other than a space.

<sup>51</sup> R will also recognize T and F, but these are not reserved, and can therefore be overwritten by the user, and it is therefore good (although tedious) to avoid them. Autocomplete can help out. See the help file for TRUE for more information.

<sup>52</sup> More information, from Bill Venables: “‘mode’ is a mutually exclusive classification of objects according to their basic structure. . . ‘class’ is a property assigned to an object that determines how generic functions operate with it.” (<http://tolstoy.newcastle.edu.au/R/e4/help/08/04/8330.html>).

```

class(name)

# [1] "character"

class(a)

# [1] "logical"

```

Technically what these output mean is that the objects are: a numeric vector, an integer vector, a character vector, and a logical vector, respectively. For more complex objects, `class` doesn't return info on whether data are numeric, character, etc., but describes only the structure of the object. Unfortunately, this approach can be a bit confusing.

The `mode` and `class` functions return identical values for the three examples above, but this is not so for more complex objects, where `mode` gives a description of the way an object is stored.

```

mode(x)

# [1] "numeric"

mode(name)

# [1] "character"

mode(a)

# [1] "logical"

```

For many analyses, it is important to distinguish between quantitative (i.e., continuous) and categorical (i.e., discrete) variables. Categorical data (nominal or ordinal) are most easily stored as a different class of data called factors, which are like vectors with additional information. Internally, factors are stored as numeric data (as a check with `mode` will tell you<sup>53</sup>), but they are handled as categorical data in statistical analyses. R automatically recognizes non-numeric data as factors when data are read in<sup>54</sup>, but if numeric data are to be used as a factor (or if character data are generated within R and not read in), conversion to a factor must be done explicitly. In R, the function `factor` does this.

```

a <- c("female", "male", "male", "female", "male")
a

# [1] "female" "male"    "male"    "female" "male"

```

<sup>53</sup> This is a good example of the difference between `mode` and `class`.

<sup>54</sup> Automatic conversion of read-in data to factors can be suppressed by specifying `as.is = TRUE` or `stringsAsFactors = FALSE` when using `read.table` and related functions (such as `read.csv`). Alternatively, it can be suppressed globally by setting `stringsAsFactors` to `FALSE` with the `options` function. In older versions of R, factors took up much less space than character vectors, but apparently this is no longer the case. By the way, you can check storage space used by an object with `object.size`.

```

a <- factor(a)
a

# [1] female male   male   female male
# Levels: female male

```

Sometimes you may want to change the ordering or factor levels, so the order will make sense in your output, or the appropriate level will be used for comparisons, for example. We are going to use the `sample()` function to create a character vector with random values of height groups for this next example.

```

h <- sample(c("short", "medium", "tall"), 10, replace = TRUE)
h

# [1] "short"  "short"  "tall"   "medium" "tall"   "medium" "medium"
# [8] "short"  "tall"   "tall"

```

Let's make it a factor, and then check the default order of levels.

```

hf <- factor(h)
levels(hf)

# [1] "medium" "short"  "tall"

hf

# [1] short  short  tall   medium tall   medium medium short  tall
# [10] tall
# Levels: medium short tall

```

So R sorts the levels based on an alphabetical sorting of the original character values. To change that, use the `levels` argument.

```

hf <- factor(h, levels = c("short", "medium", "tall"))
levels(hf)

# [1] "short"  "medium" "tall"

```

In the example above, the `factor` function changed the the object `a` from a character vector to a factor. There is an entire group of functions in R called *coercion* functions that just change objects from one type to another<sup>55</sup>. For example, to make a vector character, use `as.character`.

```

x <- 1:10
x

```

---

<sup>55</sup> There is actually an `as.factor` function also, which is essentially a simplified version of `factor` (i.e., without all the additional arguments).

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

x <- as.character(x)
x

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

You can find all the related functions with `apropos`.

```
apropos("^as\\\.")
```

```
# [1] "as.array"                      "as.array.default"
# [3] "as.call"                         "as.character"
#...
#[123] "as.vector.factor"
```

## 8.2 Overview of R data structures

<sup>56</sup> Data in R are stored in data structures (also known as data objects)—these are the objects that you perform calculations on, plot data from, etc. The vectors we have worked with are one type of data structure, and others include matrices, arrays, data frames, and lists. I will demonstrate how to make these different data structures in a following section; the examples in this section are meant to simply give you an idea of their structure. Vectors may be the most important type of data structure in R. As mentioned above, a vector is simply an ordered collection of elements.

```
x <- 1:10
x

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

Matrices are similar to vectors, but have two dimensions.

```
m <- matrix(1:30, nrow = 3)
m

#      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
# [1,]     1     4     7    10    13    16    19    22    25    28
# [2,]     2     5     8    11    14    17    20    23    26    29
# [3,]     3     6     9    12    15    18    21    24    27    30
```

Arrays are similar to matrices, but can have more than two dimensions. The example below has three dimensions.

---

<sup>56</sup>This is a good time to install the tidyverse packages: `install.packages("tidyverse")`.

```

a <- array(1:27, dim = c(3, 3, 3))
a

# , , 1
#
# [,1] [,2] [,3]
# [1,]    1    4    7
# [2,]    2    5    8
# [3,]    3    6    9
#
# , , 2
#
# [,1] [,2] [,3]
# [1,]   10   13   16
# [2,]   11   14   17
# [3,]   12   15   18
#
# , , 3
#
# [,1] [,2] [,3]
# [1,]   19   22   25
# [2,]   20   23   26
# [3,]   21   24   27

```

One feature that is shared for vectors, matrices, and arrays is that they can only store one type of data at once, e.g., numeric, character, or logical, but only one. Technically speaking, these data structures can only contain elements of the same mode<sup>57</sup>. This is the same mode that can be checked with the `mode` function.

Data frames are two-dimensional objects similar to matrices. However, a data frame can contain columns (or variables, or vectors—call them what you like) with different modes. Data frames are similar to data sets used in other statistical programs—each column represents some variable, and each row usually represents an “observation” or “record”—and are very useful for data analysis in R.

```

dat <- read.csv("../data/willowy.csv")
dat

#   plot      trt   yr yield height
# 1    1 Control 2005  2.32   1.75
# 2    2 ManureL 2005  3.88   2.00
# 3    3      Urea 2005  1.88   1.58
# 4    4 ManureH 2005  3.63   1.89
# 5    5 ManureL 2005  4.76   2.04
# 6    6 SludgeManure 2005  3.07   1.87
# 7    7 ManureH 2005  8.97   1.94
# 8    8      Sludge 2005  3.51   1.77
# 9    9 SludgeManure 2005  5.68   2.19
# 10   10      Sludge 2005  3.17   1.85
# 11   11 Control 2005  4.19   2.08
# 12   12      Sludge 2005  4.51   2.13

```

---

<sup>57</sup> Data structures that can only contain elements of all the same mode are referred to as atomic—this is not important but may save you some confusion in the future.

```

# 13   13      Urea 2005  4.59  2.05
# 14   14      ManureH 2005  3.28  1.84
# 15   15      Sludge 2005  3.60  1.95
# 16   16      Urea 2005  4.45  1.92
# 17   17      Control 2005  3.87  1.88
# 18   18      Urea 2005  6.10  1.96
# 19   19      ManureL 2005  2.59  1.59
# 20   20 SludgeManure 2005  3.74  2.03
# 21   21      ManureH 2005  2.99  1.77
# 22   22      ManureL 2005  2.11  1.63
# 23   23      Control 2005  2.59  1.69
# 24   24 SludgeManure 2005  1.68  1.53

```

Tibbles are a new type of data frame that are defined in the tibble package. The most important differences relate to the print methods (what is called when you enter the name of the object to view it) and in indexing.

```

library(tidyverse)

# -- Attaching packages ----- tidyverse 1.2.1 --
# ggplot2 2.2.1      purrrr  0.2.4
# tibble  1.4.2      dplyr   0.7.4
# tidyrr  0.8.0      stringr 1.2.0
# readr   1.1.1      forcats 0.3.0

# -- Conflicts ----- tidyverse_conflicts() --
# x tidyr::extract() masks magrittr::extract()
# x dplyr::filter()  masks stats::filter()
# x dplyr::lag()     masks stats::lag()
# x purrrr::set_names() masks magrittr::set_names()

datt <- read_csv("../data/willowy.csv")

# Parsed with column specification:
# cols(
#   plot = col_integer(),
#   trt = col_character(),
#   yr = col_integer(),
#   yield = col_double(),
#   height = col_double()
# )

datt

# # A tibble: 24 x 5
#       plot trt          yr  yield height
#   <int> <chr>      <int> <dbl>  <dbl>
# 1      1 Control    2005  2.32  1.75
# 2      2 ManureL   2005  3.88  2.00
# 3      3 Urea      2005  1.88  1.58

```

```

# 4     4 ManureH      2005 3.63  1.89
# 5     5 ManureL      2005 4.76  2.04
# 6     6 SludgeManure 2005 3.07  1.87
# 7     7 ManureH      2005 8.97  1.94
# 8     8 Sludge        2005 3.51  1.77
# 9     9 SludgeManure 2005 5.68  2.19
# 10    10 Sludge       2005 3.17  1.85
# # ... with 14 more rows

```

One advantage of tibbles is that the type of data in each column is shown at the top of the print summary.

Lists are similar to vectors, in that they are an ordered collection of elements, but with lists, the elements can be other data objects (the elements can even be other lists). It might help to think of lists as composite objects—(generally) formed by combining other objects. Lists are important in the output from many different functions (the R function for linear models, `lm`, returns its output in a list, for example).

```
L <- list("This is the first element in our list", 1:10, dat)
L
```

```

# [[1]]
# [1] "This is the first element in our list"
#
# [[2]]
# [1] 1 2 3 4 5 6 7 8 9 10
#
# [[3]]
#   plot          trt   yr yield height
# 1 1     Control 2005 2.32  1.75
# 2 2     ManureL 2005 3.88  2.00
# 3 3       Urea 2005 1.88  1.58
# 4 4     ManureH 2005 3.63  1.89
# 5 5     ManureL 2005 4.76  2.04
# 6 6 SludgeManure 2005 3.07  1.87
# 7 7     ManureH 2005 8.97  1.94
# 8 8       Sludge 2005 3.51  1.77
# 9 9 SludgeManure 2005 5.68  2.19
# 10 10      Sludge 2005 3.17  1.85
# 11 11     Control 2005 4.19  2.08
# 12 12      Sludge 2005 4.51  2.13
# 13 13       Urea 2005 4.59  2.05
# 14 14     ManureH 2005 3.28  1.84
# 15 15      Sludge 2005 3.60  1.95
# 16 16       Urea 2005 4.45  1.92
# 17 17     Control 2005 3.87  1.88
# 18 18       Urea 2005 6.10  1.96
# 19 19     ManureL 2005 2.59  1.59
# 20 20 SludgeManure 2005 3.74  2.03
# 21 21     ManureH 2005 2.99  1.77
# 22 22     ManureL 2005 2.11  1.63
# 23 23     Control 2005 2.59  1.69
# 24 24 SludgeManure 2005 1.68  1.53

```

Note that a particular data structure need not contain data to exist. This can be useful when it is necessary to set up an object for holding some data later on.

```
x <- NULL
```

### 8.3 Vectors

Vectors are very useful in R, and there are several ways to create them. Where elements are spaced by exactly 1, just separate the values of the first and last elements with a colon.

```
1:5
```

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

The colon operator can also create vectors with a spacing different from 1, if the first and last elements make sense. But the function `seq` (for sequence) is more flexible. Its typical arguments are `from`, `to`, and `by` (or, instead of `by`, you can specify `length.out`).

```
seq(-10, 10, 2)
```

```
# [1] -10 -8 -6 -4 -2 0 2 4 6 8 10
```

The `by` argument does not need to be an integer. When all the elements in a vector are identical, use the `rep` function (for repeat).

```
rep(4, 5)
```

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

The `rep` function is actually much more flexible than the above example suggests. Here are two other examples.

```
x <- 1:3  
rep(x, 4)
```

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

```
rep(x, each = 4)
```

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

```
rep(x, times = c(3, 1, 3))
```

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

For other cases, use `c` (for concatenate or combine)—this is the most flexible approach for creating vectors.

```
c(2, 1, 5, 100, 2)
```

```
# [1] 2 1 5 100 2
```

Note that you can name the elements within a vector.

```
c(a = 2, b = 1, c = 5, d = 100, e = 2)
```

#	a	b	c	d	e
#	2	1	5	100	2

Any of these expressions could be assigned to a symbolic variable, using the assignment operator.

```
v <- c(2, 1, 5, 100, 2)
```

```
# [1] 2 1 5 100 2
```

If you need to create a vector of a certain length and only later add values, there are functions named for each type of vector—they create vectors with some default value.

```
x <- numeric(50)
```

Variable names can be any combination of letters, numbers, and the symbols . and \_, but, they cannot start with a number or with <sup>58</sup>.

```
a_vector_with.a.long.name.100 <- seq(1, 3, 0.1)  
a_vector_with.a.long.name.100
```

```
# [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2 2.3 2.4 2.5  
# [17] 2.6 2.7 2.8 2.9 3.0
```

Function names are not prevented from being used for variables, although you may end up confusing yourself. A few variables names are reserved. If you are ever unsure of whether a name is available, it is easy enough to check—just type the name into the console and hit **Enter**.

The `c` function is very useful for setting up arguments for other functions, as will be shown later. As with all R functions, both variable names and function names can be used in function calls as arguments.

<sup>58</sup> You can get around this limitation by using backtick quotes around the name, although I have never encountered a need to do so.

```

x <- rep(1, 3)
y <- 4:10
z <- c(x, y)
z

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

```

Although R prints the contents of individual vectors with a horizontal orientation, R does not have “columns vectors” and “row vectors”, and vectors do not have a fixed orientation. This makes use of vectors in R very flexible.

Vectors do not need to contain numbers, but can contain data with any of the modes mentioned earlier (numeric, logical, character, and complex) as long as all the data in a vector are of the same mode.

Logical vectors are very useful in R for indexing (or subscripting) data, i.e., for isolating some part of an object that meets a certain criterion<sup>59</sup>. For relational expressions, the shorter vector is repeated as many times as necessary to carry out the requested comparison for each element in the longer vector (this repeat rule is discussed more below).

```

x <- 1:10
x>5

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

```

Also, when logical vectors are used in arithmetic, they are changed (coerced in R terms) into a numeric vector of binary elements: 1 or 0. Continuing with the above example:

```

a <- x>5
a*20

# [1] 0 0 0 0 0 20 20 20 20 20

```

So, for example, if you wanted to know the proportion of values in `x` that were greater than 3, the following command does the trick.

```

mean(x>3)

# [1] 0.7

```

## 8.4 Matrices, arrays, and lists

Arrays are multi-dimensional collections of elements and matrices are simply two-dimensional arrays. R has several operators and functions for carrying out operations on arrays, and matrices in particular (e.g., matrix multiplication). Many data analysis and plotting tasks can be carried out without using

---

<sup>59</sup> See Section 14.1.

arrays or matrices, but these data structures are useful for some tasks and are important internally for many functions.

To generate a matrix, the `matrix` function can be used<sup>60</sup>.

```
m <- matrix(1:15, nrow = 5)
m

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

Note that the filling order is by column by default (i.e., each column is filled before moving onto the next one). The “unpacking” order is the same, e.g., if you applied `as.vector` to convert the data to a vector, as is the order of calculations when carrying out operations that require recycling of a smaller vector<sup>61</sup>.

```
as.vector(m)

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

If you want to change the filling order, you can use the `byrow` argument<sup>62</sup>:

```
m <- matrix(1:15, nrow = 5, byrow = TRUE)
m

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

A similar function is available for higher-order arrays, called `array`. Here is an example with a three-dimensional array:

```
a <- array(1:30, dim = c(5, 3, 2))
a

# , , 1
#
#      [,1] [,2] [,3]
```

<sup>60</sup> Simply specifying dimensions for a vector can turn it into a matrix, or, more generally, an array. Try this: `m <- 1:15; dim(m) <- c(5, 3)`. And you can also create a matrix with `array`.

<sup>61</sup> Another way to convert an object to a vector is with the `c` function. Unlike `as.vector`, this approach retains element names.

<sup>62</sup> But this doesn’t change the unpacking order.

```

# [1,]    1    6   11
# [2,]    2    7   12
# [3,]    3    8   13
# [4,]    4    9   14
# [5,]    5   10   15
#
#
# , , 2
#
#      [,1] [,2] [,3]
# [1,]    16   21   26
# [2,]    17   22   27
# [3,]    18   23   28
# [4,]    19   24   29
# [5,]    20   25   30

```

Lists are simple to create in R—just combine the individual elements with the `list` function, or convert an existing object with `as.list`.

```

1 <- list(x, name, m)
1

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

```

```

x

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

as.list(x)

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

```

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

# 9 Data frames, data import, and data export

## 9.1 Your working directory

Try reading in a data file with this command:

```
pop <- read.csv("../data/us_pop.csv")
```

If you see an error message like this one:

```
#Error in file(file, "rt") : cannot open the connection  
#In addition: Warning message:  
#In file(file, "rt") :  
#  cannot open file 'us_pop.csv': No such file or directory
```

it means that the file is not in R's working directory, which is the directory (folder) where R looks for files, and where it writes files. This is a very common problem. To sort it out, first, check your working directory:

```
getwd()  
  
# [1] "/home/sasha/Dropbox/R course 2019/book etc/Rbook"
```

Is this the correct one? You can see what files are inside any directory with `list.files`. The following command will show the contents of your working directory.

```
list.files()
```

or

```
list.files("../data")
```

or

```
list.files("../data")
```

```
list.files()  
  
# [1] "ave_wind_us.csv"          "biohydrogen_bad.csv"  
# [3] "biohydrogen.csv"           "biohydrogen.ods"  
# [5] "biohydrogen.xlsx"          "bottle.csv"  
# ...
```

The “..” in a file path means “go up one level”. So in the last example (`list.files("../data")`), the working directory is a sub-directory at the same level as “data”. If you follow the approach I recommend for file organization, this is the code you should use for reading in files.

If you keep your input data files in your working directory, which should always contain your scripts, you would see them here. Try to find this input file in the "data files" directory, and copy it into your working directory. The rerun the `list.files()` command. Does it show up now?

Usually, it makes sense to separate your scripts from your data files. If you have kept the structure I created for the files you downloaded for this course, you will have a "data files" subdirectory with all the data files. Our `read.csv` command above implies that the working directory and But except for the simplest

If you are using the wrong directory, change it. The command below works in any version of R.

```
setwd("/home/sasha/R2018/scripts")
```

And in RStudio, you can use the **Session** menu (**Session**, then **Set working directory**, then To source file location). Perhaps the best approach is to simply use RStudio projects (Section 4.3).

Windows users: note that you need to use forward slashes or double backward slashes when you specify the file path manually—not single backward slashes. This is because the backward slash is an escape character in R, and is used to indicate that the following character should not be interpreted as a metacharacter. To keep from even thinking about this, use the **Session** menu as described above.<sup>63</sup>.

## 9.2 Reading text files

As described above, a data frame is a type of data structure in R with rows and columns, where different columns can contain data with different modes. A data frame is probably the most common data structure that you will use for storing what you might call data sets. The easiest way to create a data frame is to read in data from a file—this is done using the function `read.csv` or the `readr` package equivalent `read_csv`. These and related function work with ASCII text files.

R is very flexible in how it reads in data from text files. Typically, organization will be as follows (from the file `us_pop.csv`<sup>64</sup>).

```
year, pop
1790, 3929214
1800, 5308483
1810, 7239881
1820, 9638453
1830, 12866020
....
```

You should get used to using a text editor (e.g., Notepad or Notepad++ in Windows,TextEdit in Mac OS X) to view data files. If you opened this csv file in Excel, you would not be able to see the separators. If you combine that without being able to see the extension, there is a high probability of resulting confusion and frustration!

The data in this file can be read in with the following command.

<sup>63</sup> And you can set a default working directory if you want. See here: <https://support.rstudio.com/hc/en-us/articles/200711843-Working-Directories-and-Workspaces>.

<sup>64</sup> Did you remember to change settings as described at the top of Section 4.3? If not, do it now or during the next break!

```

pop <- read.csv("../data/us_pop.csv")
pop

#   year      pop
# 1 1790  3929214
# 2 1800  5308483
# 3 1810  7239881
# 4 1820  9638453
# 5 1830 12866020
# 6 1840 17069453
# 7 1850 23191876
# 8 1860 31443321
# 9 1870 38558371
# 10 1880 50189209
# 11 1890 62979766
# 12 1900 76212168
# 13 1910 92228496
# 14 1920 106021537
# 15 1930 123202624
# 16 1940 132164569
# 17 1950 151325798
# 18 1960 179323175
# 19 1970 203302031
# 20 1980 226542199
# 21 1990 248709873
# 22 2000 281421906

```

Using commas as separators circumvents the problem of spaces in character data and allows for easy viewing of data in any text editor (or a spreadsheet program). The `read.csv` function generally makes for less typing than the more flexible counterpart `read.table`, which is discussed below. For these reasons, I recommend using comma separated files whenever possible, and `read.csv` is the primary approach used in this book.

There is a `readr` package version of this function that produces a tibble instead of a data frame.

```

pop <- read_csv("../data/us_pop.csv")

# Parsed with column specification:
# cols(
#   year = col_integer(),
#   pop = col_integer()
# )

pop

# # A tibble: 22 x 2
#   year      pop
#   <int>    <int>
# 1 1790  3929214
# 2 1800  5308483
# 3 1810  7239881

```

```
# 4 1820 9638453
# 5 1830 12866020
# 6 1840 17069453
# 7 1850 23191876
# 8 1860 31443321
# 9 1870 38558371
# 10 1880 50189209
# # ... with 12 more rows
```

If you encounter files with a semicolon separator and a decimal comma, use the `read.csv2` or `read_csv2` function.

For more flexibility, use the `read.table` function, which is the function that is called internally by `read.csv` and related functions. This file has spaces as the separator, as you can see by opening the file in a text editor.

```
flow <- read.table("../data/flow_2006_summary.txt", header = TRUE)
head(flow)
```

```
# site.id month discharge
# 1 1509000 Jan 37237.615
# 2 1509000 Feb 24522.010
# 3 1509000 Mar 27158.420
# 4 1509000 Apr 16505.757
# 5 1509000 May 9830.955
# 6 1509000 Jun 27116.135
```

The `head` function displays the first several lines of a data frame, or the first several elements of other objects. As with the UNIX utilities, there is also a `tail` function.

With `read.table` the `header` argument is `FALSE` by default, so you must specify `header = TRUE`, or else R will interpret the row of labels (header row) as data. If you do not specify a field separator (the `sep` argument) (as in the call above), R assumes that any spaces or tabs separate the data in your text file. In this case, the number of whitespace characters separating your columns does not matter. However, any character data that contain spaces must be surrounded by quotes (otherwise, R interprets the data on either side of the whitespace as different elements).

Alternately, other separators can be specified. If you specify a separator (say `sep = "\t"` for tabs<sup>65</sup> or `sep = ", "` for commas, as in `read.csv`) two consecutive separators will be interpreted as a missing value. Conversely, with the default options, you need to explicitly identify missing values in your data file with `NA` (or any other character, as long as you tell R what it is with the `na.strings` argument).

For some field separators, there are alternate functions that can be used with the default arguments. Most of the remaining examples in this book use one: `read.csv`, which is identical to `read.table`, except for having default arguments `sep = ", "` and `header = TRUE`, along with some others. Check out the help file for `read.table` (including the *See Also* section) for more functions.

R doesn't care what the name of your file is or what its extension is, as long as it is an ASCII text file. You can include comments at the end of rows in your data file—just precede them with a `#`. And R will recognize `na`, `nan`, `inf`, and `-inf` in input files.

---

<sup>65</sup> See the help file for `regex` for more information on regular expressions.

Probably the easiest approach to handling missing values is to indicate their presence with `na` in the text file. As mentioned above, R will automatically recognize these as missing values. Since the file `flow_2006_summary.txt` uses `na` for missing values, they should have been read in properly.

```
which(is.na(flow$discharge))

# [1] 21

flow

#   site.id month discharge
# 1 1509000 Jan 37237.615
# 2 1509000 Feb 24522.010
# 3 1509000 Mar 27158.420
# 4 1509000 Apr 16505.757
# 5 1509000 May 9830.955
# 6 1509000 Jun 27116.135
# 7 1509000 Jul 26460.672
# 8 1509000 Aug 12425.920
# 9 1509000 Sep 7459.027
# 10 1509000 Oct 23286.085
# 11 1509000 Nov 29718.360
# 12 1509000 Dec 14157.208
# 13 4232730 Jan 60388.111
# 14 4232730 Feb 39808.994
# 15 4232730 Mar 7917.205
# 16 4232730 Apr 13785.879
# 17 4232730 May 10143.942
# 18 4232730 Jun 31553.909
# 19 4232730 Jul 24587.140
# 20 4232730 Aug 11300.953
# 21 4232730 Sep      NA
# 22 4232730 Oct 26051.044
# 23 4232730 Nov 49824.330
# 24 4232730 Dec 36907.146
```

If this all seems confusing, remember that you can avoid a lot of frustration by asking a few things about any new file:

1. Is a header present (`header` argument)?
2. What is the field separator (`sep` argument)?
3. What symbol is used for decimals (`dec` argument)?
4. What symbol is used for missing values (`na.strings` argument)?

Let's work on another example. The file `Ni_ec50s.txt` contains tab-separated data. It looks like this:

```
soil ec50.ni ph.soil oc ph.sol c.ni c.na c.mg c.k c.ca
"Houthalen" 54.5 3.56 1.7 3.74 36.4 15.3 9.9 21.1 32.8
```

```
"Zegveld" 1928.2 4.11 33.1 3.88 72.9 56.3 148 88.3 980
```

```
...
```

A header row is present, but row numbers are not—this is a typical and logical approach in my opinion. With tabs or spaces as the separator, we can use the default value of `sep`. But we have to indicate that a header is present.

```
ni <- read.table("../data/Ni_ec50s.txt", header = TRUE)
ni

#       soil ec50.ni ph.soil   oc ph.sol c.ni c.na c.mg   c.k
# 1 Houthalen    54.5   3.56  1.7   3.74 36.4 15.3   9.9  21.1
# 2 Zegveld     1928.2   4.11 33.1   3.88 72.9 56.3 148.0  88.3
# 3 Rhydtalog    533.5   4.20 12.5   4.19 42.7 36.1 110.0  46.8
# 4 Jyndevad     73.9   4.48  1.3   4.55 21.2 14.6  26.3  72.8
# 5 Kovlinge II   202.5   5.13  2.5   5.44 27.5 20.1  86.9  47.8
# 6 Borris        193.0   5.58  1.3   5.82 28.8 28.9 121.0 207.0
# 7 Woburn        707.6   6.07  4.3   5.70 24.2 23.2 219.0  17.6
# 8 Ter Munck     248.2   6.74  1.1   6.83 17.5 45.3 106.0 249.0
#
#       c.ca
# 1     32.8
# 2    980.0
# 3    551.0
# 4    342.0
# 5    469.0
# 6   1060.0
# 7    907.0
# 8  1110.0
```

It would also be OK to specify the separator.

```
ni <- read.table("../data/Ni_ec50s.txt", header = TRUE, sep = "\t")
ni

#       soil ec50.ni ph.soil   oc ph.sol c.ni c.na c.mg   c.k
# 1 Houthalen    54.5   3.56  1.7   3.74 36.4 15.3   9.9  21.1
# 2 Zegveld     1928.2   4.11 33.1   3.88 72.9 56.3 148.0  88.3
# 3 Rhydtalog    533.5   4.20 12.5   4.19 42.7 36.1 110.0  46.8
# 4 Jyndevad     73.9   4.48  1.3   4.55 21.2 14.6  26.3  72.8
# 5 Kovlinge II   202.5   5.13  2.5   5.44 27.5 20.1  86.9  47.8
# 6 Borris        193.0   5.58  1.3   5.82 28.8 28.9 121.0 207.0
# 7 Woburn        707.6   6.07  4.3   5.70 24.2 23.2 219.0  17.6
# 8 Ter Munck     248.2   6.74  1.1   6.83 17.5 45.3 106.0 249.0
#
#       c.ca
# 1     32.8
# 2    980.0
# 3    551.0
# 4    342.0
# 5    469.0
# 6   1060.0
# 7    907.0
# 8  1110.0
```

For another example, let's take a look at the contents of the file us\_pop.txt. It looks something like this:

```
year pop
1790 3929214
1800 5308483
1810 7239881
...
```

A header row is present, but row numbers are not. The separator is a variable number of spaces (not a very good approach). But, it should be easy to read in with the default `sep` value:

```
pop <- read.table("../data/us_pop.txt", header = TRUE)
head(pop)
```

```
#   year      pop
# 1 1790 3929214
# 2 1800 5308483
# 3 1810 7239881
# 4 1820 9638453
# 5 1830 12866020
# 6 1840 17069453
```

```
tail(pop)
```

```
#   year      pop
# 17 1950 151325798
# 18 1960 179323175
# 19 1970 203302031
# 20 1980 226542199
# 21 1990 248709873
# 22 2000 281421906
```

```
dim(pop)
```

```
# [1] 22  2
```

Note that we did not need to specify a separator, since one or more whitespace characters (spaces, in this case) are interpreted as a separator<sup>66</sup>. For files with whitespace separators, unless the file had unquoted character strings that contain whitespace characters (e.g., spaces) or missing values are actually missing (i.e., not represented by NA or some other string), there is no need to specify the separator with the `sep` argument.

So far, I've only described reading in data from text files. But how about other types of files? In many or even most cases, it makes the most sense to put your data into a text file for getting it into R. This can be done in various ways. Data downloaded from the internet are often in text files to begin with. Data can be entered directly into a text file using a text editor.

<sup>66</sup> In fact, if we had specified `sep = " "` R would have returned an error, since the number of spaces between entries is not constant in this file.

In some cases, it makes more sense to stick with the original file type. Fortunately, R has the capability to handle many different formats<sup>67</sup>. For example, the `foreign` package provides functions for reading files in about ten other formats (including SAS, Stata, and Minitab).

### 9.3 Reading spreadsheet files

Spreadsheets are great for entering, organizing, and storing data. For data that are in a spreadsheet program such as Microsoft Excel you have at least two options for getting them into R.

- Export worksheets to text files
- Use an add-on package to read data directly from the spreadsheet

Export used to be simpler. Data can be saved (or perhaps exported) directly from spreadsheet programs as a text file, e.g. as “Formatted Text (Space delimited)” (\*.prn) in Excel, or, better, as comma-separated values (\*.csv)<sup>68</sup>. Or, data can be copied and pasted into a text editor and saved as a text file—this creates a tab-delimited file<sup>69</sup>. One disadvantage is that you must export the data each time a change is made to the spreadsheet file. Reading directly from the Excel file is easier in this regard.

For Excel, there are at least four packages that provide functions for reading data directly from Excel files: `gdata`, `XLConnect`, `xlsx`, and `readxl`. The easiest is `readxl`. It is so easy that I recommend it over exporting files to a text format, as long as your Excel file will be updated. If you have a fixed version that will not change, just export it.

```
library(readxl)
h1 <- read_excel("../data/biohydrogen.xlsx", sheet = 2)
head(h1)

# # A tibble: 6 x 5
#   reactor date      time    vol conc.h2
#   <chr>   <chr>     <chr>  <dbl>   <dbl>
# 1 G171    9/18/2006 11:12     0     NA
# 2 G171    9/18/2006 14:00     0     31.2
# 3 G171    9/19/2006 9:26    11.4    35.2
# 4 G171    9/19/2006 12:51     0     NA
# 5 G171    9/19/2006 16:00     0     NA
# 6 G171    9/19/2006 22:52     0     NA
```

### 9.4 Creating data frames manually

Data frames can be made manually using the `data.frame` function:

<sup>67</sup> For reading data directly from databases, for example, check out the `RODBC` package.

<sup>68</sup> If you use Linux, check out the fantastic spreadsheet program Gnumeric. With it you can export worksheets by selecting “Export” under the “Data” menu.

<sup>69</sup> And some users may find the `file = "clipboard"` option handy for quickly reading in data. Just copy your data to the system clipboard first.

```

group <- c("A", "B", "C", "D")
mass <- rnorm(4, mean = 50)
dat <- data.frame(id = group, mean = mass)
dat

#   id      mean
# 1 A 49.67842
# 2 B 51.00124
# 3 C 52.70437
# 4 D 50.83331

```

While this approach is not an efficient way to enter data that could be read in directly, it can be handy for some applications.

As with other functions, column names (more generally, element names within objects) are specified using `=`. It is also possible to set or extract column names for an existing data frame using the function `names`<sup>70</sup>. This function can be used in the same way for other data structures as well.

```

names(dat)

# [1] "id"    "mean"

names(dat) <- c("date", "mass")
dat

#   date      mass
# 1 A 49.67842
# 2 B 51.00124
# 3 C 52.70437
# 4 D 50.83331

```

Row names (1 through 4 above) can be specified in a `data.frame` call with the `row.names` argument.

```

dat <- data.frame(id = group, mean = mass, row.names = c("a", "b", "c", "d"))
dat

#   id      mean
# a  A 49.67842
# b  B 51.00124
# c  C 52.70437
# d  D 50.83331

```

Row names can be useful for indexing data, which will be covered later. Row names for an existing data frame can also be extracted or changed with the `rownames` function (not to be confused with the `row.names` argument).

---

<sup>70</sup> R actually uses a different function for extracting names than it does to change names. But it looks the same to users. Changing names requires what is called a replacement function, where the function is called to the left of the assignment operator.

## 9.5 Working with data frames

So what do you do with data in R once it is in a data frame? Commonly, the data in a data frame will be used in some type of analysis or plotting procedure. But a first step should be some form of data checking. This can be done using summaries and simple plots. Although neither approach is guaranteed to catch all errors, they can at least help us spot gross errors, such as a decimal place in the wrong location or missing missing observations. The generic `summary` function is one option. To demonstrate, let's read in some data on hydrogen production from bacteria [9].

```
h2 <- read.csv("../data/biohydrogen.csv")
summary(h2)

#      reactor        date       time       vol
# G171     : 9  9/18/2006:30  11:12  :15  Min.   : 0.000
# G172     : 9  9/19/2006:60  12:40  :15  1st Qu.: 0.000
# G173     : 9  9/20/2006:30  12:51  :15 Median : 3.650
# G174     : 9  9/21/2006:15  13:41  :15 Mean    : 7.719
# G175     : 9                  14:00  :15  3rd Qu.:16.575
# G176     : 9                  16:00  :15 Max.    :26.600
# (Other):81                (Other):45 NA's    :1
#      conc.h2
# Min.   : 0.00
# 1st Qu.: 9.90
# Median :18.96
# Mean   :17.14
# 3rd Qu.:25.14
# Max.   :35.22
# NA's   :68
```

What `summary` lacks is (consistent) reporting of column class, which can be very helpful for identifying errors (e.g., when what should be a numeric column is actually character). The `dfsumm` function that you can find in `dfsumm.R` is an alternative.

```
source("../functions/dfsumm.R")

dfsumm(h2)

#
# 135 rows and 5 columns
# 135 unique rows
#      reactor        date       time       vol conc.h2
# Class          factor      factor factor numeric numeric
# Minimum        G171 9/18/2006  11:12       0       0
# Maximum        G185 9/21/2006  9:26    26.6   35.2
# Mean           G178 9/19/2006  14:00    7.72   17.1
# Unique (excl. NA) 15        4       9     71    59
# Missing values 0         0       0     1    68
# Sorted          TRUE     FALSE  FALSE FALSE FALSE
```

The `summary` function or something like it should probably be your first stop after reading in your data, but before analyzing it—it provides an easy way to check for obvious errors and to see the range of values.

If you use the `readr` package functions instead of the base ones, you will create tibbles, and the print summary will at least provide column types.

```
library(readr)

h2t <- read_csv("../data/biohydrogen.csv")

# Parsed with column specification:
# cols(
#   reactor = col_character(),
#   date = col_character(),
#   time = col_time(format = ""),
#   vol = col_double(),
#   conc.h2 = col_double()
# )

h2t

# # A tibble: 135 x 5
#   reactor date     time     vol conc.h2
#   <chr>    <chr>    <time>  <dbl>   <dbl>
# 1 G171    9/18/2006 11:12     0      NA
# 2 G171    9/18/2006 14:00     0     31.2
# 3 G171    9/19/2006 09:26   11.4     35.2
# 4 G171    9/19/2006 12:51     0      NA
# 5 G171    9/19/2006 16:00     0      NA
# 6 G171    9/19/2006 22:52     0      NA
# 7 G171    9/20/2006 08:52     0      NA
# 8 G171    9/20/2006 13:41   2.00    30.9
# 9 G171    9/21/2006 12:40     0     30.3
# 10 G172   9/18/2006 11:12    0      NA
# # ... with 125 more rows
```

But it does not have a good approach to handling wide data frame—it simply doesn't display all the columns.

The functions `unique()`, `sort()`, and `range()` can also be useful for checking data.

```
unique(h2$reactor)

# [1] G171 G172 G173 G174 G175 G176 G177 G178 G179 G180 G181 G182 G183
# [14] G184 G185
# 15 Levels: G171 G172 G173 G174 G175 G176 G177 G178 G179 G180 ... G185

unique(h2$date)

# [1] 9/18/2006 9/19/2006 9/20/2006 9/21/2006
# Levels: 9/18/2006 9/19/2006 9/20/2006 9/21/2006
```

But what should you look for when checking for data errors? To some degree the answer depends on the nature of the data, but some general problems are listed below.

- Numeric columns stored as character data or factors
- Impossible values
- Typos or inconsistent letter case in character data
- Typos in numeric data (decimal place errors)
- Numeric missing value codes (999, -999, 0)
- Incorrect column names
- Merged columns

For errors in numeric values, plots are also very useful—We'll cover simple plots for checking data later on.

In order to use data frame data for a plot or analysis, it is usually necessary to be able to select and identify specific columns (also referred to as vectors or variables) within data frames. There are three ways to specify a given column of data from within a data frame. The first is to use the \$ notation.

```
h2 <- read.csv("../data/biohydrogen.csv")
```

To see what the column names are, we can use `names`.

```
names(h2)
```

```
# [1] "reactor" "date"     "time"      "vol"       "conc.h2"
```

The \$ notation just uses a \$ between the data frame name and column name to specify a particular column. Say we want to look at the `vol` column, which contains the volume of biogas (a mixture of H<sub>2</sub> and CO<sub>2</sub> in this case) produced by a particular reactor<sup>71</sup>.

```
h2$vol
```

```
# [1]  0.00  0.00 11.35  0.00  0.00  0.00  0.00  2.00  0.00  0.00
# [11] 0.00 19.50 14.25  9.10 24.20 17.50  4.00  4.00  0.00  0.00
# [21] 21.40 16.20  9.90 25.50 17.40  4.00  0.00  0.00  0.00 22.50
# [31] 17.00 10.50 26.60 17.10  2.00  0.00  0.00  0.00 21.20 15.30
# [41] 10.40 23.60 16.80  7.00  4.00  0.00  0.00 20.40 12.70  9.30
# [51] 21.20 20.70  2.60    NA  0.00  0.00  0.00  4.20  0.00  0.00
# [61] 0.00  1.85  1.70  0.00  0.00  0.00  3.60  0.00  0.00  0.00
# [71] 0.00  2.50  0.00  0.00  0.00  3.70  0.00  0.00  0.00  2.00
# [81] 0.80  0.00  0.00 17.80 14.40 14.00 24.60  5.10  2.60  5.50
# [91] 0.00  0.00 22.75 20.00 16.50 14.90  7.00  4.00  3.00  0.00
```

<sup>71</sup> With the \$ notation, you don't even need to specify the complete name of the column you want—just enough to distinguish it from other columns is sufficient, so `h2$v` would also work here. However, I would not recommend doing this, since you may add columns to your data frame in future versions of your code.

```
# [101] 0.00 21.90 19.20 16.60 17.50 6.30 4.40 1.50 0.00 0.00
# [111] 25.20 19.80 17.30 17.30 6.60 2.70 2.80 0.00 0.00 24.60
# [121] 21.20 18.80 15.10 9.00 2.40 0.00 0.00 0.00 19.60 18.30
# [131] 16.30 23.00 5.00 2.10 6.20
```

Although it is handy to think of data frame columns as having a vertical orientation, this orientation is not present when they are printed individually—instead, elements are printed from left to right, and then top to bottom.

The expression `h2$vol` could be used just as you would any other vector. For example:

```
mean(h2$vol)
```

```
# [1] NA
```

R can't calculate the mean because of a single NA value. Let's remove it first using the `na.omit` function (more on this below)<sup>72</sup>:

```
mean(na.omit(h2$vol))
```

```
# [1] 7.719403
```

The second option for working with individual variables (vectors or columns) within a data frame is to use the commands `attach` and `detach`. Both of these functions take a data frame as an argument: attaching a data frame puts a copy of all the variables within that data frame in R's search path, and they can be called by using their names alone without the \$ notation.

```
attach(h2)
```

```
vol
```

```
# [1] 0.00 0.00 11.35 0.00 0.00 0.00 0.00 2.00 0.00 0.00
# [11] 0.00 19.50 14.25 9.10 24.20 17.50 4.00 4.00 0.00 0.00
# [21] 21.40 16.20 9.90 25.50 17.40 4.00 0.00 0.00 0.00 22.50
# [31] 17.00 10.50 26.60 17.10 2.00 0.00 0.00 0.00 21.20 15.30
# [41] 10.40 23.60 16.80 7.00 4.00 0.00 0.00 20.40 12.70 9.30
# [51] 21.20 20.70 2.60 NA 0.00 0.00 0.00 4.20 0.00 0.00
# [61] 0.00 1.85 1.70 0.00 0.00 0.00 3.60 0.00 0.00 0.00
# [71] 0.00 2.50 0.00 0.00 0.00 3.70 0.00 0.00 0.00 2.00
# [81] 0.80 0.00 0.00 17.80 14.40 14.00 24.60 5.10 2.60 5.50
# [91] 0.00 0.00 22.75 20.00 16.50 14.90 7.00 4.00 3.00 0.00
# [101] 0.00 21.90 19.20 16.60 17.50 6.30 4.40 1.50 0.00 0.00
# [111] 25.20 19.80 17.30 17.30 6.60 2.70 2.80 0.00 0.00 24.60
# [121] 21.20 18.80 15.10 9.00 2.40 0.00 0.00 0.00 19.60 18.30
# [131] 16.30 23.00 5.00 2.10 6.20
```

```
detach(h2)
```

---

<sup>72</sup> Or, we could specify `na.rm = TRUE`. See the help file for `mean` for more information.

When you are done using the individual variables, it is good practice to **detach** your data frame. Once the data frame is detached, R will no longer know what you mean when you specify the name of a column alone:

```
detach(h2)
```

```
#vol
```

If you modify a variable that is part of an **attached** data frame, the data within the data frame remain unchanged; you are actually working with a copy of the data frame. Usually, it is best to avoid using **attach** and **detach**, but it can speed things up when you are focusing on a particular data frame.

The third approach for extracting individual columns from a data frame is indexing with square brackets, which is described in a later section. Indexing is more flexible and powerful than the **\$** notation, but not needed for just extraction of an individual column.

The **\$** notation can also be used to add columns to a data frame. For example, if we want to add a column with combined date and time to this data frame, we can use the following code.

```
h2$date.time <- paste(h2$date, h2$time, sep = ", ")
```

And, let's say we also want a new column with biogas volume in L instead of mL:

```
h2$vol.L <- h2$vol/1000
```

Here is what our new data frame looks like:

```
head(h2)
```

```
#   reactor      date    time    vol conc.h2      date.time    vol.L
# 1    G171 9/18/2006 11:12  0.00     NA 9/18/2006, 11:12 0.00000
# 2    G171 9/18/2006 14:00  0.00   31.20 9/18/2006, 14:00 0.00000
# 3    G171 9/19/2006  9:26 11.35   35.22 9/19/2006, 9:26 0.01135
# 4    G171 9/19/2006 12:51  0.00     NA 9/19/2006, 12:51 0.00000
# 5    G171 9/19/2006 16:00  0.00     NA 9/19/2006, 16:00 0.00000
# 6    G171 9/19/2006 22:52  0.00     NA 9/19/2006, 22:52 0.00000
```

Both the **\$** notation and the **attach** and **detach** functions can be used to specify variables within any other function. However, there are other options when using functions. For some functions (e.g., **lm**, the linear model function), you can specify the data frame that should be used with the **data** argument, e.g. **data = h2**, and then refer to the column(s) directly by name. For other functions, you can use the **with** function. Although it looks a bit clunky, the **with** function can save code and help prevent user errors.

Real data often include missing observations. R has tools for working with these observations. The **na.omit** function can be used for removing **NAs** from a vector. And, several functions accept an **na.rm** argument which can be set to **TRUE** to remove missing values. Let's work with the **conc.h2** column, which contains the concentration of H<sub>2</sub> (%) volume) in the produced biogas.

```
h2$conc.h2
```

```
# [1] NA 31.20 35.22 NA NA NA NA 30.93 30.26 NA  
# [11] 0.00 7.73 10.72 15.69 21.64 23.53 26.52 25.06 NA 0.00  
# [21] 8.39 11.89 17.84 24.24 27.20 26.23 27.79 NA 0.00 7.85  
# [31] 11.54 18.72 22.38 24.77 28.46 27.06 NA 0.00 7.38 10.66  
# [41] 11.89 20.97 25.36 17.01 25.22 NA 0.00 6.57 9.29 13.08  
# [51] 16.82 22.88 22.10 NA NA NA NA NA NA NA NA  
# [61] NA  
# [71] NA  
# [81] NA NA 0.00 6.67 10.51 14.48 21.20 23.61 18.96 26.25  
# [91] NA 0.00 8.31 13.55 21.26 23.11 25.78 26.40 27.13 NA  
# [101] 0.00 7.50 11.82 17.21 21.08 22.86 22.00 26.49 NA NA  
# [111] NA  
# [121] NA  
# [131] NA NA NA NA NA
```

```
na.omit(h2$conc.h2)
```

```
# [1] 31.20 35.22 30.93 30.26 0.00 7.73 10.72 15.69 21.64 23.53 26.52  
# [12] 25.06 0.00 8.39 11.89 17.84 24.24 27.20 26.23 27.79 0.00 7.85  
# [23] 11.54 18.72 22.38 24.77 28.46 27.06 0.00 7.38 10.66 11.89 20.97  
# [34] 25.36 17.01 25.22 0.00 6.57 9.29 13.08 16.82 22.88 22.10 0.00  
# [45] 6.67 10.51 14.48 21.20 23.61 18.96 26.25 0.00 8.31 13.55 21.26  
# [56] 23.11 25.78 26.40 27.13 0.00 7.50 11.82 17.21 21.08 22.86 22.00  
# [67] 26.49  
# attr(,"na.action")  
# [1] 1 4 5 6 7 10 19 28 37 46 54 55 56 57 58 59  
# [17] 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75  
# [33] 76 77 78 79 80 81 82 91 100 109 110 111 112 113 114 115  
# [49] 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131  
# [65] 132 133 134 135  
# attr(,"class")  
# [1] "omit"
```

Although the output from `na.omit` does contain more than just the non-NA values, only the non-NA values will be used in subsequent operations<sup>73</sup>. This function can also be applied to complete data frames. In this case, any row with an NA is removed.

```
h2.clean <- na.omit(h2)  
head(h2.clean)
```

```
# reactor date time vol conc.h2 date.time vol.L  
# 2 G171 9/18/2006 14:00 0.00 31.20 9/18/2006, 14:00 0.00000  
# 3 G171 9/19/2006 9:26 11.35 35.22 9/19/2006, 9:26 0.01135  
# 8 G171 9/20/2006 13:41 2.00 30.93 9/20/2006, 13:41 0.00200  
# 9 G171 9/21/2006 12:40 0.00 30.26 9/21/2006, 12:40 0.00000  
# 11 G172 9/18/2006 14:00 0.00 0.00 9/18/2006, 14:00 0.00000  
# 12 G172 9/19/2006 9:26 19.50 7.73 9/19/2006, 9:26 0.01950
```

<sup>73</sup> The other components are attributes, which can be handy if you need them, but can generally be ignored. In this case, `na.action` records the original row numbers of elements that were removed.

It is often necessary to identify NAs present in a data structure. The `is.na` function can be used for this—it can also be negated using the `!` character.

```
is.na(h2$conc.h2)

# [1] TRUE FALSE FALSE TRUE TRUE TRUE TRUE FALSE FALSE TRUE
# [11] FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE
# [21] FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE
# [31] FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE
# [41] FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE
# [51] FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
# [61] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
# [71] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
# [81] TRUE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
# [91] TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE
# [101] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE
# [111] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
# [121] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
# [131] TRUE TRUE TRUE TRUE TRUE
```

We could count the NAs with this:

```
sum(is.na(h2$conc.h2))

# [1] 68
```

Or with `dfsumm`.

```
dfsumm(h2)

#
# 135 rows and 7 columns
# 135 unique rows
#          reactor      date    time     vol conc.h2
# Class       factor      factor factor numeric numeric
# Minimum      G171 9/18/2006 11:12      0      0
# Maximum      G185 9/21/2006  9:26   26.6   35.2
# Mean         G178 9/19/2006 14:00    7.72   17.1
# Unique (excl. NA)    15        4      9    71    59
# Missing values    0        0      0      1    68
# Sorted        TRUE     FALSE FALSE FALSE FALSE
#                  date.time    vol.L
# Class           character numeric
# Minimum        9/18/2006, 11:12      0
# Maximum        9/21/2006, 12:40  0.0266
# Mean           9/19/2006, 22:52 0.00772
# Unique (excl. NA)            9      71
# Missing values            0      1
# Sorted        FALSE     FALSE
```

In base R, the function `names()` can be used to both extract or check and change column names within a data frame. It works well for small dataframes (small number of columns), or when it is easy to refer to to columns by position.

```
pres <- read.csv("../data/presidents.csv")
dfsumm(pres)

#
# 44 rows and 8 columns
# 44 unique rows
#          order      name   birth   death
# Class     integer    factor  integer integer
# Minimum           1 Abraham Lincoln    1732    1799
# Maximum          44 Zachary Taylor   1961    2006
# Mean            22 James Madison   1833    1885
# Unique (excl. NA) 44             43     38     35
# Missing values    0              0      0      5
# Sorted           TRUE    FALSE  FALSE  FALSE
#                  party   first   last        vp
# Class           factor integer integer       factor
# Minimum         Democrat    1789    1797
# Maximum          Whig     2009    2009 William Wheeler
# Mean            National Union   1885    1889 John Adams
# Unique (excl. NA) 7        42     41      41
# Missing values    0        0      3      0
# Sorted           FALSE   TRUE   TRUE  FALSE
```

We'll use a subset here, to make the output smaller.

```
pres19 <- pres[pres$first > 1899, ]
```

To change names, including integer indexing on the left-hand side.

```
p <- pres19

names(p)

# [1] "order" "name"   "birth"  "death"  "party"  "first"  "last"   "vp"

names(p)[3:4] <- c("dob", "dod")

head(p)

#   order      name   dob   dod      party first last
# 26    26 Theodore Roosevelt 1858 1919 Republican 1901 1909
# 27    27 William Taft 1857 1930 Republican 1909 1913
# 28    28 Woodrow Wilson 1856 1924 Democrat 1913 1921
# 29    29 Warren Harding 1865 1923 Republican 1921 1923
# 30    30 Calvin Coolidge 1872 1933 Republican 1923 1929
# 31    31 Herbert C. Hoover 1874 1964 Republican 1929 1933
```

```

#           vp
# 26 Charles Fairbanks
# 27   James Sherman
# 28 Thomas Marshall
# 29 Calvin Coolidge
# 30   Charles Dawes
# 31   Charles Curtis

```

Otherwise, it is possible but clunky for referring to columns by name.

```

p <- pres19

head(p)

#   order          name birth death      party first last
# 26    26 Theodore Roosevelt 1858 1919 Republican 1901 1909
# 27    27 William Taft     1857 1930 Republican 1909 1913
# 28    28 Woodrow Wilson 1856 1924 Democrat   1913 1921
# 29    29 Warren Harding 1865 1923 Republican 1921 1923
# 30    30 Calvin Coolidge 1872 1933 Republican 1923 1929
# 31    31 Herbert C. Hoover 1874 1964 Republican 1929 1933
#
#           vp
# 26 Charles Fairbanks
# 27   James Sherman
# 28 Thomas Marshall
# 29 Calvin Coolidge
# 30   Charles Dawes
# 31   Charles Curtis

names(p)[names(p) %in% c("birth", "death")] <- c("dob", "dod")

head(p)

#   order          name   dob   dod      party first last
# 26    26 Theodore Roosevelt 1858 1919 Republican 1901 1909
# 27    27 William Taft     1857 1930 Republican 1909 1913
# 28    28 Woodrow Wilson 1856 1924 Democrat   1913 1921
# 29    29 Warren Harding 1865 1923 Republican 1921 1923
# 30    30 Calvin Coolidge 1872 1933 Republican 1923 1929
# 31    31 Herbert C. Hoover 1874 1964 Republican 1929 1933
#
#           vp
# 26 Charles Fairbanks
# 27   James Sherman
# 28 Thomas Marshall
# 29 Calvin Coolidge
# 30   Charles Dawes
# 31   Charles Curtis

```

The dplyr function `rename()` function is much easier.

```

library(dplyr)

p <- pres19
head(p)

#      order           name birth death      party first last
# 26    26 Theodore Roosevelt 1858 1919 Republican 1901 1909
# 27    27 William Taft     1857 1930 Republican 1909 1913
# 28    28 Woodrow Wilson 1856 1924 Democrat   1913 1921
# 29    29 Warren Harding 1865 1923 Republican 1921 1923
# 30    30 Calvin Coolidge 1872 1933 Republican 1923 1929
# 31    31 Herbert C. Hoover 1874 1964 Republican 1929 1933
#
#      vp
# 26 Charles Fairbanks
# 27 James Sherman
# 28 Thomas Marshall
# 29 Calvin Coolidge
# 30 Charles Dawes
# 31 Charles Curtis

p <- rename(p, dob = birth, dod = death)
head(p)

#      order           name   dob   dod      party first last
# 26    26 Theodore Roosevelt 1858 1919 Republican 1901 1909
# 27    27 William Taft     1857 1930 Republican 1909 1913
# 28    28 Woodrow Wilson 1856 1924 Democrat   1913 1921
# 29    29 Warren Harding 1865 1923 Republican 1921 1923
# 30    30 Calvin Coolidge 1872 1933 Republican 1923 1929
# 31    31 Herbert C. Hoover 1874 1964 Republican 1929 1933
#
#      vp
# 26 Charles Fairbanks
# 27 James Sherman
# 28 Thomas Marshall
# 29 Calvin Coolidge
# 30 Charles Dawes
# 31 Charles Curtis

```

Note that both plyr and dplyr have `rename` functions. These two packages \*do not\* play well together.

## 9.6 Writing data to files

With R, it is easy to write data to files. The function `write.table` (or the related function `write.csv`) is the best function for writing out data frames. Given only a data frame name and a file name, this function will write the data contained in the data frame to a text file, using spaces for separators, and putting double quotes around all character data. There are several characteristics of the file that is created that can be controlled using this function, as seen in the complete list of arguments.

```
args(write.table)

# function (x, file = "", append = FALSE, quote = TRUE, sep = " ",
#         eol = "\n", na = "NA", dec = ".", row.names = TRUE, col.names = TRUE,
#         qmethod = c("escape", "double"), fileEncoding = "")
# NULL
```

For example, if we wanted to write out the entire contents of the `h2` data frame we could use the following code<sup>74</sup>:

```
write.table(h2, "../output/h2.txt")
```

Setting the `append` argument to `TRUE` lets you add data to a file that already exists. The other arguments can be used to further control the format of the file.

Since csv files are generally easier to work with, I recommend `write.csv`. And I prefer to omit row names.

```
write.csv(h2, "../output/h2.csv", row.names = FALSE)
```

These functions are designed for data frames (or objects that can be converted to data frames), and cannot be used with all data structures in R (it won't work for lists, for example). In some cases, `write` may work for these other objects, but if not, it is always possible to send all R output to a file using the `sink` function. The following command would send all the subsequent output from R to the file `R_stuff.out`.

```
sink("R_stuff.out")
```

To go back to the default “sink”—i.e., the GUI itself, use:

```
sink()
```

---

<sup>74</sup> Note that the extension you give the file (`.txt` here) has no effect on its contents—it will always be an ASCII text file.

# 10 Working with vectors

## 10.1 Vector arithmetic and vectorized functions

In R, vectors can be used directly in arithmetic expressions, and operations are applied on an element-by-element basis. This can be referred to as “vectorized” arithmetic, and, along with vectorized functions (described below), it makes R a very efficient programming language<sup>75</sup>.

For an operation carried out on two vectors the mathematical operation is applied on an element-by-element basis.

```
midterm <- c(85, 91, 62)
final <- c(81, 94, 75)
```

```
midterm
```

```
# [1] 85 91 62
```

```
final
```

```
# [1] 81 94 75
```

```
midterm + final
```

```
# [1] 166 185 137
```

When two vectors that have different numbers of elements are used in an expression together, R will repeat the smaller vector. For example, with a vector of length one, i.e., a single number:

```
midterm*5
```

```
# [1] 90 96 67
```

If the number of rows in the larger vector is not a multiple of the smaller vector (often indicative of a code error) R will return a warning.

```
midterm*1:2
```

```
# Warning in midterm + 1:2: longer object length is not a multiple of shorter object
length
```

```
# [1] 86 93 63
```

Some simple functions that are useful for vector math include:

<sup>75</sup> Efficient for code-writers, that is. R, which is an interpreted language, will generally be slower than compiled languages, although the speed of R has improved over time.

<code>min</code>	minimum value of a set of numbers
<code>max</code>	maximum of a set of numbers
<code>pmin</code>	parallel minima (compares multiple vectors or arrays element-by-element)
<code>pmax</code>	parallel maxima
<code>sum</code>	sum of all elements
<code>length</code>	length of a vector (or the number of columns in a data frame)
<code>unique</code>	unique values present in a vector
<code>NROW</code>	number of rows in a vector or data frame
<code>all.equal</code>	comparison that includes error tolerance—use for non-integers
<code>identical</code>	compare any two objects
<code>mean</code>	arithmetic mean
<code>sd</code>	standard deviation
<code>rnorm</code>	generates a vector of normally-distributed random numbers
<code>signif</code>	round to a specified number of digits
<code>ceiling</code>	round up
<code>floor</code>	round down
<code>round</code>	round to a specified number of decimal places
<code>diff</code>	difference between adjacent elements

Order-of-operation rules apply to vectorized arithmetic as well. So

```
10^1:5
```

```
# [1] 10 9 8 7 6 5
```

is different from:

```
10^(1:5)
```

```
# [1] 1e+01 1e+02 1e+03 1e+04 1e+05
```

Many functions in R are capable of accepting vectors as input for single arguments, and returning an object with the same structure. These vectorized functions make vector manipulations very efficient. Examples of such functions include `log`, `sin`, and `sqrt`. For example,

```
x <- 1:10
sqrt(x)
```

```
# [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
# [8] 2.828427 3.000000 3.162278
```

or

```
sqrt(1:10)
```

```
# [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
# [8] 2.828427 3.000000 3.162278
```

These expressions are different from the following, where all the numbers are interpreted as individual values for multiple arguments.

```
#sqrt(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

There are also some functions designed for making vectorized (or list-sized?) operations on lists, matrices, and arrays: these include `apply`, `lapply`, and the powerful functions in the `dplyr` and `plyr` packages. We won't cover these in this workshop.

## 10.2 Working with character data

The `base` package includes many functions for manipulating character data. Here, I'll discuss a few that are particularly useful. Let's read in a data frame to work with.

```
eb <- read.csv("../data/eagles.csv")
head(eb)
```

```
#                  site achlor
# 1      magee Marsh   1.25
# 2      Magee Marsh   1.55
# 3      Magee Marsh   1.74
# 4      Magee Marsh   1.77
# 5 Davis Besse-Toussaint   1.82
# 6 Carroll Twp-Camp Perry  2.33
```

Let's work with the `site` column.

```
class(eb$site)

# [1] "factor"
```

It is a factor, since R automatically converts character data to factors. We could change it to character with `as.character`:

```
eb$site <- as.character(eb$site)
class(eb$site)

# [1] "character"
```

Or, we could have suppressed the conversion from the start.

```
eb <- read.csv("../data/eagles.csv", as.is = TRUE)
class(eb$site)

# [1] "character"
```

It is clear from looking at the first few rows that there is a problem with capitalization. But let's take a closer look.

```

unique(eb$site)

# [1] "mägee Marsh"           "Magee Marsh"
# [3] "Davis Besse-Toussaint" "Carroll Twp-Camp Perry"
# [5] "Ottawa NWR"            "Camp Perry"
# [7] "Cedar Pt NWR"          "Carroll Twp-Toussaint"
# [9] "Rossford"               "Rookery"
# [11] "Mud Ck"                 "Peach Is"
# [13] "CR 306"                 "Gibsonburg"
# [15] "Cr 306"                 "Pickeral Ck"
# [17] "Old Womans Ck"         "Ottawa SC"
# [19] "Sandusky Airport"      "Gonya"
# [21] "Ballville"              "Metzger-Peach Is"
# [23] "Sass"                   "Ft Seneca"
# [25] "Killdeer"                "Knobby"
# [27] "Indian Mill-Sycamore" "Smithville-Sycamore"
# [29] "Garlo"                  "Mercer"
# [31] "Shendngo"                "Rockwell"
# [33] "Meander"                 "Mosquito"
# [35] "Kinsman"                  "Shandngo"
# [37] "Snow Lk"                  "Pymatuning"
# [39] "Tri Valley-Wills Ck"     "Wills Ck"

```

```

length(unique(eb$site))

# [1] 40

```

We could solve this problem by making all the letters lowercase with the `tolower` function.

```

eb$site <- tolower(eb$site)
head(eb)

#                               site achlor
# 1      magee marsh    1.25
# 2      magee marsh    1.55
# 3      magee marsh    1.74
# 4      magee marsh    1.77
# 5  davis besse-toussaint  1.82
# 6 carroll twp-camp perry  2.33

length(unique(eb$site))

# [1] 38

```

The help file for this function gives the code for a more sophisticated function `capwords`. We can define the function by running the code.

```

capwords <- function(s, strict = FALSE) {
  cap <- function(s) paste(toupper(substring(s, 1, 1)),
    {s <- substring(s, 2); if (strict) tolower(s) else s},
    sep = "", collapse = " " )
  sapply(strsplit(s, split = " "), cap, USE.NAMES = !is.null(names(s)))
}

eb$site <- capwords(eb$site)
head(eb)

#          site achlor
# 1      Magee Marsh  1.25
# 2      Magee Marsh  1.55
# 3      Magee Marsh  1.74
# 4      Magee Marsh  1.77
# 5 Davis Besse-toussaint  1.82
# 6 Carroll Twp-camp Perry  2.33

length(unique(eb$site))

# [1] 38

```

What if we wanted to replace some particular character with another one? Say, for example, that we wanted to eliminate all the spaces in the `site` column? We could use the `gsub` function (for global substitution) to replace them with something else<sup>76</sup>.

```

eb$site2 <- gsub(pattern = " ", replacement = "_", x = eb$site)
head(eb)

#          site achlor          site2
# 1      Magee Marsh  1.25  Magee_Marsh
# 2      Magee Marsh  1.55  Magee_Marsh
# 3      Magee Marsh  1.74  Magee_Marsh
# 4      Magee Marsh  1.77  Magee_Marsh
# 5 Davis Besse-toussaint  1.82  Davis_Besse-toussaint
# 6 Carroll Twp-camp Perry  2.33  Carroll_Twp-camp_Perry

```

We could also use `gsub` for more sophisticated operations. Say, for example, that we wanted to spell out `Creek` instead of using `Ck`.

```

eb$site2 <- gsub("Ck", "Creek", eb$site)
eb[grep("Ck", eb$site), ]

#          site achlor          site2
# 32     Mud Ck   1.610  Mud Creek

```

<sup>76</sup> We could have also used the `chartr` function here, but `gsub` can do much more besides, so I thought I would introduce it instead. And note that `sub` is like `gsub`, but it only carries out the first substitution.

```

# 39      Pickeral Ck  0.375      Pickeral Creek
# 44      Old Womans Ck 1.010      Old Womans Creek
# 45      Pickeral Ck  1.420      Pickeral Creek
# 51      Mud Ck       2.220      Mud Creek
# 56      Pickeral Ck  1.100      Pickeral Creek
# 57      Pickeral Ck  1.180      Pickeral Creek
# 62      Pickeral Ck  1.610      Pickeral Creek
# 64      Mud Ck       1.810      Mud Creek
# 65      Mud Ck       2.170      Mud Creek
# 68      Pickeral Ck  1.380      Pickeral Creek
# 69      Pickeral Ck  1.440      Pickeral Creek
# 72      Mud Ck       1.810      Mud Creek
# 74      Old Womans Ck 1.920      Old Womans Creek
# 76      Mud Ck       1.990      Mud Creek
# 79      Mud Ck       2.920      Mud Creek
# 82      Old Womans Ck 0.375      Old Womans Creek
# 143     Tri Valley-wills Ck 3.820     Tri Valley-wills Creek
# 144     Wills Ck      1.730      Wills Creek
# 145     Wills Ck      5.450      Wills Creek
# 146     Wills Ck      3.170      Wills Creek
# 147     Wills Ck      4.140      Wills Creek

```

What if we just wanted to know where elements with certain characters resided? I used the `grep` function in the last example to do just this<sup>77</sup>.

```

grep("Ck", eb$site)

# [1] 32 39 44 45 51 56 57 62 64 65 68 69 72 74 76 79
# [17] 82 143 144 145 146 147

```

We could find all those rows where `site` contains a space with this command<sup>78</sup>:

```

grep(" ", eb$site)

# [1]   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
# [17] 17  18  19  20  21  22  24  25  26  27  28  32  33  34  36  38
# [33] 39  42  43  44  45  46  48  49  50  51  52  54  56  57  61  62
# [49] 63  64  65  68  69  70  72  73  74  75  76  79  82  83  86  88
# [65] 89  90  96  97 103 117 119 127 128 143 144 145 146 147

```

Sometimes it is necessary to split character data. In the simplest case, we can use character position to do this<sup>79</sup>. For example, what if we need to extract the year from a date?

<sup>77</sup> I used it in combination with indexing to return just those rows that had `Ck` in them. Indexing is covered in Section 14.1.

<sup>78</sup> There is also a `grep1` function that returns a logical vector.

<sup>79</sup> For more difficult cases, we can split by a particular character or sequence of characters using `strsplit`. But note that the value returned by this function is a list, not a vector, so modifying a data frame column with it is a little trickier.

```

h2 <- read.csv("../data/biohydrogen.csv", as.is = TRUE)
head(h2)

#   reactor      date    time    vol conc.h2
# 1    G171 9/18/2006 11:12  0.00     NA
# 2    G171 9/18/2006 14:00  0.00  31.20
# 3    G171 9/19/2006  9:26 11.35  35.22
# 4    G171 9/19/2006 12:51  0.00     NA
# 5    G171 9/19/2006 16:00  0.00     NA
# 6    G171 9/19/2006 22:52  0.00     NA

h2$year <- substring(h2$date, first = 6, last = 9)
head(h2)

#   reactor      date    time    vol conc.h2 year
# 1    G171 9/18/2006 11:12  0.00     NA 2006
# 2    G171 9/18/2006 14:00  0.00  31.20 2006
# 3    G171 9/19/2006  9:26 11.35  35.22 2006
# 4    G171 9/19/2006 12:51  0.00     NA 2006
# 5    G171 9/19/2006 16:00  0.00     NA 2006
# 6    G171 9/19/2006 22:52  0.00     NA 2006

```

One function that is commonly used on character data is `paste`. It concatenates character data (and can also work with numeric and logical elements—these become character data).

```

paste("A", "B", "C", TRUE, 42)

# [1] "A B C TRUE 42"

```

Note that the `paste` function is very different from `c`. The `paste` function combines its arguments into a single character value<sup>80</sup>, while the `c` function combines its arguments into a vector, where each argument becomes a single element. The `paste` function becomes handy when you want to combine the character data that are stored in several symbolic variables.

```

m <- "May"
d <- 9
yr <- 2012
paste("Today is ", m, " ", d, " ", yr, sep = "")

# [1] "Today is May 9, 2012"

```

Note that the separator is specified using the `sep` argument (default is a single space, " ")<sup>81</sup>.

Did you notice that all the examples we've covered in this section are vectorized, with the exception of the `paste` examples? The `paste` function is also vectorized.

<sup>80</sup> When each argument has only one element; it is a vectorized function.

<sup>81</sup> There is actually a new (as of v. 2.15.0) `paste0` function that is equivalent to `paste` with `sep = ""`.

```

h2$id <- paste(h2$reactor, h2$year, sep = "-")
head(h2)

#   reactor      date    time    vol conc.h2 year      id
# 1    G171 9/18/2006 11:12   0.00     NA 2006 G171-2006
# 2    G171 9/18/2006 14:00   0.00   31.20 2006 G171-2006
# 3    G171 9/19/2006  9:26  11.35   35.22 2006 G171-2006
# 4    G171 9/19/2006 12:51   0.00     NA 2006 G171-2006
# 5    G171 9/19/2006 16:00   0.00     NA 2006 G171-2006
# 6    G171 9/19/2006 22:52   0.00     NA 2006 G171-2006

```

## 11 Factors

Categorical data are most easily stored in R as factors, which are like vectors<sup>82</sup> with additional information on the acceptable levels. Factors are automatically handled as categorical data in statistical analyses. By default, R converts non-numeric data as factors when they are read in.

```

eag <- read.csv("../data/eagles.csv")
dfsumm(eag)

#
# 147 rows and 2 columns
# 126 unique rows
#           site achlor
# Class      factor numeric
# Minimum     Ballville  0.375
# Maximum     Wills Ck   5.45
# Mean        Mercer    1.54
# Unique (excl. NA) 40      92
# Missing values 0       0
# Sorted      FALSE   FALSE

```

Often, this will make your life easier<sup>83</sup>, but it won't always, and some of the time you may want to suppress this behavior. Automatic conversion can be suppressed by specifying `as.is = TRUE`<sup>84</sup> or `stringsAsFactors = FALSE` when using `read.table()` and related functions.

```

eag <- read.csv("../data/eagles.csv", as.is = TRUE)
dfsumm(eag)

#
# 147 rows and 2 columns
# 126 unique rows

```

<sup>82</sup> Internally, factors are stored as numeric data, as a check with `mode()` will tell you.

<sup>83</sup> Because the statistical analyses you carry out will go smoothly. Also, in older versions of R, factors took up much less space than character vectors, so conversion saved memory, but apparently this is no longer the case. By the way, you can check storage space used by an object with `object.size()`.

<sup>84</sup> Alternatively, this argument can be a logical vector with a value for each column, or an integer vector with the positions of the columns for which you want to suppress conversion.

```

#           site achlor
# Class      character numeric
# Minimum    Ballville  0.375
# Maximum    Wills Ck   5.45
# Mean       Meander   1.54
# Unique (excl. NA) 40     92
# Missing values 0     0
# Sorted      FALSE   FALSE

```

The `as.is` argument can actually be used to specify a value for each column—this is how it differs from `stringsAsFactors`. You can also set the behavior of `read.table()`-like functions by changing a global option using the `options()` function.

```
options(stringsAsFactors = FALSE)
```

If you ran the above command, you can change this option back with the following command.

```
options(stringsAsFactors = TRUE)
```

This is a good point to mention that changes like this, that you want to take place everytime you use R, can be to a file named `Rprofile.site` or, better, `.Rprofile`<sup>85</sup>, which will be evaluated every time you start R.

The tidyverse package developers deliberately avoided automatic conversion to factors. So the use of the `readr` and `readxl` package functions will not automatically result in factors.

Getting back to factors, in some cases you may need to create a factor from scratch, or least change an existing factor. Use the `factor()` function for these tasks.

```

a <- c("female", "male", "male", "female", "male")
a

# [1] "female" "male"    "male"    "female" "male"

a <- factor(a)
a

# [1] female male    male    female male
# Levels: female male

```

Sometimes you may want to change the ordering or factor levels, so the order will make sense in your output, or the appropriate level will be used for comparisons, for example. We are going to use the `sample()` function to create a character vector with random values of height groups for this next example.

<sup>85</sup> The file `.Rprofile` should be saved in your default working directory or your home directory. You can find your home directory with `R.home()` (don't bother to try to change it). To see what you might put in this file, look at the help file for `Startup`. For Windows, `Rprofile.site` is in `C:/Program Files/R/R-xxx/etc`. See <http://www.statmethods.net/interface/customizing.html> and `?Startup` for more information. For Linux, the file is in `etc/R`. And for Mac, check out <http://cran.r-project.org/bin/macosx/RMacOSX-FAQ.html>. As given in the example code in the file itself as installed, you can add commands that modifies the list of base packages.

```

h <- sample(c("short", "medium", "tall"), 10, replace = TRUE)
h

# [1] "short"  "medium" "tall"    "medium" "short"   "tall"    "tall"
# [8] "medium"  "medium" "tall"

```

Let's make it a factor, and then check the default order of levels.

```

hf <- factor(h)
levels(hf)

# [1] "medium" "short"  "tall"

hf

# [1] short  medium tall   medium short  tall   tall   medium medium
# [10] tall
# Levels: medium short tall

```

So R sorts the levels based on an alphabetical sorting of the original character values. To change that, use the `levels` argument.

```

hf <- factor(h, levels = c("short", "medium", "tall"))
levels(hf)

# [1] "short"  "medium" "tall"

```

There are actually two types of factors in R: unordered (what we have looked at so far) and ordered.

```

is.ordered(hf)

# [1] FALSE

```

R handles the two types differently in statistical analysis—orthogonal polynomials are used for ordered factors. To make an ordered factor, use the `ordered` argument.

```

hf <- factor(h, levels = c("short", "medium", "tall"), ordered = TRUE)
hf

# [1] short  medium tall   medium short  tall   tall   medium medium
# [10] tall
# Levels: short < medium < tall

is.ordered(hf)

# [1] TRUE

```

If you just need to change the reference level of a factor, which is relevant to interpretation of lm output, for example, you can do so with the `relevel` function.

```
hf <- factor(h)
hr <- relevel(hf, ref = "tall")
hr

# [1] short medium tall   medium short  tall   tall   medium medium
# [10] tall
# Levels: tall medium short
```

You would see a clearer effect in the summary from an lm model.

To convert numeric or integer data to a factor, just use `factor()` as above. Factor levels will be determined from the unique values.

```
x <- sample(1:4, 50, replace = TRUE)
x <- factor(x)
x

# [1] 2 3 1 4 2 3 2 2 3 2 3 3 1 1 1 3 2 2 1 2 3 2 2 3 1 1 1 3 1 4 3 4 3
# [34] 3 4 2 2 3 3 3 1 1 2 4 1 3 1 3 4 3
# Levels: 1 2 3 4
```

If you want to assign some meaningful names to integer factor levels, use the `labels` argument. Let's pretend, for example, that the levels 1 through 4 represent different kinds of supermarket products.

```
x <- factor(x, labels = c("produce", "packaged food", "dairy", "cleaning supplies"))
```

If you need to split a numeric vector into groups, use the `cut()` function.

```
w <- rnorm(15, mean = 80, sd = 15)
w

# [1] 73.89980 82.24517 102.52196 75.42059 81.32126 54.27451
# [7] 33.23667 88.69026 77.78697 51.81962 95.96783 68.27020
# [13] 74.83279 82.31583 65.33976

wf <- cut(w, breaks = 3)
wf

# [1] (56.3,79.4] (79.4,103]  (79.4,103]  (56.3,79.4] (79.4,103]
# [6] (33.2,56.3] (33.2,56.3] (79.4,103]  (56.3,79.4] (33.2,56.3]
# [11] (79.4,103]  (56.3,79.4] (56.3,79.4] (79.4,103]  (56.3,79.4]
# Levels: (33.2,56.3] (56.3,79.4] (79.4,103]
```

To convert from a factor to another type of data, you can use coercion functions.

```

a <- as.character(a)
a

# [1] "female" "male"    "male"    "female" "male"

Things are a bit trickier for going back to numeric data.

x <- sample(100:105, 20, replace = TRUE)
x

# [1] 101 105 101 101 104 105 105 100 103 105 103 102 104 103 105 100
# [17] 104 104 102 101

x <- factor(x)
x

# [1] 101 105 101 101 104 105 105 100 103 105 103 102 104 103 105 100
# [17] 104 104 102 101
# Levels: 100 101 102 103 104 105

as.numeric(x)

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

as.numeric(as.character(x))

# [1] 101 105 101 101 104 105 105 100 103 105 103 102 104 103 105 100
# [17] 104 104 102 101

```

Do you see what's going on? If you just use `as.numeric()`, R returns the underlying integers for the factor.

There is a tidyverse package dedicated to factor manipulation called `forcats`, but in my opinion the base functions will usually be sufficient.

## 12 Dates and times

Dates are handled relatively easily in R. The base packge in R includes classes for dates and combined dates and times, and functions to convert other objects into date and time objects. Importantly, dates and times are not automatically recognized by R as dates and times. It is possible to rely on only these functions, but the `lubridate` package (one of the tidyverse packages) makes dates and times easier. We'll use `lubridate` functions first.

## 12.1 The lubridate package

Let's start by loading the package<sup>86</sup>.

```
library(lubridate)

#
# Attaching package:  'lubridate'

# The following object is masked from 'package:base':
#
#   date
```

Typically, date and time data will be read in as character data<sup>87</sup>. The first step is to convert them to date and time objects<sup>88</sup>. For dates, lubridate includes a few three-letter functions that have the date format in the name, e.g., `dmy` for any “day, month, year” format.

```
d <- dmy("01 06 2010")

d

# [1] "2010-06-01"
```

It is important to know the structure of your original data! But once you do, it is pretty easy to convert them using lubridate functions.

```
dmy("01 06 2010")

# [1] "2010-06-01"

mdy("06 01 2010")

# [1] "2010-06-01"

ymd("2010 06 01")

# [1] "2010-06-01"
```

Separators generally do not matter.

```
dmy("01-06-2010")

# [1] "2010-06-01"
```

<sup>86</sup> Remember that is must be installed first.

<sup>87</sup> Or factors.

<sup>88</sup> The base functions distinguish between date and combined dates and time, but lubridate uses only combined dates and times. Dates without time information are assumed to be at 00:00.

```
dmy("01, 06, 2010")
```

```
# [1] "2010-06-01"
```

```
dmy("1 6 2010")
```

```
# [1] "2010-06-01"
```

```
dmy("1 June, 2010")
```

```
# [1] "2010-06-01"
```

These objects can be manipulated like numeric vectors; for example, you can add or subtract days. To add one day, for example, use this:

```
d + 1
```

```
# [1] "2010-06-02"
```

If this seems confusing, you can use the lubridate `d*` functions instead. Here, `d` is for duration. For example, to add one day, use `ddays`.

```
d + ddays(1)
```

```
# [1] "2010-06-02"
```

Subtraction can be done with the normal `-` operator.

```
d1 <- dmy("01 06 2010")
```

```
d2 <- dmy("10 06 2010")
```

```
d2 - d1
```

```
# Time difference of 9 days
```

The output has class `difftime`, which includes a `units` attribute. For control over units, use `difftime` instead.

```
difftime(d2, d1, units = "hours")
```

```
# Time difference of 216 hours
```

How about working with combined date and time objects? To specify hours, minutes, and seconds, append some combination of `h`, `m`, or `s` to the functions used above.

```

dmy_hm("01 06 2010 12:01")

# [1] "2010-06-01 12:01:00 UTC"

dmy_hms("01 06 2010 12:01:10")

# [1] "2010-06-01 12:01:10 UTC"

```

Again, there is a lot of flexibility in what the functions can parse.

Let's work on a more realistic example. Say you've read in the data in a file, and need to convert the time information in order to, e.g., calculate elapsed time. Here we'll work with the first 10 rows from the file biohydrogen.csv.

```

h2 <- read.csv("../data/biohydrogen.csv", as.is = TRUE)

h2 <- h2[1:10, ]

head(h2)

#   reactor      date    time    vol conc.h2
# 1    G171 9/18/2006 11:12  0.00     NA
# 2    G171 9/18/2006 14:00  0.00  31.20
# 3    G171 9/19/2006  9:26 11.35  35.22
# 4    G171 9/19/2006 12:51  0.00     NA
# 5    G171 9/19/2006 16:00  0.00     NA
# 6    G171 9/19/2006 22:52  0.00     NA

```

Let's add a new column with combine date and time.

To create a combined date and time vector, we'll need to first combine these two together.

```

h2$date.time <- paste(h2$date, h2$time)
head(h2)

#   reactor      date    time    vol conc.h2      date.time
# 1    G171 9/18/2006 11:12  0.00     NA 9/18/2006 11:12
# 2    G171 9/18/2006 14:00  0.00  31.20 9/18/2006 14:00
# 3    G171 9/19/2006  9:26 11.35  35.22 9/19/2006  9:26
# 4    G171 9/19/2006 12:51  0.00     NA 9/19/2006 12:51
# 5    G171 9/19/2006 16:00  0.00     NA 9/19/2006 16:00
# 6    G171 9/19/2006 22:52  0.00     NA 9/19/2006 22:52

```

And we can now use the appropriate lubridate function.

```

h2$date.time <- mdy_hm(h2$date.time)
head(h2)

#   reactor      date    time    vol conc.h2      date.time
# 1    G171 9/18/2006 11:12  0.00     NA 2010-06-01 12:01:00
# 2    G171 9/18/2006 14:00  0.00  31.20 2010-06-01 14:00:00
# 3    G171 9/19/2006  9:26 11.35  35.22 2010-06-01 09:26:00
# 4    G171 9/19/2006 12:51  0.00     NA 2010-06-01 12:51:00
# 5    G171 9/19/2006 16:00  0.00     NA 2010-06-01 16:00:00
# 6    G171 9/19/2006 22:52  0.00     NA 2010-06-01 22:52:00

```

```

# 1 G171 9/18/2006 11:12 0.00      NA 2006-09-18 11:12:00
# 2 G171 9/18/2006 14:00 0.00    31.20 2006-09-18 14:00:00
# 3 G171 9/19/2006  9:26 11.35   35.22 2006-09-19 09:26:00
# 4 G171 9/19/2006 12:51 0.00      NA 2006-09-19 12:51:00
# 5 G171 9/19/2006 16:00 0.00      NA 2006-09-19 16:00:00
# 6 G171 9/19/2006 22:52 0.00      NA 2006-09-19 22:52:00

```

And, for example, to calculate the elapsed time since the first observation, we could use the following expression.

```

h2$date.time - h2$date.time[1]

# Time differences in secs
# [1]      0 10080 80040 92340 103680 128400 164400 181740 264480
# [10]      0

```

Or, maybe this is more useful.

```

difftime(h2$date.time, h2$date.time[1], units = "hours")

# Time differences in hours
# [1] 0.00000 2.80000 22.23333 25.65000 28.80000 35.66667 45.66667
# [8] 50.48333 73.46667 0.00000

```

The lubridate package includes several other functions for working with dates and times that are not covered here<sup>89</sup>

## 12.2 Dates and times in base R

Functions available in the base package can be used for the same operations that were described above for lubridate. The functions are a bit less forgiving though. In most cases (maybe all), you can avoid using them.

If you are working with just dates and do not need information on time, work with **Dates** objects, which are represented as the number of days since the beginning of 1970 (generally as integer data). Let's work with some date data.

```

flow <- read.csv("../data/river_flow.csv")
dfsumm(flow)

#
# 730 rows and 5 columns
# 730 unique rows
#           agency     site       date discharge flag.discharge
# Class          factor  integer    factor   integer        factor
# Minimum        USGS 1509000 2006-01-01        31            A

```

<sup>89</sup> You can find more details here: <https://cran.r-project.org/web/packages/lubridate/vignettes/lubridate.html> or in the package vignette.

```

# Maximum          USGS 4232730 2006-12-31      6050      P
# Mean            USGS 1509000 2006-07-02      612       Ae
# Unique (excl. NA)    1      2      365      497      3
# Missing values      0      0      0      1      1
# Sorted           TRUE FALSE FALSE FALSE FALSE

```

Well, if we want to work with dates explicitly, we can't have R treating that column as a factor. To convert to a `Date` object, use `as.Date()`.

```

flow$date <- as.Date(flow$date)
dfsumm(flow)

#
# 730 rows and 5 columns
# 730 unique rows
#
#              agency     site      date discharge flag.discharge
# Class        factor integer     Date   integer      factor
# Minimum      USGS 1509000 2006-01-01      31          A
# Maximum      USGS 4232730 2006-12-31      6050          P
# Mean         USGS 1509000 2006-07-02      612       Ae
# Unique (excl. NA)    1      2      365      497      3
# Missing values      0      0      0      1      1
# Sorted           TRUE FALSE FALSE FALSE FALSE

```

Note that it is not necessary to convert factors to character data before converting them to a `Date` object. However, if the original data are integers or numeric, you generally will have to convert them to character data first with `as.character()`<sup>90</sup>. The dates are now of class `Date`, and it is possible to carry out mathematical operations on them.

```

class(flow$date)

# [1] "Date"

```

For example, if we wanted to subtract one year, or calculate the mean date,

```

x <- flow$date - 365
head(x)

# [1] "2005-01-01" "2005-01-02" "2005-01-03" "2005-01-04" "2005-01-05"
# [6] "2005-01-06"

mean(flow$date)

# [1] "2006-07-02"

```

If you prefer, it is possible to specify that a file column contains `Date` data when those data are read in.

---

<sup>90</sup> Unless they are the number of days after some particular date. See the help file for more information.

```

flow <- read.csv("../data/river_flow.csv", colClasses = c("character", "integer", "Date", "numeric",
dfsumm(flow)

#
# 730 rows and 5 columns
# 730 unique rows
#          agency    site      date discharge
# Class      character integer     Date   numeric
# Minimum      USGS 1509000 2006-01-01        31
# Maximum      USGS 4232730 2006-12-31       6050
# Mean         USGS 1509000 2006-07-02       843
# Unique (excl. NA) 1      2      365      497
# Missing values 0      0      0      1
# Sorted        TRUE FALSE FALSE FALSE
#          flag.discharge
# Class      character
# Minimum      A
# Maximum      P
# Mean         A
# Unique (excl. NA) 3
# Missing values 1
# Sorted        FALSE

```

The above examples gloss over an important point: R can't always determine the format of your dates automatically. The default format for dates in R starts with the year, then month, then day, e.g., 2009-12-10 for December 10, 2009, which is specified in R as "%Y-%m-%d". Our `date` column matched this format, so we didn't need to specify a format, but that isn't always the case. A list of date and time abbreviations can be found in the help file for `strptime()`, and a partial list is given below.

Symbol	It means	Example
%Y	Year with century	2012
%y	Year without century	12
%B	Full month	December
%b	Abbreviated month	Dec
%m	Integer month	12
%d	Day	8
%x	Date	2012/12/08

If your dates are in a different format, you need to indicate the format in order to convert them to a `Date` object. Here is an example with a date format that doesn't match the default.

```

h2 <- read.csv("../data/biohydrogen.csv")
head(h2$date)

# [1] 9/18/2006 9/18/2006 9/19/2006 9/19/2006 9/19/2006 9/19/2006
# Levels: 9/18/2006 9/19/2006 9/20/2006 9/21/2006

```

Looks like it is "%m/%d/%Y".

```

h2$date <- as.Date(h2$date, format = c("%m/%d/%Y"))
head(h2$date)

# [1] "2006-09-18" "2006-09-18" "2006-09-19" "2006-09-19" "2006-09-19"
# [6] "2006-09-19"

```

To go from Date objects to numeric or character classes, use `as.numeric()` and `as.character()`. These and other coercion functions are actually generic, and the methods for Date objects accept a `format` argument.

```

as.character(h2$date, format = "%b")

# [1] "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep"
# [11] "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep"
# [21] "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep"
# [31] "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep"
# [41] "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep"
# [51] "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep"
# [61] "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep"
# [71] "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep"
# [81] "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep"
# [91] "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep"
# [101] "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep"
# [111] "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep"
# [121] "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep" "Sep"
# [131] "Sep" "Sep" "Sep" "Sep" "Sep" "Sep"

```

```

head(as.character(h2$date, format = "%B %d %Y"))

# [1] "September 18 2006" "September 18 2006" "September 19 2006"
# [4] "September 19 2006" "September 19 2006" "September 19 2006"

```

Moving on to date-time objects, there are two basic date-time classes in R: `POSIXct` and `POSIXlt`. The first class contains the number of seconds since the beginning of 1970 as a numeric vector, while the second is actually a list that contains a vectors of: dates, times, and the time zone. You can use the `$` notation to return specific columns.

To create a date-time object with the data in `h2`, we will need to combine the `date` and `time` columns. Then we can convert the new column to a combined date/time class.

```

h2 <- read.csv("../data/biohydrogen.csv")
h2$dt <- paste(h2$date, h2$time)
head(h2)

#   reactor      date    time    vol conc.h2          dt
# 1     G171 9/18/2006 11:12  0.00      NA 9/18/2006 11:12
# 2     G171 9/18/2006 14:00  0.00  31.20 9/18/2006 14:00
# 3     G171 9/19/2006  9:26 11.35  35.22 9/19/2006  9:26
# 4     G171 9/19/2006 12:51  0.00      NA 9/19/2006 12:51

```

```
# 5      G171 9/19/2006 16:00  0.00      NA 9/19/2006 16:00
# 6      G171 9/19/2006 22:52  0.00      NA 9/19/2006 22:52
```

`POSIXct` objects are a better fit for data frames, and we'll use that class here. As with `as.Date()`, there are abbreviations used for the different time units. An abbreviated list is given below, and you can find more details in the help file for `strptime()`.

Symbol	It means	Example
%H	00-24 hour	13
%I	00-12 hour	1
%p	am or pm (%I only!)	pm
%M	Minute	32
%S	Second	59
%X	Time	13:32:59

Now for the conversion.

```
h2$dt <- as.POSIXct(h2$dt, format = "%m/%d/%Y %H:%M", tz = "EST")
head(h2$dt)

# [1] "2006-09-18 11:12:00 EST" "2006-09-18 14:00:00 EST"
# [3] "2006-09-19 09:26:00 EST" "2006-09-19 12:51:00 EST"
# [5] "2006-09-19 16:00:00 EST" "2006-09-19 22:52:00 EST"
```

It looks like that worked. Note that we could have nested the `paste()` call in the `as.POSIXlt()` call. And we don't actually need to provide a time zone argument, by the way. Now we can work with this new column as you might expect.

```
mean(h2$dt)

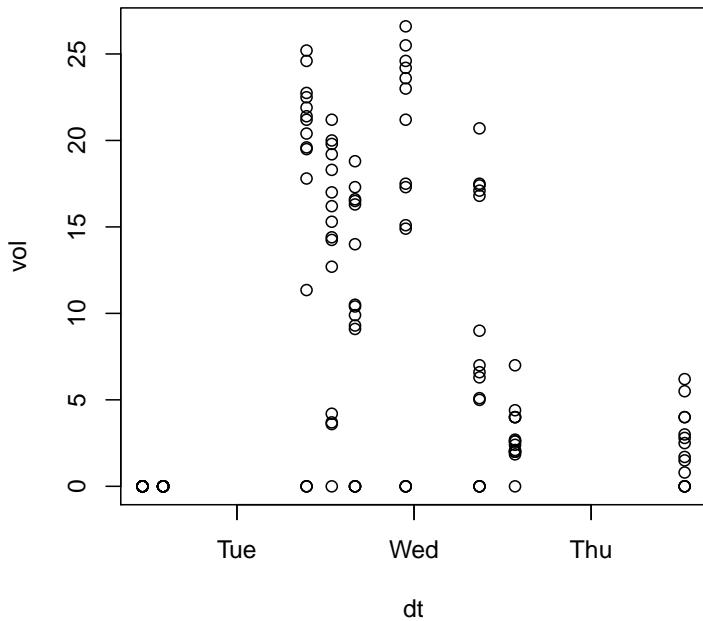
# [1] "2006-09-19 18:50:26 EST"
```

One of the good things about date-time objects in R is that all the graphics functions seem to handle them intelligently automatically.

```
head(h2)

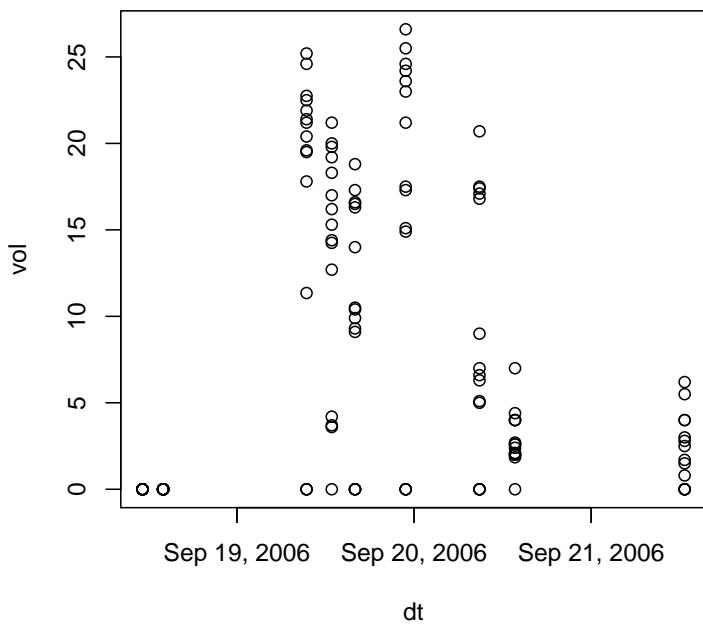
#   reactor      date    time    vol conc.h2          dt
# 1      G171 9/18/2006 11:12  0.00      NA 2006-09-18 11:12:00
# 2      G171 9/18/2006 14:00  0.00  31.20 2006-09-18 14:00:00
# 3      G171 9/19/2006  9:26 11.35  35.22 2006-09-19 09:26:00
# 4      G171 9/19/2006 12:51  0.00      NA 2006-09-19 12:51:00
# 5      G171 9/19/2006 16:00  0.00      NA 2006-09-19 16:00:00
# 6      G171 9/19/2006 22:52  0.00      NA 2006-09-19 22:52:00

plot(vol ~ dt, data = h2)
```



And they are easy to change if you don't like the defaults.

```
plot(vol ~ dt, data = h2, xaxt = "n")
axis.POSIXct(1, h2$dt, format = "%b %d, %Y")
```



The default display of the other combined date/time object class (`POSIXlt`) looks similar, but it is a list with separate vectors for seconds, minutes, days, etc.

```
h2$dtlt <- as.POSIXlt(paste(h2$date, h2$time), format = "%m/%d/%Y %H:%M", tz = "EST")
head(h2)

#   reactor      date    time    vol conc.h2          dt
# 1     G171 9/18/2006 11:12    0.00      NA 2006-09-18 11:12:00
# 2     G171 9/18/2006 14:00    0.00    31.20 2006-09-18 14:00:00
# 3     G171 9/19/2006  9:26   11.35    35.22 2006-09-19 09:26:00
# 4     G171 9/19/2006 12:51    0.00      NA 2006-09-19 12:51:00
# 5     G171 9/19/2006 16:00    0.00      NA 2006-09-19 16:00:00
# 6     G171 9/19/2006 22:52    0.00      NA 2006-09-19 22:52:00
#
#                  dtlt
# 1 2006-09-18 11:12:00
# 2 2006-09-18 14:00:00
# 3 2006-09-19 09:26:00
# 4 2006-09-19 12:51:00
# 5 2006-09-19 16:00:00
# 6 2006-09-19 22:52:00
```

This is handy when you need to access individual components of a combined date/time object—they can be accessed through indexing.

```
h2$dtlt$mday
```

```
# [1] 18 18 19 19 19 19 20 20 21 18 18 19 19 19 19 20 20 21 18 18 19
# [22] 19 19 19 20 20 21 18 18 19 19 19 19 20 20 21 18 18 19 19 19 19
# [43] 20 20 21 18 18 19 19 19 19 20 20 21 18 18 19 19 19 19 19 20 20 21
# [64] 18 18 19 19 19 19 20 20 21 18 18 19 19 19 19 20 20 21 18 18 19
# [85] 19 19 19 20 20 21 18 18 19 19 19 19 20 20 21 18 18 19 19 19 19
# [106] 20 20 21 18 18 19 19 19 19 20 20 21 18 18 19 19 19 19 20 20 21
# [127] 18 18 19 19 19 19 20 20 21
```

```
h2$dtlt$hour
```

```
# [1] 11 14  9 12 16 22  8 13 12 11 14  9 12 16 22  8 13 12 11 14  9
# [22] 12 16 22  8 13 12 11 14  9 12 16 22  8 13 12 11 14  9 12 16 22
# [43]  8 13 12 11 14  9 12 16 22  8 13 12 11 14  9 12 16 22  8 13 12
# [64] 11 14  9 12 16 22  8 13 12 11 14  9 12 16 22  8 13 12 11 14  9
# [85] 12 16 22  8 13 12 11 14  9 12 16 22  8 13 12 11 14  9 12 16 22
# [106] 8 13 12 11 14  9 12 16 22  8 13 12 11 14  9 12 16 22  8 13 12
# [127] 11 14  9 12 16 22  8 13 12
```

You can find more information in the help file for the class, i.e., `POSIXlt`.

Time differences between combined date/time objects can be calculated using arithmetic operators.

```
h2$dt - as.POSIXct("2006/01/01")
```

```
# Time differences in days
```

```

# [1] 260.4667 260.5833 261.3931 261.5354 261.6667 261.9528 262.3694
# [8] 262.5701 263.5278 260.4667 260.5833 261.3931 261.5354 261.6667
# [15] 261.9528 262.3694 262.5701 263.5278 260.4667 260.5833 261.3931
# [22] 261.5354 261.6667 261.9528 262.3694 262.5701 263.5278 260.4667
# [29] 260.5833 261.3931 261.5354 261.6667 261.9528 262.3694 262.5701
# [36] 263.5278 260.4667 260.5833 261.3931 261.5354 261.6667 261.9528
# [43] 262.3694 262.5701 263.5278 260.4667 260.5833 261.3931 261.5354
# [50] 261.6667 261.9528 262.3694 262.5701 263.5278 260.4667 260.5833
# [57] 261.3931 261.5354 261.6667 261.9528 262.3694 262.5701 263.5278
# [64] 260.4667 260.5833 261.3931 261.5354 261.6667 261.9528 262.3694
# [71] 262.5701 263.5278 260.4667 260.5833 261.3931 261.5354 261.6667
# [78] 261.9528 262.3694 262.5701 263.5278 260.4667 260.5833 261.3931
# [85] 261.5354 261.6667 261.9528 262.3694 262.5701 263.5278 260.4667
# [92] 260.5833 261.3931 261.5354 261.6667 261.9528 262.3694 262.5701
# [99] 263.5278 260.4667 260.5833 261.3931 261.5354 261.6667 261.9528
# [106] 262.3694 262.5701 263.5278 260.4667 260.5833 261.3931 261.5354
# [113] 261.6667 261.9528 262.3694 262.5701 263.5278 260.4667 260.5833
# [120] 261.3931 261.5354 261.6667 261.9528 262.3694 262.5701 263.5278
# [127] 260.4667 260.5833 261.3931 261.5354 261.6667 261.9528 262.3694
# [134] 262.5701 263.5278

```

For control over the output units use `difftime()`.

```

difftime(h2$dt, as.POSIXct("2006/01/01"), units = "hours")

# Time differences in hours
# [1] 6251.200 6254.000 6273.433 6276.850 6280.000 6286.867 6296.867
# [8] 6301.683 6324.667 6251.200 6254.000 6273.433 6276.850 6280.000
# [15] 6286.867 6296.867 6301.683 6324.667 6251.200 6254.000 6273.433
# [22] 6276.850 6280.000 6286.867 6296.867 6301.683 6324.667 6251.200
# [29] 6254.000 6273.433 6276.850 6280.000 6286.867 6296.867 6301.683
# [36] 6324.667 6251.200 6254.000 6273.433 6276.850 6280.000 6286.867
# [43] 6296.867 6301.683 6324.667 6251.200 6254.000 6273.433 6276.850
# [50] 6280.000 6286.867 6296.867 6301.683 6324.667 6251.200 6254.000
# [57] 6273.433 6276.850 6280.000 6286.867 6296.867 6301.683 6324.667
# [64] 6251.200 6254.000 6273.433 6276.850 6280.000 6286.867 6296.867
# [71] 6301.683 6324.667 6251.200 6254.000 6273.433 6276.850 6280.000
# [78] 6286.867 6296.867 6301.683 6324.667 6251.200 6254.000 6273.433
# [85] 6276.850 6280.000 6286.867 6296.867 6301.683 6324.667 6251.200
# [92] 6254.000 6273.433 6276.850 6280.000 6286.867 6296.867 6301.683
# [99] 6324.667 6251.200 6254.000 6273.433 6276.850 6280.000 6286.867
# [106] 6296.867 6301.683 6324.667 6251.200 6254.000 6273.433 6276.850
# [113] 6280.000 6286.867 6296.867 6301.683 6324.667 6251.200 6254.000
# [120] 6273.433 6276.850 6280.000 6286.867 6296.867 6301.683 6324.667
# [127] 6251.200 6254.000 6273.433 6276.850 6280.000 6286.867 6296.867
# [134] 6301.683 6324.667

```

Two other useful time-related functions are `Sys.time()`, which will return the current time, and `system.time()`, which tells you how long an expression takes to run.

```
Sys.time()

# [1] "2019-08-08 22:19:37 EDT"

system.time(
  for (i in 1:1E5) {
    rnorm(1000)
  }
)

#   user  system elapsed
# 6.887  0.000  6.888
```

## 13 Exploratory data analysis

### 13.1 Summary statistics

Let's demonstrate calculation of summary statistics with data set on metal toxicity in soil:

```
cu <- read.csv("../data/Cu_ec50s.csv")
cu

#          soil ec50.cu ph.soil      oc ph.sol c.cu c.na  c.mg   c.k
# 1  Nottingham    150.5    3.36  5.20    3.72 1.33 29.0  34.3  65.6
# 2   Houthalen     38.9    3.38  1.90    3.71 0.70 10.1   4.0  31.0
# 3  Rhydtalog     194.0    4.20 12.90    4.47 0.83 25.1  49.4  44.1
# 4    Zegveld      570.5    4.75 23.30    3.88 0.48 38.6 124.0  76.7
# 5  Kovlinge I    101.4    4.76  1.60    4.88 0.30 12.7  37.5  30.2
# 6    Souli I      64.8    4.80  0.41    5.09 0.06 11.1  13.4   0.0
# 7  Kovlinge II   168.8    5.06  2.40    5.22 0.27 14.0  41.1  34.7
# 8  Montpellier    61.2    5.18  0.76    5.63 0.95 24.9  38.4  23.9
# 9  Aluminusa     147.3    5.44  0.87    6.12 0.05 62.3  43.2  15.7
# 10   Woburn       392.2    6.36  4.40    7.37 0.50 17.6  94.5  24.1
# 11 Ter Munck     227.6    6.80  0.98    7.35  NA 32.7  65.9 124.0
#  c.ca
# 1  98.4
# 2  21.4
# 3 214.0
# 4 799.0
# 5 140.0
# 6  72.0
# 7 218.0
# 8 147.0
# 9 120.0
# 10 412.0
# 11 704.0
```

Here are some useful functions:

```
mean(cu$ec50)

# [1] 192.4727

median(cu$ec50)

# [1] 150.5

sd(cu$ec50)

# [1] 159.313

var(cu$ec50)

# [1] 25380.62
```

R has a function for summarizing vectors or data frames called `summary`. This function is a generic function—what it returns is dependent on the type of data submitted to it<sup>91</sup>. Let's apply `summary` to the `cu` data frame:

```
summary(cu)

#       soil      ec50.cu      ph.soil        oc
# Aluminusa :1   Min.   : 38.9   Min.   :3.360   Min.   : 0.410
# Houthalen  :1   1st Qu.: 83.1   1st Qu.:4.475   1st Qu.: 0.925
# Kovlinge I :1   Median :150.5   Median :4.800   Median : 1.900
# Kovlinge II:1   Mean   :192.5   Mean   :4.917   Mean   : 4.975
# Montpellier:1   3rd Qu.:210.8   3rd Qu.:5.310   3rd Qu.: 4.800
# Nottingham :1   Max.   :570.5   Max.   :6.800   Max.   :23.300
# (Other)    :5

#       ph.sol      c.cu      c.na        c.mg
# Min.   :3.710   Min.   :0.0500   Min.   :10.10   Min.   : 4.00
# 1st Qu.:4.175   1st Qu.:0.2775   1st Qu.:13.35   1st Qu.: 35.90
# Median :5.090   Median :0.4900   Median :24.90   Median : 41.10
# Mean   :5.222   Mean   :0.5470   Mean   :25.28   Mean   : 49.61
# 3rd Qu.:5.875   3rd Qu.:0.7975   3rd Qu.:30.85   3rd Qu.: 57.65
# Max.   :7.370   Max.   :1.3300   Max.   :62.30   Max.   :124.00
# NA's    :1

#       c.k      c.ca
# Min.   : 0.00   Min.   :21.4
# 1st Qu.: 24.00  1st Qu.:109.2
# Median : 31.00  Median :147.0
# Mean   : 42.73  Mean   :267.8
# 3rd Qu.: 54.85  3rd Qu.:315.0
# Max.   :124.00  Max.   :799.0
#
```

Notice the difference between numeric and categorical variables—the `summary` function can be used to check the classes of columns in a data frame. But the `str` function (for structure) is a more direct way to do this.

```
str(cu)

# 'data.frame': 11 obs. of  10 variables:
# $ soil    : Factor w/ 11 levels "Aluminusa","Houthalen",...: 6 2 7 11 3 8 4 5 1 10 ...
# $ ec50.cu: num  150.5 38.9 194 570.5 101.4 ...
# $ ph.soil: num  3.36 3.38 4.2 4.75 4.76 4.8 5.06 5.18 5.44 6.36 ...
# $ oc     : num  5.2 1.9 12.9 23.3 1.6 0.41 2.4 0.76 0.87 4.4 ...
# $ ph.sol : num  3.72 3.71 4.47 3.88 4.88 5.09 5.22 5.63 6.12 7.37 ...
# $ c.cu   : num  1.33 0.7 0.83 0.48 0.3 0.06 0.27 0.95 0.05 0.5 ...
# $ c.na   : num  29 10.1 25.1 38.6 12.7 11.1 14 24.9 62.3 17.6 ...
# $ c.mg   : num  34.3 4 49.4 124 37.5 13.4 41.1 38.4 43.2 94.5 ...
# $ c.k    : num  65.6 31 44.1 76.7 30.2 0 34.7 23.9 15.7 24.1 ...
# $ c.ca   : num  98.4 21.4 214 799 140 72 218 147 120 412 ...
```

---

<sup>91</sup> Another important use of the `summary` function is in summarizing the results of statistical models, which is discussed in a later section.

## 13.2 Counts and contingency tables

To count observations that meet some criterion, you can nest a logical expression in a call to the `sum` function.

```
sum(cu$ph.soil > 4)
```

```
# [1] 9
```

But the `table` function is a better approach when dealing with multiple factor levels for individual variables, or even multiple variables. For example, let's take a look at some data on soil respiration in Alaskan ecosystems.

```
gas <- read.csv("../data/gas_emis.csv")
dfsumm(gas)
```

```
#
# 326 rows and 8 columns
# 326 unique rows
#          ecosystem      date    tmt   block morphology
# Class      factor      factor factor integer   factor
# Minimum    HEATH 1993/06/21     CT      1       I
# Maximum    TUSSOCK 1993/08/18    NP      4       T
# Mean        SEDGE 1993/07/21    GH      2       T
# Unique (excl. NA) 3           11      3       4       2
# Missing values 0           0       0       0       0
# Sorted      FALSE     TRUE  FALSE  FALSE  FALSE
#             CH4      CO2    temp
# Class      numeric integer numeric
# Minimum    -14.7      16    1.98
# Maximum    648       12783   13.6
# Mean       44.7      1627   5.66
# Unique (excl. NA) 322       311    54
# Missing values 0           0    206
# Sorted      FALSE  FALSE  FALSE
```

The `table` function can be used for counts for all combinations of factor levels.

```
table(gas$ecosyste, gas$tmt)
```

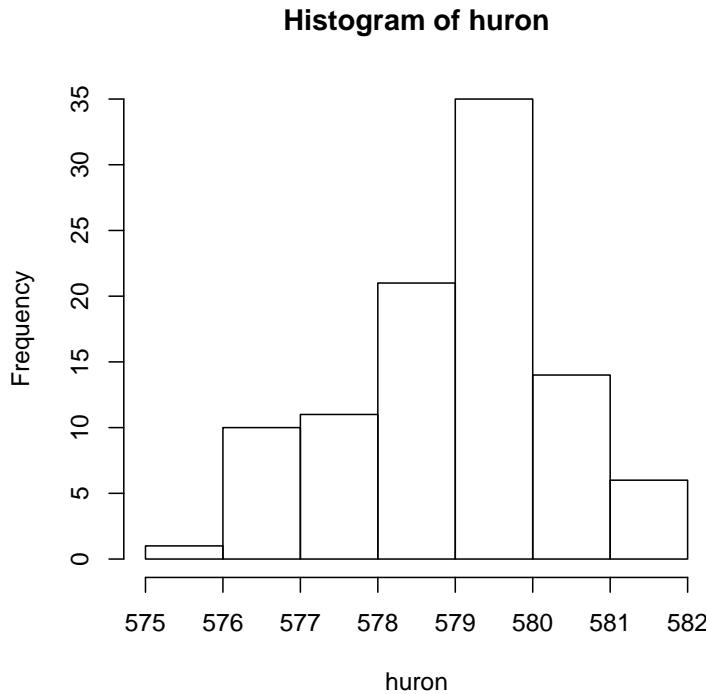
```
#
#          CT  GH  NP
# HEATH    30  26  30
# SEDGE    30  30  30
# TUSSOCK 50  50  50
```

## 13.3 Histograms and other summary plots

Histograms, stripcharts, and boxplot are simple but useful ways of summarizing data. You can generate a histogram in R using the function `hist`. Let's start out with the `LakeHuron` data set,

which has data on the water level of the lake for almost 100 years.

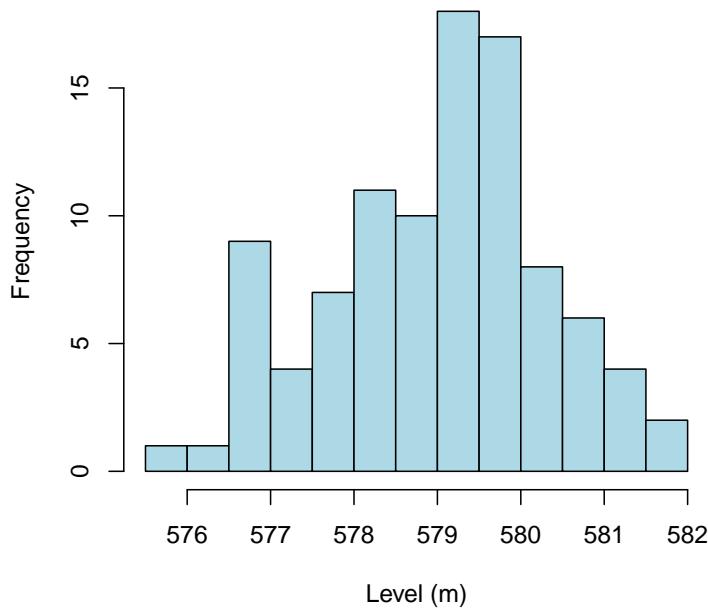
```
huron <- as.numeric(LakeHuron)
hist(huron)
```



This plot can be made to look a little nicer, as can all the plots covered in this workshop (several arguments, such as `xlab`, `ylab`, and `main`, can be used in most plotting functions). You can also specify the number or location of breaks in the `hist` function, and you should try a few different values for any data set. For example, try the following command.

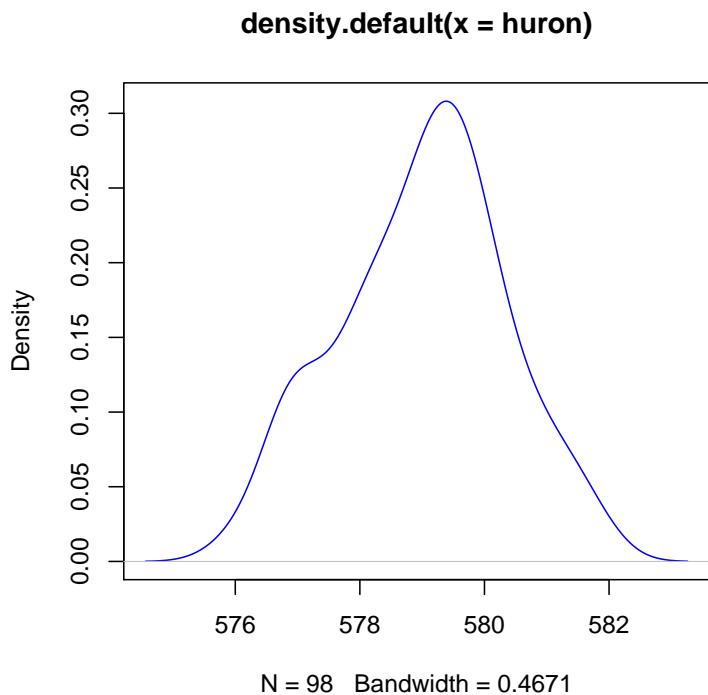
```
hist(huron, breaks = 20, col = "lightblue", main = "Lake Huron level", xlab = "Level (m)")
```

## Lake Huron level



If you have a large data set, and are interested in the distribution of the data, try plotting the output from the `density` function, which returns estimates of the probability density.

```
plot(density(huron), col = "blue")
```

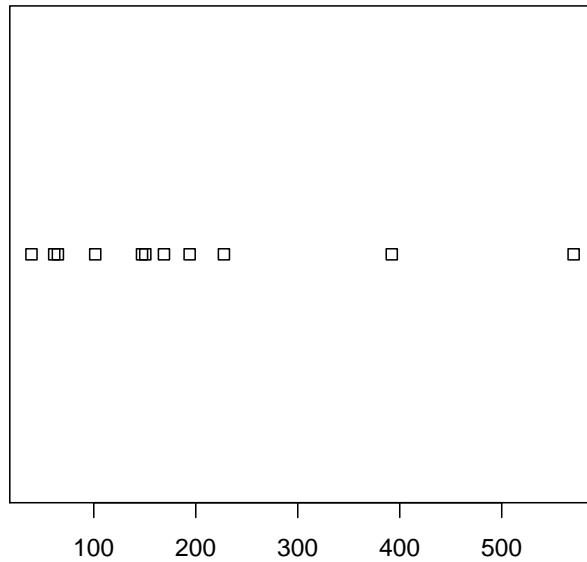


For data sets with a small number of observations, the **stripchart** function is a good choice.

```
cu <- read.csv("../data/Cu_ec50s.csv")
cu

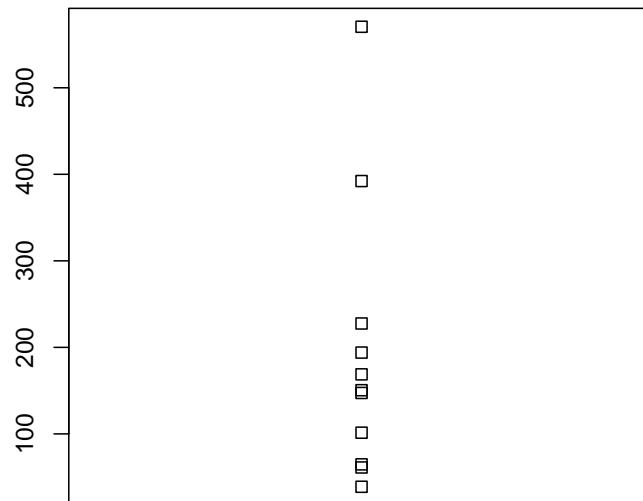
#      soil ec50.cu ph.soil      oc ph.sol c.cu c.na c.mg   c.k
# 1 Nottingham  150.5    3.36  5.20  3.72 1.33 29.0  34.3 65.6
# 2 Houthalen   38.9     3.38  1.90  3.71 0.70 10.1   4.0 31.0
# 3 Rhydtalog   194.0    4.20 12.90  4.47 0.83 25.1  49.4 44.1
# 4 Zegveld     570.5    4.75 23.30  3.88 0.48 38.6 124.0 76.7
# 5 Kovlinge I 101.4    4.76  1.60  4.88 0.30 12.7  37.5 30.2
# 6 Souli I     64.8     4.80  0.41  5.09 0.06 11.1  13.4  0.0
# 7 Kovlinge II 168.8    5.06  2.40  5.22 0.27 14.0  41.1 34.7
# 8 Montpellier 61.2     5.18  0.76  5.63 0.95 24.9  38.4 23.9
# 9 Aluminusa   147.3    5.44  0.87  6.12 0.05 62.3  43.2 15.7
# 10 Woburn     392.2    6.36  4.40  7.37 0.50 17.6  94.5 24.1
# 11 Ter Munck  227.6    6.80  0.98  7.35  NA 32.7  65.9 124.0
#
#      c.ca
# 1 98.4
# 2 21.4
# 3 214.0
# 4 799.0
# 5 140.0
# 6 72.0
# 7 218.0
# 8 147.0
# 9 120.0
# 10 412.0
# 11 704.0

stripchart(cu$ec50)
```



You may find it more intuitive to change the orientation (I do).

```
stripchart(cu$ec50, vertical = TRUE)
```

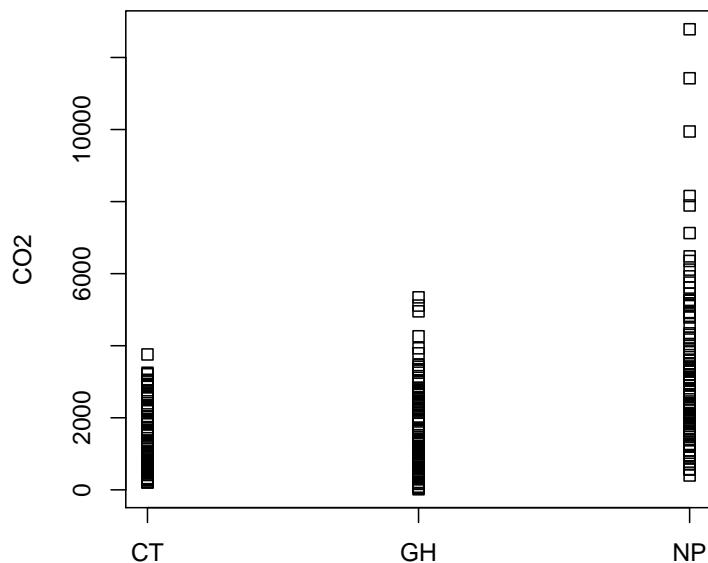


I think `stripchart` is one of the best functions for visualizing data after reading them in. The `stripchart` function can also plot a response for multiple levels of a factor. For example, let's take a look at some data on soil respiration in Alaskan ecosystems.

```
gas <- read.csv("../data/gas_emis.csv")
dfsumm(gas)

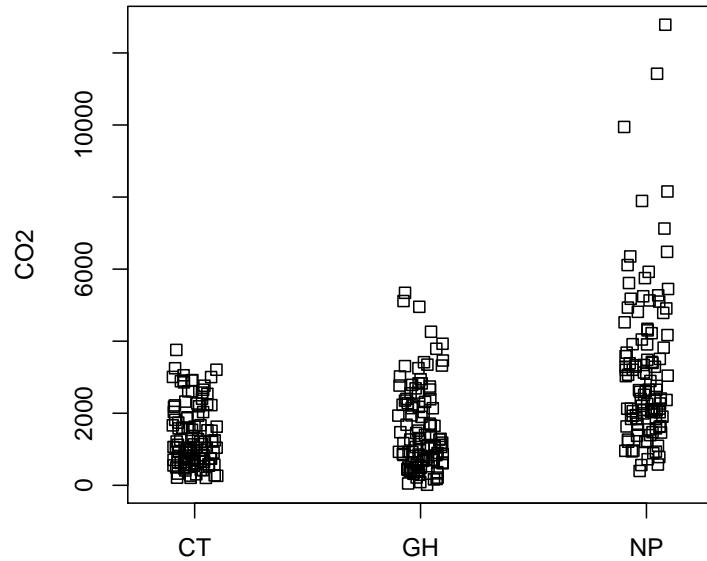
#
# 326 rows and 8 columns
# 326 unique rows
#          ecosystem      date    tmt   block morphology
# Class       factor      factor factor integer     factor
# Minimum     HEATH 1993/06/21      CT      1        I
# Maximum     TUSSOCK 1993/08/18     NP      4        T
# Mean        SEDGE 1993/07/21     GH      2        T
# Unique (excl. NA) 3           11      3      4        2
# Missing values 0           0      0      0        0
# Sorted      FALSE      TRUE  FALSE FALSE FALSE
#             CH4      CO2   temp
# Class      numeric integer numeric
# Minimum    -14.7      16    1.98
# Maximum     648     12783   13.6
# Mean        44.7     1627    5.66
# Unique (excl. NA) 322     311    54
# Missing values 0      0    206
# Sorted      FALSE FALSE FALSE
```

```
stripchart(CO2 ~ tmt, data = gas, vertical = TRUE)
```



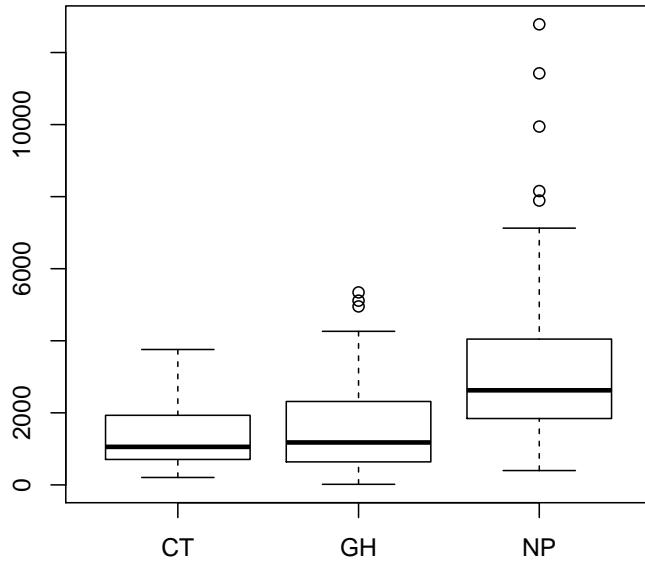
But in cases with a lot of observations (as in this case—see `dfsumm` output above), you should use the `jitter` or `stack` methods.

```
stripchart(CO2 ~ tmt, data = gas, method = "jitter", vertical = TRUE)
```



The `boxplot` function is an alternative, but it is not a good choice for small datasets.

```
boxplot(CO2 ~ tmt, data = gas)
```



By default, the heavy line shows the median, the box shows the 25<sup>th</sup> and 75<sup>th</sup> percentiles, the “whiskers” show the extreme values, and points show any outliers beyond these<sup>92</sup>.

Beanplots may be a more informative alternative to boxplots.

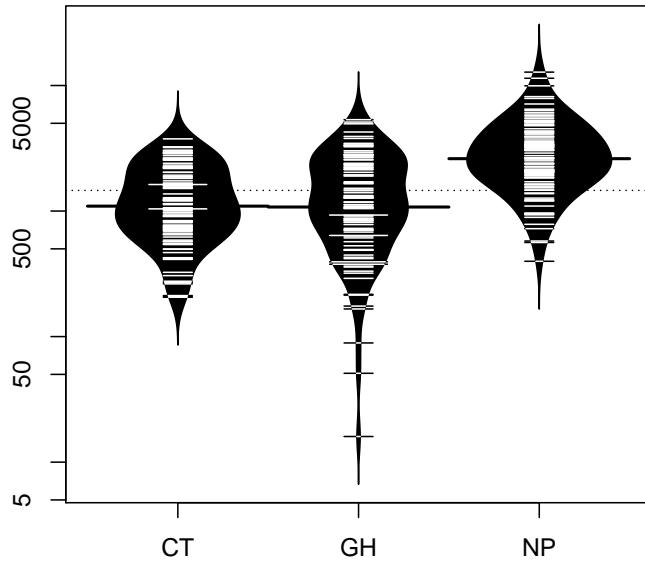
```
install.packages("beanplot")
```

```
library(beanplot)
beanplot(CO2 ~ tmt, data = gas)

# log="y" selected
```

---

<sup>92</sup> The help file for `boxplot.stats` provides additional information.

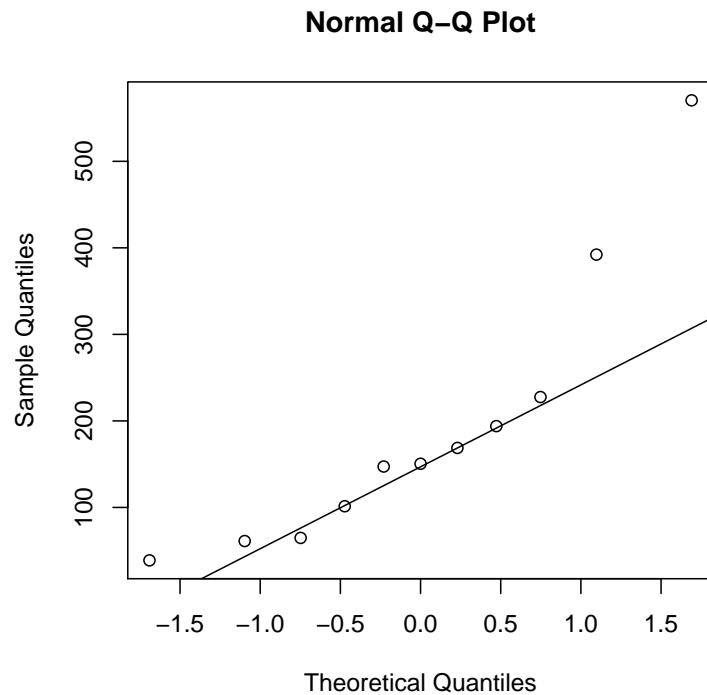


Note the use of the tilde symbol ( $\sim$ ) in the above commands. The code  $y \sim a$  is analogous to a model formula in this case, and simply indicates that  $y$  is described by  $a$  and should be split up based on the value of this variable. We will see more of this character with the specification of statistical models.

### 13.4 Normal quantile and cumulative probability plots

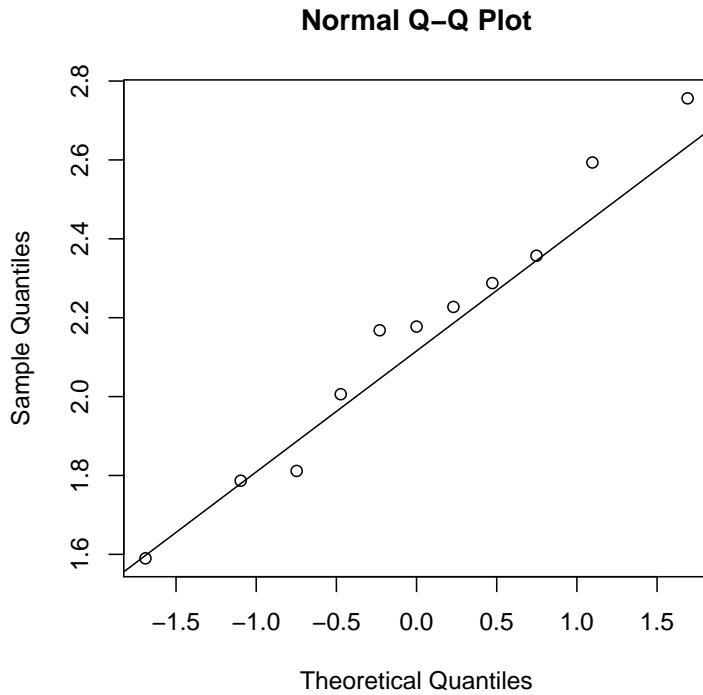
One way to assess the normality of the distribution of a given variable is with a quantile-quantile plot. This plot shows data values vs. quantiles based on a normal distribution (i.e. a  $z$  distribution).

```
qqnorm(cu$ec50)
qqline(cu$ec50)
```



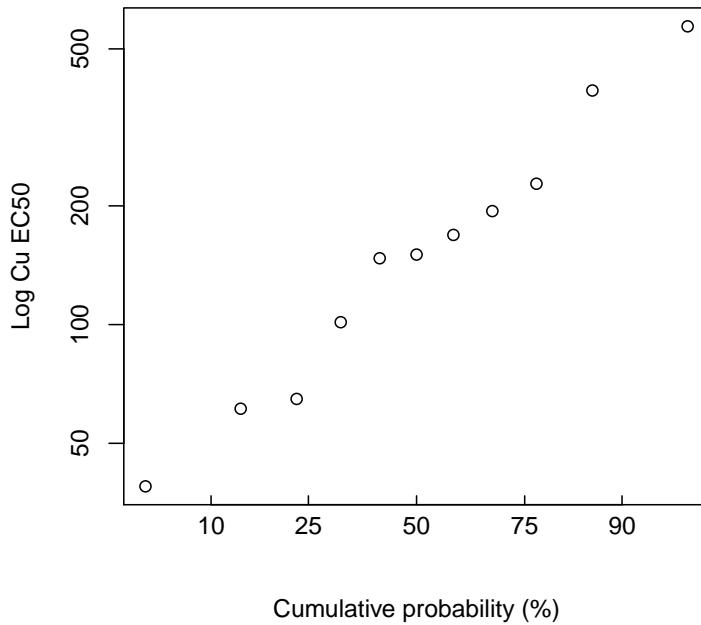
There seems to be some deviation from normality here. A common distribution for toxicity data is log-normal—let's see if this distribution works any better.

```
cu$l.ec50 <- log10(cu$ec50)
qqnorm(cu$l.ec50)
qqline(cu$l.ec50)
```



While R does not have a cumulative probability function in its base packages, it is available in at least one package on CRAN, and it is easy to write your own function. We can make a simple cumulative probability plot with two lines of code.

```
x <- sort(cu$ec50)
plot(qnorm(ppoints(x)), x, log = "y", xaxt = "n", xlab = "Cumulative probability (%)",
      ylab = "Log Cu EC50")
axis(1, qnorm(c(0.1, 0.25, 0.5, 0.75, 0.9)), labels = c(10, 25, 50, 75, 90), las = 1,
      tcl = 0.3, mgp = c(0, 0.2, 0))
```



R will return the quantiles of a given data set with the `quantile` function. Note that there are nine different algorithms available for doing this—you can find descriptions in the help file for `quantile`.

```
quantile(cu$l.ec50)

#      0%      25%      50%      75%      100%
# 1.589950 1.908806 2.177536 2.322487 2.756256

quantile(cu$l.ec50, 0.05)

#      5%
# 1.688351
```

## 14 Basic data manipulation

### 14.1 Indexing and subsetting

Indexing and subsetting are ways to extract specific parts of a data structure (such as specific rows within a data frame) within R. Indexing (also known as subscripting) can also be used to change or add elements to a data structure, and it is one of the features that make R an efficient programming language. Indexing is generally done using square brackets<sup>93</sup>.

```
x <- c(5, 1, 12, 8)
```

Say we want the 3rd observation:

```
x[3]
```

```
# [1] 12
```

R is very flexible in terms of what can be selected or excluded.

For the 1<sup>st</sup> through 3<sup>rd</sup> observation:

```
x[1:3]
```

```
# [1] 5 1 12
```

While this returns all but the 4<sup>th</sup> observation:

```
x[-4]
```

```
# [1] 5 1 12
```

But you cannot mix positive and negative index elements.

So far all these examples have used integer vectors for indexing. There are other options: non-integer numeric vectors can be used, and will be rounded toward zero; factors can be used, and are converted to numeric values first; logical vectors work; and so do character data. For example,

```
i <- c(TRUE, FALSE, FALSE, FALSE)
x[i]
```

```
# [1] 5
```

You typically wouldn't manually create a logical vector for use in indexing though. A more common approach is to use a relational expression. For example, if we want only those observations that are less than 5<sup>94</sup>:

<sup>93</sup> Remember that to use ? to find help files for [] and other operators, you need to surround them with quotes: ?"[".

<sup>94</sup> Note that x is repeated in the command. R does not assume that the vector used as the indexes within the brackets is in any way related to the vector to the left of the brackets. So this, for example, does *not* work: x[<5].

```
x[x<5]
```

```
# [1] 1
```

This may seem a bit confusing, but if we evaluate each piece separately, it becomes more clear:

```
x<5
```

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

```
x[c(FALSE, TRUE, TRUE, FALSE)]
```

```
# [1] 1 12
```

If the elements within an object are named, the names can also be used for indexing.

```
names(x) <- c("a", "b", "c", "d")
```

```
x
```

```
# a b c d  
# 5 1 12 8
```

```
x[c("b", "d")]
```

```
# b d  
# 1 8
```

So far we have just had R return certain elements within the `x` object. But indexing can also be used to change the value of elements.

```
x
```

```
# a b c d  
# 5 1 12 8
```

```
x[2] <- 100
```

```
x
```

```
# a b c d  
# 5 100 12 8
```

And, unlike some other programming languages, the size of a vector in R is not limited by its initial assignment. This is true for other data structures as well<sup>95</sup>. To increase the size of a vector, just assign a value to a position that doesn't currently exist.

<sup>95</sup> But note changing the size of an object may be slower (in terms of evaluation) than adding data to an existing location within an object.

```

x <- c(5, 1, 3, 8)
x

# [1] 5 1 3 8

x[8] <- 10
x

# [1] 5 1 3 8 NA NA NA 10

```

Indexing can be applied to other data structures in a similar manner as shown above. For data frames and matrices, however, we are now working with two dimensions. In specifying indices, row numbers are given first. To demonstrate indexing as applied to data frames, let's read in a file:

```

flow <- read.csv("../data/flow_2006_summary.csv")
dim(flow)

# [1] 24 3

head(flow)

#   site.id month discharge
# 1 1509000 Jan 37237.615
# 2 1509000 Feb 24522.010
# 3 1509000 Mar 27158.420
# 4 1509000 Apr 16505.757
# 5 1509000 May 9830.955
# 6 1509000 Jun 27116.135

```

Let's say we want only the first five rows and first two columns<sup>96</sup>:

```
flow[1:5, 1:2]
```

```
#   site.id month
# 1 1509000 Jan
# 2 1509000 Feb
# 3 1509000 Mar
# 4 1509000 Apr
# 5 1509000 May
```

If an index is left out, R returns all values in that dimension. You need to include the comma though<sup>97</sup>.

---

<sup>96</sup> Note that the row labels (1. . 5 printed to the left of the data) and the row positions are not always identical. This is obvious when merging data. If you need to refer to a row label that is a number in an indexing expression, surround it in quotes to make it character data.

<sup>97</sup> For dataframes, if you provide just one index vector, R will return a set of columns.

```

flow[1:5, ]

#   site.id month discharge
# 1 1509000   Jan 37237.615
# 2 1509000   Feb 24522.010
# 3 1509000   Mar 27158.420
# 4 1509000   Apr 16505.757
# 5 1509000   May  9830.955

```

You can also specify row or column names directly within the brackets—this can be handy when column order may change in future versions of your code.

```

flow[1:5, "site.id"]

# [1] 1509000 1509000 1509000 1509000 1509000

```

And, you can specify multiple row or column names using the `c` function to make a character vector<sup>98</sup>.

```

flow[1:5, c("month", "discharge")]

#   month discharge
# 1   Jan 37237.615
# 2   Feb 24522.010
# 3   Mar 27158.420
# 4   Apr 16505.757
# 5   May  9830.955

```

Relational expressions (which produce logical vectors) can also be used for indexing data frames. Let's pull out only those rows with discharge below, say, the 25<sup>th</sup> percentile.

```

quantile(flow$discharge, 0.25, na.rm = TRUE)

#      25%
# 13105.9

flow[flow$discharge < 13105.9, ]

#   site.id month discharge
# 5 1509000   May  9830.955
# 8 1509000   Aug 12425.920
# 9 1509000   Sep  7459.027
# 15 4232730   Mar  7917.205
# 17 4232730   May 10143.942
# 20 4232730   Aug 11300.953
# NA     NA <NA>       NA

```

<sup>98</sup> Since you are using character data to identify columns in the last two examples (both `"site"` and `c("agency", "site")` are character vectors), it is also possible to use the function `paste`. This could be handy, for example, when you are working in a loop, and want to select a different row or column with each iteration.

Notice that R cannot make a determination for one row, and so returns a row of NA values. While indexing can clearly be used to create a subset of data that meet certain criteria, the **subset** function is often easier and shorter to use for data frames<sup>99</sup>. Subsetting is used to select a subset of a vector, data frame, or matrix that meets a certain criterion (or criteria). To return what was given in the last example<sup>100</sup>.

```
subset(flow, discharge<13105.9)

#   site.id month discharge
# 5  1509000   May  9830.955
# 8  1509000   Aug  12425.920
# 9  1509000   Sep  7459.027
# 15 4232730   Mar  7917.205
# 17 4232730   May 10143.942
# 20 4232730   Aug 11300.953
```

Note that the \$ notation does not need to be used in the **subset** function. As with indexing, multiple constraints can also be used:

```
subset(flow, discharge>13105.9 & site.id == 4232730)

#   site.id month discharge
# 13 4232730   Jan  60388.11
# 14 4232730   Feb  39808.99
# 16 4232730   Apr  13785.88
# 18 4232730   Jun  31553.91
# 19 4232730   Jul  24587.14
# 22 4232730   Oct  26051.04
# 23 4232730   Nov  49824.33
# 24 4232730   Dec  36907.15
```

In some cases you may want to select observations that include any one value out of a set of possibilities. Say we only want those observations for May, June, and July. We could use this:

```
subset(flow, month == "May" | month == "Jun" | month == "Jul")

#   site.id month discharge
# 5  1509000   May  9830.955
# 6  1509000   Jun  27116.135
# 7  1509000   Jul  26460.672
# 17 4232730   May 10143.942
# 18 4232730   Jun  31553.909
# 19 4232730   Jul  24587.140
```

But, this is an easier way:

---

<sup>99</sup> A note about the **subset** function that may be useful: by default, a “subsetted” object retains all the levels of a factor. If you want to remove unused levels, see **droplevels**.

<sup>100</sup> Well, the result is almost identical to the previous result—note the different handling of NA values.

```

subset(flow, month %in% c("May", "Jun", "Jul"))

#   site.id month discharge
# 5  1509000  May  9830.955
# 6  1509000  Jun  27116.135
# 7  1509000  Jul  26460.672
# 17 4232730  May 10143.942
# 18 4232730  Jun 31553.909
# 19 4232730  Jul 24587.140

```

When only a single index is given with no comma, R will return individual columns from the data frame<sup>101</sup>.

By default, R will drop dimensions that are no longer needed in the results from an indexing operation. You can override this by specifying `drop = FALSE` within the brackets<sup>102</sup>.

```

head(flow)

#   site.id month discharge
# 1 1509000  Jan 37237.615
# 2 1509000  Feb 24522.010
# 3 1509000  Mar 27158.420
# 4 1509000  Apr 16505.757
# 5 1509000  May  9830.955
# 6 1509000  Jun  27116.135

flow[, "month"]

# [1] Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec Jan Feb Mar Apr
# [17] May Jun Jul Aug Sep Oct Nov Dec
# Levels: Apr Aug Dec Feb Jan Jul Jun Mar May Nov Oct Sep

flow[, "month", drop = FALSE]

#   month
# 1  Jan
# 2  Feb
# 3  Mar
# 4  Apr
# 5  May
# 6  Jun
# 7  Jul
# 8  Aug
# 9  Sep
# 10 Oct
# 11 Nov

```

---

<sup>101</sup> The data in data frames are actually stored as a list, so this result makes some sense.

<sup>102</sup> This can be handy for handling indexing within functions, when an index may or may not be of length one, but consistent behavior is needed.

```

# 12 Dec
# 13 Jan
# 14 Feb
# 15 Mar
# 16 Apr
# 17 May
# 18 Jun
# 19 Jul
# 20 Aug
# 21 Sep
# 22 Oct
# 23 Nov
# 24 Dec

class(flow[, "month", drop = FALSE])

# [1] "data.frame"

```

At the risk of creating confusion, I will also describe a tidyverse approach to creating subsets of data frames. There are two functions in the dplyr package that can replace most or all of the extracting that you might otherwise do with indexing and the `subset` function. They do not handle replacement, which is a powerful feature of indexing described below.

If you simply load the tidyverse package, dplyr and all the other tidyverse packages will be loaded.

```
library(tidyverse)
```

Here are the examples from above, solved using the `filter` function.

```

filter(flow, discharge < 13105.9)

#   site.id month discharge
# 1 1509000   May  9830.955
# 2 1509000   Aug  12425.920
# 3 1509000   Sep  7459.027
# 4 4232730   Mar  7917.205
# 5 4232730   May  10143.942
# 6 4232730   Aug  11300.953

```

With multiple constraints, you can use a comma instead of the amersand &.

```
filter(flow, discharge>13105.9, site.id == 4232730)
```

```

#   site.id month discharge
# 1 4232730   Jan  60388.11
# 2 4232730   Feb  39808.99
# 3 4232730   Apr  13785.88
# 4 4232730   Jun  31553.91
# 5 4232730   Jul  24587.14

```

```

# 6 4232730 Oct 26051.04
# 7 4232730 Nov 49824.33
# 8 4232730 Dec 36907.15

filter(flow, discharge>13105.9 & site.id == 4232730)

#   site.id month discharge
# 1 4232730 Jan 60388.11
# 2 4232730 Feb 39808.99
# 3 4232730 Apr 13785.88
# 4 4232730 Jun 31553.91
# 5 4232730 Jul 24587.14
# 6 4232730 Oct 26051.04
# 7 4232730 Nov 49824.33
# 8 4232730 Dec 36907.15

```

```

filter(flow, month %in% c("May", "Jun", "Jul"))

#   site.id month discharge
# 1 1509000 May 9830.955
# 2 1509000 Jun 27116.135
# 3 1509000 Jul 26460.672
# 4 4232730 May 10143.942
# 5 4232730 Jun 31553.909
# 6 4232730 Jul 24587.140

```

For selection of rows by integer index, use the `slice` function.

So to do this:

```

flow[1:5, ]

#   site.id month discharge
# 1 1509000 Jan 37237.615
# 2 1509000 Feb 24522.010
# 3 1509000 Mar 27158.420
# 4 1509000 Apr 16505.757
# 5 1509000 May 9830.955

```

with `slice`, use this:

```

slice(flow, 1:5)

# # A tibble: 5 x 3
#   site.id month   discharge
#       <int> <fct>     <dbl>
# 1 1509000 Jan      37238
# 2 1509000 Feb      24522

```

```
# 3 1509000 Mar      27158
# 4 1509000 Apr      16506
# 5 1509000 May      9831
```

This function does have some handy features. For example, if you wanted the last row.

```
slice(flow, n())

# # A tibble: 1 x 3
#   site.id month discharge
#       <int> <fct>     <dbl>
# 1 4232730 Dec        36907
```

Or, how about the last 5 rows?

```
slice(flow, n() - 0:5)

# # A tibble: 6 x 3
#   site.id month discharge
#       <int> <fct>     <dbl>
# 1 4232730 Dec        36907
# 2 4232730 Nov        49824
# 3 4232730 Oct        26051
# 4 4232730 Sep         NA
# 5 4232730 Aug        11301
# 6 4232730 Jul        24587
```

These functions also work on grouped data, which will be described below.

The `slice` and `filter` functions differ from indexing the `[ ]` in an important way: they only work on rows. To select columns, you need to use the `select` function. Let's look at a new data frame with more columns, and more interesting names.

```
wind <- read.csv("../data/ave_wind_us.csv")
head(wind)

#           location no.yr jan feb mar apr may jun jul aug sep
# 1 13876BIRMINGHAM AP_AL    65  8.1  8.7  9.0  8.2  6.8  6.0  5.7  5.4  6.3
# 2 03856HUNTSVILLE_AL     41  9.0  9.4  9.7  9.2  7.9  6.8  5.9  5.8  6.7
# 3 13894MOBILE_AL        60 10.1 10.3 10.5 10.1 8.7  7.5  6.9  6.7  7.7
# 4 13895MONTGOMERY_AL    64  7.7  8.2  8.3  7.3  6.1  5.8  5.7  5.2  5.9
# 5 26451ANCHORAGE_AK     55  6.4  6.8  7.1  7.3  8.5  8.4  7.3  6.9  6.7
# 6 25308ANNETTE_AK       44 11.2 11.2 10.4 10.2 8.6  8.3  7.5  7.6  8.3
#   oct nov dec ann
# 1 6.2  7.2  7.7  7.1
# 2 7.2  8.0  8.9  7.9
# 3 8.0  8.9  9.6  8.8
# 4 5.7  6.5  7.1  6.6
# 5 6.7  6.4  6.3  7.1
# 6 10.8 11.2 11.6 9.8
```

```

dfsumm(wind)

#
# 275 rows and 15 columns

# Error in paste(location = structure(c(56L, 9L, 65L, 66L, 219L, 207L, 227L, : formal
argument "sep" matched by multiple actual arguments

```

There is a bug in `dfsumm`

```

summary(wind)

#           location      no.yr       jan
# 03017DENVER_ CO   : 1  Min.   : 3.0  Min.   : 2.900
# 03103FLAGSTAFF_ AZ: 1  1st Qu.:44.5  1st Qu.: 7.800
# 03812ASHEVILLE_ NC: 1 Median  :57.0  Median : 9.500
# 03813MACON_ GA   : 1 Mean    :52.2  Mean   : 9.662
# 03816PADUCAH KY  : 1 3rd Qu.:61.5  3rd Qu.:11.300
# 03820AUGUSTA_GA : 1 Max.    :81.0  Max.   :46.200
# (Other)          :269

#           feb      mar      apr      may
# Min.   : 3.900  Min.   : 5.00  Min.   : 4.30  Min.   : 4.200
# 1st Qu.: 8.200  1st Qu.: 8.60  1st Qu.: 8.50  1st Qu.: 7.900
# Median : 9.600  Median :10.10  Median :10.20  Median : 9.300
# Mean   : 9.849  Mean   :10.35  Mean   :10.32  Mean   : 9.451
# 3rd Qu.:11.200  3rd Qu.:11.70  3rd Qu.:11.70  3rd Qu.:10.700
# Max.   :44.400  Max.   :41.40  Max.   :35.90  Max.   :29.500
#
#           jun      jul      aug      sep
# Min.   : 3.900  Min.   : 3.600  Min.   : 3.3  Min.   : 3.200
# 1st Qu.: 7.200  1st Qu.: 6.700  1st Qu.: 6.4  1st Qu.: 6.650
# Median : 8.500  Median : 7.900  Median : 7.5  Median : 8.000
# Mean   : 8.835  Mean   : 8.196  Mean   : 7.9  Mean   : 8.227
# 3rd Qu.:10.100  3rd Qu.: 9.300  3rd Qu.: 9.1  3rd Qu.: 9.400
# Max.   :27.400  Max.   :25.100  Max.   :24.7  Max.   :28.800
#
#           oct      nov      dec      ann
# Min.   : 3.200  Min.   : 3.600  Min.   : 3.000  Min.   : 4.100
# 1st Qu.: 6.700  1st Qu.: 7.300  1st Qu.: 7.600  1st Qu.: 7.500
# Median : 8.400  Median : 9.000  Median : 9.300  Median : 8.900
# Mean   : 8.608  Mean   : 9.215  Mean   : 9.377  Mean   : 9.176
# 3rd Qu.:10.000  3rd Qu.:10.750  3rd Qu.:10.900  3rd Qu.:10.450
# Max.   :33.800  Max.   :39.500  Max.   :44.700  Max.   :35.100
#

```

To keep the output small, let's take the first 4 rows.

```

wind <- wind[1:5, ]

```

Here, the power of `select`, and differences from base R syntax both become clear. Let's work through some examples.

Select a single column.

```
dplyr::select(wind, jan)
```

```
#      jan
# 1  8.1
# 2  9.0
# 3 10.1
# 4  7.7
# 5  6.4
```

Select all columns except for one.

```
dplyr::select(wind, -ann)
```

```
#           location no.yr  jan  feb  mar  apr  may  jun  jul  aug  sep
# 1 13876BIRMINGHAM AP_AL    65  8.1  8.7  9.0  8.2  6.8  6.0  5.7  5.4  6.3
# 2 03856HUNTSVILLE_ AL     41  9.0  9.4  9.7  9.2  7.9  6.8  5.9  5.8  6.7
# 3 13894MOBILE_ AL       60 10.1 10.3 10.5 10.1 8.7  7.5  6.9  6.7  7.7
# 4 13895MONTGOMERY_ AL    64  7.7  8.2  8.3  7.3  6.1  5.8  5.7  5.2  5.9
# 5 26451ANCHORAGE_ AK     55  6.4  6.8  7.1  7.3  8.5  8.4  7.3  6.9  6.7
#   oct nov dec
# 1 6.2 7.2 7.7
# 2 7.2 8.0 8.9
# 3 8.0 8.9 9.6
# 4 5.7 6.5 7.1
# 5 6.7 6.4 6.3
```

Return a range of columns.

```
dplyr::select(wind, jan:mar)
```

```
#      jan  feb  mar
# 1  8.1  8.7  9.0
# 2  9.0  9.4  9.7
# 3 10.1 10.3 10.5
# 4  7.7  8.2  8.3
# 5  6.4  6.8  7.1
```

Combine this range with a single column.

```
dplyr::select(wind, location, jan:mar)
```

```
#           location  jan  feb  mar
# 1 13876BIRMINGHAM AP_AL  8.1  8.7  9.0
# 2 03856HUNTSVILLE_ AL  9.0  9.4  9.7
# 3 13894MOBILE_ AL 10.1 10.3 10.5
# 4 13895MONTGOMERY_ AL  7.7  8.2  8.3
# 5 26451ANCHORAGE_ AK  6.4  6.8  7.1
```

Select all columns that start with the letter “j”.

```
dplyr::select(wind, starts_with("j"))

#   jan jun jul
# 1 8.1 6.0 5.7
# 2 9.0 6.8 5.9
# 3 10.1 7.5 6.9
# 4 7.7 5.8 5.7
# 5 6.4 8.4 7.3
```

Or all columns that contain the letter “a”.

```
select(wind, contains("a"))

#           location jan mar apr may aug ann
# 1 13876BIRMINGHAM_AP_AL 8.1 9.0 8.2 6.8 5.4 7.1
# 2 03856HUNTSVILLE_AL 9.0 9.7 9.2 7.9 5.8 7.9
# 3 13894MOBILE_AL 10.1 10.5 10.1 8.7 6.7 8.8
# 4 13895MONTGOMERY_AL 7.7 8.3 7.3 6.1 5.2 6.6
# 5 26451ANCHORAGE_AK 6.4 7.1 7.3 8.5 6.9 7.1
```

These last few operations would be much more difficult with only base R.

Indexing of matrices and arrays is similar to vectors and data frames. For example:

```
m <- matrix(rnorm(25), nrow = 5)
m

#          [,1]      [,2]      [,3]      [,4]      [,5]
# [1,] -0.3573581  0.03757808 -0.6789215 -0.7268932  1.39540170
# [2,]  0.4809392 -1.30056110  0.4219053 -0.2901432 -0.02293646
# [3,]  1.5393675 -0.21412196 -1.2293125  0.5341867 -1.60232709
# [4,]  0.3162443 -0.02763268  1.1956100  1.9809221  1.56459513
# [5,]  1.0258346 -2.57529097  0.7192867  0.5568139 -0.52047123

m[3, 2]

# [1] -0.214122

m[, 2]

# [1]  0.03757808 -1.30056110 -0.21412196 -0.02763268 -2.57529097
```

Let’s come up with some numbers that have a pattern, to make these examples a bit more clear.

```
m <- matrix(round(5*(sin(1:25/5*2*pi-2.5)+1:25/10), 0), nrow = 5)
```

If our object has row and column names, we can use those for indexing<sup>103</sup>.

```
dimnames(m) <- list(y = 1:5, x = c("a", "b", "c", "d", "e"))
m["5", ]
```

```
#   a   b   c   d   e
# 0  2  5  7 10
```

Of course, any of the indexing examples covered so far could be used to change values.

```
m[4, 1] <- 0
```

```
m
```

```
#      x
# y   a   b   c   d   e
# 1 -4 -2  1  3  6
# 2  1  4  6  9 11
# 3  6  9 11 14 16
# 4  0  7 10 12 15
# 5  0  2  5  7 10
```

We could change an entire row too.

```
m[4, ] <- 0
```

```
m
```

```
#      x
# y   a   b   c   d   e
# 1 -4 -2  1  3  6
# 2  1  4  6  9 11
# 3  6  9 11 14 16
# 4  0  0  0  0  0
# 5  0  2  5  7 10
```

Or

```
m[1, ] <- m[2, ] - 10
```

```
m
```

```
#      x
# y   a   b   c   d   e
# 1 -9 -6 -4 -1  1
# 2  1  4  6  9 11
# 3  6  9 11 14 16
# 4  0  0  0  0  0
# 5  0  2  5  7 10
```

<sup>103</sup> We could use the `rownames` and `colnames` functions for this, but `dimnames` is faster here. Note that the row and column names of `v` are character data, even though they might look like numbers. This is *always* the case.

You can use only a single index (and no comma) for a matrix, and R will behave as if you first converted the matrix to a vector.

```
m[1:10]  
  
# [1] -9 1 6 0 0 -6 4 9 0 2
```

Compare that to this.

```
as.vector(m)[1:10]  
  
# [1] -9 1 6 0 0 -6 4 9 0 2
```

This feature is useful for extracting or changing values that meet some criteria. For example, what if we wanted all elements that are less than 10<sup>104</sup>.

```
m[m<10]  
  
# [1] -9 1 6 0 0 -6 4 9 0 2 -4 6 0 5 -1 9 0 7 1 0
```

Or if we wanted to change these values to NA<sup>105</sup>,

```
m[m<10] <- NA  
  
#      x  
# y     a   b   c   d   e  
# 1 NA  NA  NA  NA  NA  
# 2 NA  NA  NA  NA  11  
# 3 NA  NA  11  14  16  
# 4 NA  NA  NA  NA  NA  
# 5 NA  NA  NA  NA  10
```

Here is another trick for working with matrices: using a matrix for the indices. This is the approach to use when you want to return individual cells, and not all the overlap between the first and second indices. An example may be more clear than the explanation. Say we want to extract (or change) cells 3, 4 (14) and 5, 5 (10) from m. This doesn't work:

```
m[c(3, 5), c(4, 5)]  
  
#      x  
# y     d   e  
# 3 14 16  
# 5 NA 10
```

<sup>104</sup> We are actually using a logical matrix for indexing here.

<sup>105</sup> To just get the locations of these values, use `which`. If you want the indices for all dimensions, set the `arr.ind` argument to `TRUE`, e.g., `which(m<10, arr.ind = TRUE)`. The result will be a matrix of indices.

Instead, set up a two-column matrix with pairs of indices given in each row.

```
i <- matrix(c(3, 5, 4, 5), nrow = 2)
i

#      [,1] [,2]
# [1,]    3    4
# [2,]    5    5

m[i]

# [1] 14 10

m[i] <- -99
m

#      x
# y   a   b   c   d   e
# 1 NA NA NA NA NA
# 2 NA NA NA NA 11
# 3 NA NA 11 -99 16
# 4 NA NA NA NA NA
# 5 NA NA NA NA -99
```

Indexing arrays follows indexing matrices. Let's work with another object in the `Datasets` package: `UCBAdmissions`, which has admissions data for UC Berkeley departments<sup>106</sup>.

```
ad <- UCBAdmissions
ad

# , , Dept = A
#
#          Gender
# Admit     Male Female
#   Admitted 512     89
#   Rejected 313     19
#
# , , Dept = B
#
#          Gender
# Admit     Male Female
#   Admitted 353     17
#   Rejected 207      8
#
# , , Dept = C
#
#          Gender
```

---

<sup>106</sup> This object actually has the class `table`, which means it is an array of integers. But it is an array after all, and works for us here.

```

# Admit      Male Female
#   Admitted 120    202
#   Rejected 205    391
#
# , , Dept = D
#
#           Gender
# Admit      Male Female
#   Admitted 138    131
#   Rejected 279    244
#
# , , Dept = E
#
#           Gender
# Admit      Male Female
#   Admitted 53     94
#   Rejected 138    299
#
# , , Dept = F
#
#           Gender
# Admit      Male Female
#   Admitted 22     24
#   Rejected 351    317

```

`dim(ad)`

`# [1] 2 2 6`

`dimnames(ad)`

```

# $Admit
# [1] "Admitted" "Rejected"
#
# $Gender
# [1] "Male"     "Female"
#
# $Dept
# [1] "A" "B" "C" "D" "E" "F"

```

So if we wanted to extract the first three departments,

```

ad <- ad[, ,1:3]
ad

# , , Dept = A
#
#           Gender
# Admit      Male Female
#   Admitted 512    89

```

```

#   Rejected  313     19
#
# , , Dept = B
#
#           Gender
# Admit      Male Female
#   Admitted  353     17
#   Rejected  207     8
#
# , , Dept = C
#
#           Gender
# Admit      Male Female
#   Admitted  120     202
#   Rejected  205     391

```

Let's take a look at admissions for females only, for all departments.

```

ad[, "Female", ]

#
#           Dept
# Admit      A   B   C
#   Admitted 89  17 202
#   Rejected 19   8 391

```

Or, how about all admissions data for department A only.

```

ad[, , "A"]

#
#           Gender
# Admit      Male Female
#   Admitted 512    89
#   Rejected 313    19

```

Notice how the structure of the extracted data depends on the dimensions you request. If possible, dimensions are dropped by default<sup>107</sup>

And, of course, we could use any of the above expressions to change values.

```

ad[, "Male", "A"] <- NA
ad

#
# , , Dept = A
#
#           Gender
# Admit      Male Female
#   Admitted          89

```

---

<sup>107</sup> So the result from the last example is actually a matrix. Again, to prevent dropping dimensions, add `drop = TRUE`.

```

#   Rejected      19
#
# , , Dept = B
#
#           Gender
# Admit     Male Female
#   Admitted 353    17
#   Rejected 207     8
#
# , , Dept = C
#
#           Gender
# Admit     Male Female
#   Admitted 120    202
#   Rejected 205    391

```

The approaches used above with matrices and relational expressions for indices also apply to arrays (matrices are actually just two-dimensional arrays in R).

Indexing is a little different for lists—it is generally best to use double square brackets, `[[i]]`, to extract an element from a list.

```

l <- list("A", "B", "C")
l[[1]]

# [1] "A"

```

If the list has named elements, you can use the `$` notation or brackets.

Single brackets can actually be used as well. The difference from double brackets is that `[ ]` returns a list, while `[[ ]]` returns an object with the class of the element within the list that was selected.

Although you may not run into a need for it, it is possible to use double, triple, etc. indexing with all types of data structures. R evaluates the expression from left to right.

## 14.2 Sorting data and locating observations

It is often necessary to sort data. For a single vector, this is done with the function `sort`.

```

x <- rnorm(5)
x

# [1] 1.3195684505 -0.0598917056 -0.2129669317 -1.1916022693
# [5] 0.0002471681

sort(x)

# [1] -1.1916022693 -0.2129669317 -0.0598917056 0.0002471681
# [5] 1.3195684505

```

But what if you want to sort an entire data frame by one column? In this case use the function `order`, in combination with indexing.

```
flow[order(flow$discharge), ]
```

```
#   site.id month discharge
# 9  1509000   Sep  7459.027
# 15 4232730   Mar  7917.205
# 5  1509000   May  9830.955
# 17 4232730   May 10143.942
# 20 4232730   Aug 11300.953
# 8  1509000   Aug 12425.920
# 16 4232730   Apr 13785.879
# 12 1509000   Dec 14157.208
# 4  1509000   Apr 16505.757
# 10 1509000   Oct 23286.085
# 2  1509000   Feb 24522.010
# 19 4232730   Jul 24587.140
# 22 4232730   Oct 26051.044
# 7  1509000   Jul 26460.672
# 6  1509000   Jun 27116.135
# 3  1509000   Mar 27158.420
# 11 1509000   Nov 29718.360
# 18 4232730   Jun 31553.909
# 24 4232730   Dec 36907.146
# 1  1509000   Jan 37237.615
# 14 4232730   Feb 39808.994
# 23 4232730   Nov 49824.330
# 13 4232730   Jan 60388.111
# 21 4232730   Sep       NA
```

The function `order` returns a vector that contains the row positions of the ranked data:

```
order(flow$discharge)
```

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

Notice that in the reordered data frame, the automatic row labels that are added by R stay with the observation that they were originally associated with when the data frame was first created.

The dplyr function `arrange()` provides an easier alternative to the use of `order()`. Both examples below do the same thing—sort the data frame by date of birth and then by name.

```
pres19[order(pres19$birth, pres19$name), ]
```

```
#   order          name birth death      party
# 28    28 Woodrow Wilson 1856 1924 Democrat
# 27    27 William Taft  1857 1930 Republican
# 26    26 Theodore Roosevelt 1858 1919 Republican
# 29    29 Warren Harding 1865 1923 Republican
```

# 30	30	Calvin Coolidge	1872	1933	Republican
# 31	31	Herbert C. Hoover	1874	1964	Republican
# 32	32	Franklin Delano Roosevelt	1882	1945	Democrat
# 33	33	Harry S Truman	1884	1972	Democrat
# 34	34	Dwight David Eisenhower	1890	1969	Republican
# 36	36	Lyndon Baines Johnson	1908	1973	Democrat
# 40	40	Ronald Wilson Reagan	1911	2004	Republican
# 38	38	Gerald R. Ford	1913	2006	Republican
# 37	37	Richard Milhous Nixon	1913	1994	Republican
# 35	35	John Fitzgerald Kennedy	1917	1963	Democrat
# 41	41	George H. W. Bush	1924	NA	Republican
# 39	39	James (Jimmy) Earl Carter	1924	NA	Democrat
# 43	43	George W. Bush	1946	NA	Republican
# 42	42	William (Bill) Jefferson Clinton	1946	NA	Democrat
# 44	44	Barack Obama	1961	NA	Democrat
#	first last			vp	
# 28	1913 1921			Thomas Marshall	
# 27	1909 1913			James Sherman	
# 26	1901 1909			Charles Fairbanks	
# 29	1921 1923			Calvin Coolidge	
# 30	1923 1929			Charles Dawes	
# 31	1929 1933			Charles Curtis	
# 32	1933 1945	John Garner_ Henry Wallace_ Harry S. Truman			
# 33	1945 1953			Alben Barkley	
# 34	1953 1961			Richard Milhous Nixon	
# 36	1963 1969			Hubert Humphrey	
# 40	1981 1989			George H. W. Bush	
# 38	1974 1977			Nelson Rockefeller	
# 37	1969 1974			Spiro Agnew_ Gerald R. Ford	
# 35	1961 1963			Lyndon Johnson	
# 41	1989 1993			James Danforth_ Dan Quayle	
# 39	1977 1981			Walter Mondale	
# 43	2001 2009			Richard Cheney	
# 42	1993 2001			Al Gore	
# 44	2009 NA			Joseph Biden	

```
arrange(pres19, birth, name)
```

#	order		name	birth	death	party
# 1	28		Woodrow Wilson	1856	1924	Democrat
# 2	27		William Taft	1857	1930	Republican
# 3	26		Theodore Roosevelt	1858	1919	Republican
# 4	29		Warren Harding	1865	1923	Republican
# 5	30		Calvin Coolidge	1872	1933	Republican
# 6	31		Herbert C. Hoover	1874	1964	Republican
# 7	32	Franklin Delano Roosevelt	1882	1945	Democrat	
# 8	33		Harry S Truman	1884	1972	Democrat
# 9	34		Dwight David Eisenhower	1890	1969	Republican
# 10	36		Lyndon Baines Johnson	1908	1973	Democrat
# 11	40		Ronald Wilson Reagan	1911	2004	Republican
# 12	38		Gerald R. Ford	1913	2006	Republican
# 13	37		Richard Milhous Nixon	1913	1994	Republican
# 14	35		John Fitzgerald Kennedy	1917	1963	Democrat

# 15	41	George H. W. Bush	1924	NA	Republican
# 16	39	James (Jimmy) Earl Carter	1924	NA	Democrat
# 17	43	George W. Bush	1946	NA	Republican
# 18	42	William (Bill) Jefferson Clinton	1946	NA	Democrat
# 19	44	Barack Obama	1961	NA	Democrat
#	first last			vp	
# 1	1913 1921	Thomas Marshall			
# 2	1909 1913	James Sherman			
# 3	1901 1909	Charles Fairbanks			
# 4	1921 1923	Calvin Coolidge			
# 5	1923 1929	Charles Dawes			
# 6	1929 1933	Charles Curtis			
# 7	1933 1945	John Garner_ Henry Wallace_ Harry S. Truman			
# 8	1945 1953	Alben Barkley			
# 9	1953 1961	Richard Milhous Nixon			
# 10	1963 1969	Hubert Humphrey			
# 11	1981 1989	George H. W. Bush			
# 12	1974 1977	Nelson Rockefeller			
# 13	1969 1974	Spiro Agnew_ Gerald R. Ford			
# 14	1961 1963	Lyndon Johnson			
# 15	1989 1993	James Danforth_ Dan Quayle			
# 16	1977 1981	Walter Mondale			
# 17	2001 2009	Richard Cheney			
# 18	1993 2001	Al Gore			
# 19	2009 NA	Joseph Biden			

And both could be used to sort in descending order.

```
pres19[order(pres19$birth, pres19$name, decreasing = TRUE), ]
```

#	order	name	birth	death	party
# 44	44	Barack Obama	1961	NA	Democrat
# 42	42	William (Bill) Jefferson Clinton	1946	NA	Democrat
# 43	43	George W. Bush	1946	NA	Republican
# 39	39	James (Jimmy) Earl Carter	1924	NA	Democrat
# 41	41	George H. W. Bush	1924	NA	Republican
# 35	35	John Fitzgerald Kennedy	1917	1963	Democrat
# 37	37	Richard Milhous Nixon	1913	1994	Republican
# 38	38	Gerald R. Ford	1913	2006	Republican
# 40	40	Ronald Wilson Reagan	1911	2004	Republican
# 36	36	Lyndon Baines Johnson	1908	1973	Democrat
# 34	34	Dwight David Eisenhower	1890	1969	Republican
# 33	33	Harry S Truman	1884	1972	Democrat
# 32	32	Franklin Delano Roosevelt	1882	1945	Democrat
# 31	31	Herbert C. Hoover	1874	1964	Republican
# 30	30	Calvin Coolidge	1872	1933	Republican
# 29	29	Warren Harding	1865	1923	Republican
# 26	26	Theodore Roosevelt	1858	1919	Republican
# 27	27	William Taft	1857	1930	Republican
# 28	28	Woodrow Wilson	1856	1924	Democrat
#	first last	vp			
# 44	2009 NA	Joseph Biden			

# 42	1993	2001	Al Gore
# 43	2001	2009	Richard Cheney
# 39	1977	1981	Walter Mondale
# 41	1989	1993	James Danforth_ Dan Quayle
# 35	1961	1963	Lyndon Johnson
# 37	1969	1974	Spiro Agnew_ Gerald R. Ford
# 38	1974	1977	Nelson Rockefeller
# 40	1981	1989	George H. W. Bush
# 36	1963	1969	Hubert Humphrey
# 34	1953	1961	Richard Milhous Nixon
# 33	1945	1953	Alben Barkley
# 32	1933	1945	John Garner_ Henry Wallace_ Harry S. Truman
# 31	1929	1933	Charles Curtis
# 30	1923	1929	Charles Dawes
# 29	1921	1923	Calvin Coolidge
# 26	1901	1909	Charles Fairbanks
# 27	1909	1913	James Sherman
# 28	1913	1921	Thomas Marshall

pres19[order(-pres19\$birth, pres19\$name), ]

#	order		name	birth	death	party
# 44	44		Barack Obama	1961	NA	Democrat
# 43	43		George W. Bush	1946	NA	Republican
# 42	42	William (Bill)	Jefferson Clinton	1946	NA	Democrat
# 41	41		George H. W. Bush	1924	NA	Republican
# 39	39		James (Jimmy) Earl Carter	1924	NA	Democrat
# 35	35		John Fitzgerald Kennedy	1917	1963	Democrat
# 38	38		Gerald R. Ford	1913	2006	Republican
# 37	37		Richard Milhous Nixon	1913	1994	Republican
# 40	40		Ronald Wilson Reagan	1911	2004	Republican
# 36	36		Lyndon Baines Johnson	1908	1973	Democrat
# 34	34		Dwight David Eisenhower	1890	1969	Republican
# 33	33		Harry S Truman	1884	1972	Democrat
# 32	32		Franklin Delano Roosevelt	1882	1945	Democrat
# 31	31		Herbert C. Hoover	1874	1964	Republican
# 30	30		Calvin Coolidge	1872	1933	Republican
# 29	29		Warren Harding	1865	1923	Republican
# 26	26		Theodore Roosevelt	1858	1919	Republican
# 27	27		William Taft	1857	1930	Republican
# 28	28		Woodrow Wilson	1856	1924	Democrat
#	first	last			vp	
# 44	2009	NA			Joseph Biden	
# 43	2001	2009			Richard Cheney	
# 42	1993	2001			Al Gore	
# 41	1989	1993			James Danforth_ Dan Quayle	
# 39	1977	1981			Walter Mondale	
# 35	1961	1963			Lyndon Johnson	
# 38	1974	1977			Nelson Rockefeller	
# 37	1969	1974			Spiro Agnew_ Gerald R. Ford	
# 40	1981	1989			George H. W. Bush	
# 36	1963	1969			Hubert Humphrey	
# 34	1953	1961			Richard Milhous Nixon	

```

# 33 1945 1953 Alben Barkley
# 32 1933 1945 John Garner_ Henry Wallace_ Harry S. Truman
# 31 1929 1933 Charles Curtis
# 30 1923 1929 Charles Dawes
# 29 1921 1923 Calvin Coolidge
# 26 1901 1909 Charles Fairbanks
# 27 1909 1913 James Sherman
# 28 1913 1921 Thomas Marshall

```

```
arrange(pres19, desc(birth), desc(name))
```

#	order		name	birth	death	party
# 1	44		Barack Obama	1961	NA	Democrat
# 2	42	William (Bill)	Jefferson Clinton	1946	NA	Democrat
# 3	43		George W. Bush	1946	NA	Republican
# 4	39	James (Jimmy)	Earl Carter	1924	NA	Democrat
# 5	41		George H. W. Bush	1924	NA	Republican
# 6	35	John Fitzgerald	Kennedy	1917	1963	Democrat
# 7	37		Richard Milhous Nixon	1913	1994	Republican
# 8	38		Gerald R. Ford	1913	2006	Republican
# 9	40		Ronald Wilson Reagan	1911	2004	Republican
# 10	36		Lyndon Baines Johnson	1908	1973	Democrat
# 11	34	Dwight David	Eisenhower	1890	1969	Republican
# 12	33		Harry S Truman	1884	1972	Democrat
# 13	32	Franklin Delano	Roosevelt	1882	1945	Democrat
# 14	31		Herbert C. Hoover	1874	1964	Republican
# 15	30		Calvin Coolidge	1872	1933	Republican
# 16	29		Warren Harding	1865	1923	Republican
# 17	26		Theodore Roosevelt	1858	1919	Republican
# 18	27		William Taft	1857	1930	Republican
# 19	28		Woodrow Wilson	1856	1924	Democrat
#	first	last			vp	
# 1	2009	NA			Joseph Biden	
# 2	1993	2001			Al Gore	
# 3	2001	2009			Richard Cheney	
# 4	1977	1981			Walter Mondale	
# 5	1989	1993			James Danforth_ Dan Quayle	
# 6	1961	1963			Lyndon Johnson	
# 7	1969	1974			Spiro Agnew_ Gerald R. Ford	
# 8	1974	1977			Nelson Rockefeller	
# 9	1981	1989			George H. W. Bush	
# 10	1963	1969			Hubert Humphrey	
# 11	1953	1961			Richard Milhous Nixon	
# 12	1945	1953			Alben Barkley	
# 13	1933	1945	John Garner_ Henry Wallace_	Harry S. Truman		
# 14	1929	1933			Charles Curtis	
# 15	1923	1929			Charles Dawes	
# 16	1921	1923			Calvin Coolidge	
# 17	1901	1909			Charles Fairbanks	
# 18	1909	1913			James Sherman	
# 19	1913	1921			Thomas Marshall	

```
arrange(pres19, desc(birth), name)
```

#	order		name	birth	death	party
# 1	44		Barack Obama	1961	NA	Democrat
# 2	43		George W. Bush	1946	NA	Republican
# 3	42	William (Bill)	Jefferson Clinton	1946	NA	Democrat
# 4	41		George H. W. Bush	1924	NA	Republican
# 5	39		James (Jimmy) Earl Carter	1924	NA	Democrat
# 6	35		John Fitzgerald Kennedy	1917	1963	Democrat
# 7	38		Gerald R. Ford	1913	2006	Republican
# 8	37		Richard Milhous Nixon	1913	1994	Republican
# 9	40		Ronald Wilson Reagan	1911	2004	Republican
# 10	36		Lyndon Baines Johnson	1908	1973	Democrat
# 11	34		Dwight David Eisenhower	1890	1969	Republican
# 12	33		Harry S Truman	1884	1972	Democrat
# 13	32	Franklin Delano Roosevelt		1882	1945	Democrat
# 14	31		Herbert C. Hoover	1874	1964	Republican
# 15	30		Calvin Coolidge	1872	1933	Republican
# 16	29		Warren Harding	1865	1923	Republican
# 17	26	Theodore Roosevelt		1858	1919	Republican
# 18	27		William Taft	1857	1930	Republican
# 19	28		Woodrow Wilson	1856	1924	Democrat
#	first	last			vp	
# 1	2009	NA			Joseph Biden	
# 2	2001	2009			Richard Cheney	
# 3	1993	2001			Al Gore	
# 4	1989	1993		James Danforth_	Dan Quayle	
# 5	1977	1981			Walter Mondale	
# 6	1961	1963			Lyndon Johnson	
# 7	1974	1977			Nelson Rockefeller	
# 8	1969	1974	Spiro Agnew_	Gerald R. Ford		
# 9	1981	1989			George H. W. Bush	
# 10	1963	1969			Hubert Humphrey	
# 11	1953	1961		Richard Milhous Nixon		
# 12	1945	1953			Alben Barkley	
# 13	1933	1945	John Garner_	Henry Wallace_	Harry S. Truman	
# 14	1929	1933			Charles Curtis	
# 15	1923	1929			Charles Dawes	
# 16	1921	1923			Calvin Coolidge	
# 17	1901	1909			Charles Fairbanks	
# 18	1909	1913			James Sherman	
# 19	1913	1921			Thomas Marshall	

The previous discussion in this section shows how to isolate data that meet certain criteria from a data structure. But sometimes it is important to know where data reside in the original data structure. Two functions that are handy for locating data within an R data structure are `match` and `which`. The `match` function will tell you where a specific value resides, while the `which` function will return the locations of values that meet certain criteria.

```
match(7917.205, flow$discharge)
```

```
# [1] 15
```

Note that this function matches the first observation only. It is a vectorized function.

```
match(c(7917.205, 37237.615), flow$discharge)

# [1] 15 1
```

Let's check this result:

```
flow$discharge[c(1, 15)]

# [1] 37237.615 7917.205
```

The `match` function is useful for finding the location of the unique values, such as the maximum<sup>108</sup>.

```
match(max(flow$discharge), flow$discharge)

# [1] 21
```

Let's take a look at the value.

```
flow$discharge[21]

# [1] NA
```

Oops. Try again.

```
match(max(na.omit(flow$discharge)), flow$discharge)

# [1] 13
```

The `which` function, on the other hand, will return all locations that meet the criterion or criteria.

```
which(flow$discharge<20000)

# [1] 4 5 8 9 12 15 16 17 20
```

Of course, you can specify multiple constraints.

```
which(flow$discharge<20000 & flow$discharge>10000)

# [1] 4 8 12 16 17 20
```

The `which` function can be useful for locating missing values.

---

<sup>108</sup> Another way to do the same thing is with `which.max`.

```

which(is.na(flow$discharge))

# [1] 21

```

### 14.3 Combining data frames and matrices

Data frames (or vectors or matrices) often need to be combined for analysis or plotting. Two R functions that are very useful for combining data in data frames or matrices are `rbind()`, and `merge()`. The function `rbind()` (“row bind”) simply “stacks” objects on top of each other to make a new object. Its behavior is pretty straightforward for matrices.

```

m <- matrix(1:12, nrow = 3)

m

```

```

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

```

```

mm <- rbind(m, 1:4)

mm

```

```

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

```

```

mm <- rbind(m, 10)

mm

```

```

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

```

The `rbind()` function can be used in the same way for data frames, but it can also be used to match up columns by name.

```

d <- data.frame(ID = c("A", "B", "C"), response = 1:3)

d

```

```

#   ID response
# 1  A         1
# 2  B         2
# 3  C         3

```

```

dd <- rbind(d, c("D", 4))

# Warning in '[<-.factor'('*tmp*', ri, value = "D"): invalid factor level, NA generated

```

R is trying to be smart with a factor here. Here is one of several options:

```

d$ID <- as.character(d$ID)
dd <- rbind(d, c("D", 4))
dd

```

```

# ID response
# 1 A      1
# 2 B      2
# 3 C      3
# 4 D      4

```

The `rbind()` function can also use column names.

```

s <- read.csv("../data/soil_cat.csv", as.is = TRUE)
s

#          soil    mg     k     ca
# 1 Nottingham 34.3 65.6 98.4
# 2 Houthalen   4.0 31.0 21.4
# 3 Rhydtalog  49.4 44.1 214.0
# 4 Zegveld    124.0 76.7 799.0
# 5 Kovlinge I 37.5 30.2 140.0
# 6 Souli I    13.4  0.0  72.0
# 7 Kovlinge II 41.1 34.7 218.0
# 8 Montpellier 38.4 23.9 147.0

m <- read.csv("../data/soils.csv", as.is = TRUE)
m

#          soil    mg     ca     k
# 1 Aluminusa 43.2 120 15.7
# 2 Woburn    94.5 412 24.1
# 3 Ter Munck 65.9 704 124.0

```

When we add a row (or multiple rows), note that column order does not need to match. But, the added row(s) must be a data frame!

```

ss <- rbind(s, m)
ss

#          soil    mg     k     ca
# 1 Nottingham 34.3 65.6 98.4
# 2 Houthalen   4.0 31.0 21.4

```

```

# 3     Rhydtalog 49.4 44.1 214.0
# 4     Zegveld 124.0 76.7 799.0
# 5   Kovlinge I 37.5 30.2 140.0
# 6     Souli I 13.4 0.0 72.0
# 7  Kovlinge II 41.1 34.7 218.0
# 8 Montpellier 38.4 23.9 147.0
# 9   Aluminusa 43.2 15.7 120.0
# 10    Woburn 94.5 24.1 412.0
# 11 Ter Munck 65.9 124.0 704.0

```

This function will only work if the number of columns or rows in the two objects you want to combine are identical. What if you don't want to add rows, but want to combine rows? The `merge()` function is more powerful—it is used to combine data frames by some common variable or variables. To demonstrate its use, let's read in some data.

```

pop <- read.csv("../data/us_pop.csv")
dfsumm(pop)

#
# 22 rows and 2 columns
# 22 unique rows
#           year      pop
# Class       integer  integer
# Minimum      1790 3929214
# Maximum      2000 281421906
# Mean         1890 62979766
# Unique (excl. NA)    22      22
# Missing values        0      0
# Sorted        TRUE     TRUE

pop

#      year      pop
# 1 1790 3929214
# 2 1800 5308483
# 3 1810 7239881
# 4 1820 9638453
# 5 1830 12866020
# 6 1840 17069453
# 7 1850 23191876
# 8 1860 31443321
# 9 1870 38558371
# 10 1880 50189209
# 11 1890 62979766
# 12 1900 76212168
# 13 1910 92228496
# 14 1920 106021537
# 15 1930 123202624
# 16 1940 132164569
# 17 1950 151325798
# 18 1960 179323175

```

```

# 19 1970 203302031
# 20 1980 226542199
# 21 1990 248709873
# 22 2000 281421906

gdp <- read.csv("../data/us_gdp.csv")
dfsumm(gdp)

#
# 220 rows and 3 columns
# 220 unique rows
#          year  gdp.nom gdp.real
# Class      integer  integer   integer
# Minimum     1790      187     4027
# Maximum     2009 14441400 13312200
# Mean        1899      19506    412475
# Unique (excl. NA)    220      220     220
# Missing values      0       0       0
# Sorted        TRUE     FALSE    FALSE

```

To merge these two data frames by year, we can use the following command:

```

us <- merge(pop, gdp, by = "year")
us

#      year      pop  gdp.nom gdp.real
# 1  1790 3929214      187     4027
# 2  1800 5308483      476     7398
# 3  1810 7239881      699    10626
# 4  1820 9638453      703    14414
# 5  1830 12866020     1012    22162
# 6  1840 17069453     1559    31461
# 7  1850 23191876     2556    49586
# 8  1860 31443321     4345    82107
# 9  1870 38558371     7737   112276
# 10 1880 50189209    10362   191814
# 11 1890 62979766    15077   319077
# 12 1900 76212168    20567   422843
# 13 1910 92228496    33423   533767
# 14 1920 106021537   88393   687704
# 15 1930 123202624   91200   892800
# 16 1940 132164569  101400  1166900
# 17 1950 151325798  293700  2006000
# 18 1960 179323175  526400  2830900
# 19 1970 203302031 1038300  4269900
# 20 1980 226542199 2788100  5839000
# 21 1990 248709873 5800500  8033900
# 22 2000 281421906 9951500 11226000

```

Notice that us does not have rows for all the years that are present in the gdp data frame, because most of them didn't have a matching year in pop.

```

dim(gdp)

# [1] 220 3

dim(pop)

# [1] 22 2

dim(us)

# [1] 22 4

```

If you want to keep all rows regardless, use `all = TRUE`.

```

us <- merge(pop, gdp, by = "year", all = TRUE)
dfsumm(us)

#
# 220 rows and 4 columns
# 220 unique rows
#          year      pop gdp.nom gdp.real
# Class    integer  integer  integer  integer
# Minimum   1790 3929214      187     4027
# Maximum   2009 281421906 14441400 13312200
# Mean      1899 62979766     19506    412475
# Unique (excl. NA)    220       22     220     220
# Missing values        0       198       0       0
# Sorted      TRUE      TRUE     FALSE    FALSE

head(us)

#   year      pop gdp.nom gdp.real
# 1 1790 3929214      187     4027
# 2 1791       NA      204     4268
# 3 1792       NA      223     4583
# 4 1793       NA      249     4947
# 5 1794       NA      312     5601
# 6 1795       NA      380     5956

```

Do you see what R does for the rows with no match in `pop`?

Merge operations can get more complicated than the simple example demonstrated above. For example, what if the variables you would like to merge by don't have the same name in both data frames? In this case, just use the arguments `by.x` and `by.y` instead of `by`. If the data frames have variables that you are not merging by with the same names, R will add extensions to the variable names (`.x` and `.y` by default—see `suffix` argument) in the new data frame. You can merge by multiple variables by specifying a vector with the names of all variables, e.g., `by = c("var1", "var2")`.

## 15 More on data manipulation: aggregating and summarizing data

R has some powerful functions for aggregation of data. For most operations, there are multiple functions available in the base packages that could be used, often with slight differences. Unfortunately, this quality makes data aggregation in R a bit complicated. Furthermore, the capability of functions in packages developed by Hadley Wickham, dplyr, plyr, and reshape2, overlap with base functions (and with each other). My recommendation is to use dplyr, plyr, and reshape2 functions for all but the simplest operations. In this section, we will start with some (relatively) simple operations, and then discuss approaches for more advanced aggregation.

### 15.1 Counts

The `table()` function is handy for summarizing counts of factor data. To demonstrate, let's read in some data on freshwater fish in Oregon.

```
fish <- read.csv("../data/salmonids.csv")
dfsumm(fish)

#
# 22453 rows and 18 columns
# 22453 unique rows
#
#          DBCODE ENTITY VERT_INDEX SITECODE      YEAR SECTION
# Class     factor integer    integer   factor integer   factor
# Minimum   AS006      1           1 MACKAL-L    1987      AL
# Maximum   AS006      1           22453 MACKOG-U   2010      OG
# Mean      AS006      1           11226 MACKOG-L   2001      CC
# Unique (excl. NA) 1           1           22453      8       24      3
# Missing values 0           0           0           0       0       0
# Sorted     TRUE        TRUE        TRUE      FALSE      TRUE      FALSE
#
#          REACH PASS UNITTYPE UNITNUM PITNUMBER SPECIES
# Class     factor integer    factor numeric   integer   factor
# Minimum   L         1           1           2615067
# Maximum   U         2           SC          26228419  RHOL
# Mean      M         1           IP          8.06     16394808  ONCL
# Unique (excl. NA) 3           2           8           23       1316      4
# Missing values 0           0           0           610      20943      0
# Sorted     FALSE      FALSE      FALSE      FALSE      FALSE      FALSE
#
#          LENGTH1 LENGTH2 WEIGHT CLIP SAMPLEDATE NOTES
# Class     integer integer numeric factor   factor   factor
# Minimum   19        32        0.09    1987-10-06
# Maximum   253       278       135     RV 2010-09-10 WOUNDED
# Mean      66        101       9.63    2000-08-22 40% FLOW
# Unique (excl. NA) 181       213       2431      4       66       135
# Missing values 9        15025      13190      0       0       0
# Sorted     FALSE      FALSE      FALSE      FALSE      FALSE      FALSE
```

Let make all the column names lowercase to make typing easier.

```
names(fish) <- tolower(names(fish))
```

If we take a close look at the summary, we can tell that some of the factors have a blank level. R will recognize blank entries in a file as NA for just about everything except character data, so these values should have been NA in the data file. Let's use an indexing operation on the whole data frame to remove all of these<sup>109</sup>

```
fish[fish == ""] <- NA
fish <- droplevels(fish)
dfsumm(fish)

#
# 22453 rows and 18 columns
# 22453 unique rows
#
#          dbcode entity vert_index sitecode      year section
# Class      factor integer    integer factor integer factor
# Minimum    AS006      1           1 MACKAL-L  1987     AL
# Maximum    AS006      1        22453 MACKOG-U  2010     OG
# Mean       AS006      1        11226 MACKOG-L  2001     CC
# Unique (excl. NA) 1       1        22453      8      24      3
# Missing values 0       0           0      0      0      0
# Sorted      TRUE    TRUE      TRUE FALSE    TRUE FALSE
#          reach   pass unittype unitnum pitnumber species
# Class      factor integer    factor numeric integer factor
# Minimum    L       1           C       1 2615067 DITE
# Maximum    U       2           SC      20 26228419 RHOL
# Mean       M       1           IP      8.06 16394808 ONCL
# Unique (excl. NA) 3       2           7       23 1316      3
# Missing values 0       0           610     610 20943      5
# Sorted      FALSE  FALSE  FALSE FALSE  FALSE FALSE
#          length1 length2 weight  clip sampledate
# Class      integer integer numeric factor   factor
# Minimum    19       32     0.09    LV 1987-10-06
# Maximum    253      278     135     RV 2010-09-10
# Mean       66       101    9.63    LVRV 2000-08-22
# Unique (excl. NA) 181      213    2431      3      66
# Missing values 9       15025  13190  19783      0
# Sorted      FALSE  FALSE  FALSE FALSE  FALSE FALSE
#          notes
# Class      factor
# Minimum    1/2 TAIL GONE
# Maximum    WOUNDED
# Mean       LARGE BITE ON SIDE
# Unique (excl. NA) 134
# Missing values 20873
# Sorted      FALSE
```

There are a lot of data here, but we'll just focus on a few variables.

The `table()` function can be used to provide counts of observations in a variety of ways.

---

<sup>109</sup> What we are really doing in the first command below is using a logical matrix to index the data frame.

```
table(fish$species)
```

```
#  
# DITE ONCL RHOL  
# 7671 14770 7
```

Let's say we are interested in when these species were found.

```
table(fish$year, fish$species)
```

```
#  
# DITE ONCL RHOL  
# 1987 0 603 0  
# 1988 0 302 0  
# 1989 0 308 0  
# 1990 0 513 0  
# 1991 0 626 0  
# 1992 0 616 0  
# 1993 255 615 0  
# 1994 306 642 2  
# 1995 193 390 0  
# 1996 336 799 0  
# 1997 229 872 0  
# 1998 236 888 0  
# 1999 232 564 0  
# 2000 379 673 0  
# 2001 323 692 0  
# 2002 645 544 1  
# 2003 732 561 3  
# 2004 443 563 1  
# 2005 569 538 0  
# 2006 424 617 0  
# 2007 557 758 0  
# 2008 631 663 0  
# 2009 609 862 0  
# 2010 572 561 0
```

Or how numbers for each species varied by year and also location within each river or stream.

```
table(fish$year, fish$species, fish$reach)
```

```
# , , = L  
#  
# DITE ONCL RHOL  
# 1987 0 187 0  
# 1988 0 107 0  
# 1989 0 113 0  
# 1990 0 191 0  
# 1991 0 213 0
```

```

# 1992   0 220  0
# 1993   83 201  0
# 1994  116 234  2
# 1995   82 156  0
# 1996  109 285  0
# 1997   90 305  0
# 1998   72 303  0
# 1999   52 172  0
# 2000   69 160  0
# 2001   73 234  0
# 2002  204 172  1
# 2003  201 186  2
# 2004  152 176  1
# 2005  141 160  0
# 2006  103 190  0
# 2007  127 219  0
# 2008  174 187  0
# 2009  176 273  0
# 2010  167 180  0
#
# , , = M
#
#
#      DITE ONCL RHOL
# 1987   0 184  0
# 1988   0 110  0
# 1989   0 100  0
# 1990   0 160  0
# 1991   0 193  0
# 1992   0 173  0
# 1993   67 174  0
# 1994   78 196  0
# 1995   38  77  0
# 1996   87 201  0
# 1997   66 267  0
# 1998   92 311  0
# 1999   87 181  0
# 2000  164 210  0
# 2001  125 200  0
# 2002  195 177  0
# 2003  257 199  0
# 2004  128 187  0
# 2005  162 160  0
# 2006  125 177  0
# 2007  189 266  0
# 2008  216 238  0
# 2009  207 257  0
# 2010  206 184  0
#
# , , = U
#
#
#      DITE ONCL RHOL

```

```

#   1987    0  232    0
#   1988    0   85    0
#   1989    0   95    0
#   1990    0  162    0
#   1991    0  220    0
#   1992    0  223    0
#   1993  105  240    0
#   1994  112  212    0
#   1995   73  157    0
#   1996  140  313    0
#   1997   73  300    0
#   1998   72  274    0
#   1999   93  211    0
#   2000  146  303    0
#   2001  125  258    0
#   2002  246  195    0
#   2003  274  176    1
#   2004  163  200    0
#   2005  266  218    0
#   2006  196  250    0
#   2007  241  273    0
#   2008  241  238    0
#   2009  226  332    0
#   2010  199  197    0

```

The `table()` function can also be used with relational constraints (it is really just being used with logical vectors). Say we want to know how many of each species was below the 25<sup>th</sup> percentile for weight.

```
quantile(fish$weight, 0.25, na.rm = TRUE)
```

```

# 25%
# 1.45

```

```
table(fish$weight < 1.45, fish$species)
```

```

#
#          DITE ONCL RHOL
# FALSE 2062 4913     1
# TRUE   194 2092     1

```

Or, how the missing values of weight are distributed among species.

```
table(is.na(fish$weight), fish$species)
```

```

#
#          DITE ONCL RHOL
# FALSE 2256 7005     2
# TRUE  5415 7765     5

```

Of course, `table` will accept more than two factors<sup>110</sup>, which makes `table()` handy for assessing balance and replication in factorial studies.

```
respir <- read.csv("../data/respiration.csv")
dfsumm(respir)

#
# 72 rows and 4 columns
# 69 unique rows
#
#          sp   temp    sex   resp
# Class      integer factor factor numeric
# Minimum        1     high      F      1
# Maximum        3     med      M     3.6
# Mean           2     low      M     2.33
# Unique (excl. NA) 3       3      2     24
# Missing values 0       0      0      0
# Sorted         TRUE FALSE FALSE FALSE
```

This data set is from a factorial experiment on the respiration rate of both sexes of two crab species at two temperatures <sup>111</sup>. To check replication and balance, just include all the factors in the function call.

```
table(respir$sp, respir$temp, respir$sex)
```

```
# , , = F
#
#
#      high low med
# 1     4   4   4
# 2     4   4   4
# 3     4   4   4
#
# , , = M
#
#
#      high low med
# 1     4   4   4
# 2     4   4   4
# 3     4   4   4
```

It is balanced and fully crossed, with  $n = 4$ .

## 15.2 Data aggregation and grouped operations

For more flexible data aggregation, the base packages in R include the following functions: `apply()`, `lapply()`, `aggregate()`, `by()`, and `tapply()`. The functions in the `plyr` and `reshape2` packages are more powerful and flexible than the functions included in the base packages. And, the tidyverse

---

<sup>110</sup> Although `xtabs()` or `ftable()` may be better alternatives. And, the `ddply()` function can produce similar output in a data frame format.

<sup>111</sup> I suspect that these data are actually fabricated. I copied them from Zar (1999).

approach, defined in the newer `dplyr` package, focuses on data frames. The `tidyverse` and `plyr` functions have a common syntax while the base packages functions do not. I recommend that you focus on `dplyr` and, if needed (if you are not working with a data frame), `plyr`, but be aware that other functions are available.

The `plyr` functions employ a common approach: an input object (e.g., data frame) is broken apart by, e.g., the value of some variable, some operation is applied to each part separately, and finally, the pieces are put back together again. The syntax of the `plyr` function names is simple: the first letter indicates the type of data object that is being operated on and the second letter indicates the structure of the output (`d` for data frame; `a` for vectors, matrices, and arrays; `l` for list; and `_` for nothing). For example, to operate on a data frame and return the results as a list, we would use `dlply`.

The following table should give you an idea of how to select a function for a particular data.

Input object	Groups defined by	Output	Base function	plyr function
array	rows, columns, etc.	array	<code>apply()</code>	<code>aapply()</code>
vector	values in other vector	list	<code>tapply()</code>	<code>alply()</code>
data frame	variable(s) within	data frame	<code>aggregate()</code>	<code>ddply()</code>
data frame	variable(s) within	list	<code>by()</code>	<code>dlply()</code>
list	elements in list	list	<code>lapply(), sapply()</code>	<code>llply()</code>
list	elements in list	none		<code>l_ply()</code>
data frame	variable(s) within	none		<code>d_ply()</code>

The table above doesn't show all of the options that are available in the `plyr` package, e.g., `adply()` also exists. And `dplyr` functions do not fit neatly into this table. Relevant ones include `group_by()`, `summarise()`, and others. For these, input and output is always a data frame (a tibble, really), and the `group_by` operation splits the data frame, and the others carry out the operation on each part.

Starting with the most simple of these functions, sometimes it is necessary to carry out some operation on complete rows or columns within arrays or data frames. The base function `apply` can be used for this. Let's look at data on the call of a black-throated green warbler. We'll work with a spectrogram, which is a matrix with sound volume for multiple frequencies over time.

```
warb <- as.matrix(read.csv("../data/warbler.csv", row.names = 1))
dim(warb)

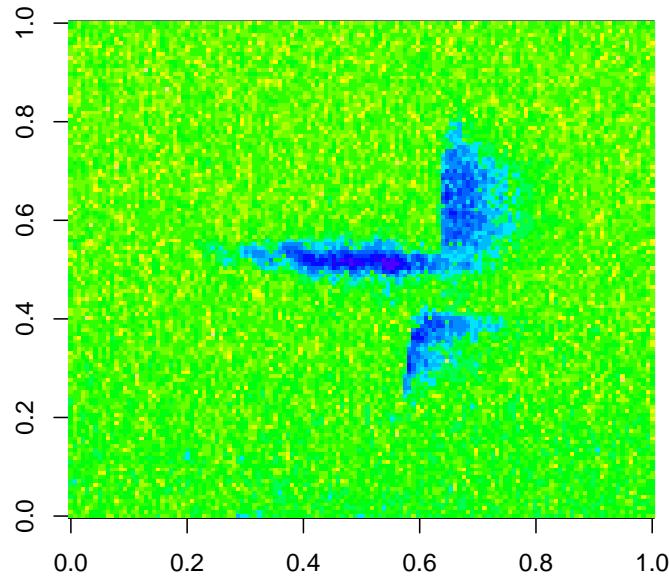
# [1] 128 140

warb[1:5, 1:5]

#          t0 t0.02133 t0.04267 t0.064 t0.08533
# f2.016 -55.33   -56.41   -54.12 -56.18   -55.40
# f2.062 -61.35   -54.39   -50.04 -57.54   -66.01
# f2.109 -54.52   -45.99   -55.57 -50.96   -68.65
# f2.156 -51.72   -47.51   -58.44 -47.25   -51.88
# f2.203 -57.51   -49.49   -66.01 -49.04   -50.92
```

A good way to “view” a not-so-small matrix like this is with the `image()` function. In the call below, I also use the `t()` function, which transposes the matrix so time is shown on the x axis.

```
#image(t(warb), col = terrain.colors(20))
image(t(-warb), col = topo.colors(20))
```



We'll work with just a part of this matrix, so we can display results in this book easily.

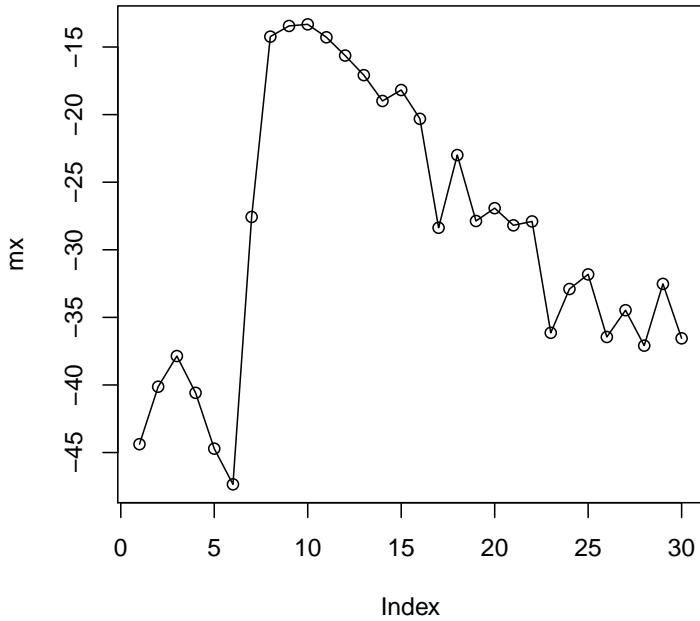
```
warb <- warb[30:59, 75:104]
```

Now, say we wanted to know what the maximum or minimum volumes (amplitudes) were for each time bin.

```
mx <- apply(warb, 2, max)
mx
```

```
# t1.579   t1.6   t1.621  t1.643  t1.664  t1.685  t1.707  t1.728  t1.749  t1.771
# -44.38  -40.13  -37.86  -40.58  -44.71  -47.35  -27.57  -14.23  -13.44  -13.32
# t1.792   t1.813  t1.835  t1.856  t1.877  t1.899  t1.92   t1.941  t1.963  t1.984
# -14.28  -15.63  -17.08  -18.99  -18.18  -20.31  -28.37  -22.99  -27.87  -26.92
# t2.005   t2.027  t2.048  t2.069  t2.091  t2.112  t2.133  t2.155  t2.176  t2.197
# -28.19  -27.90  -36.14  -32.90  -31.82  -36.45  -34.47  -37.09  -32.52  -36.55
```

```
plot(mx, type = "o")
```



Or, if we wanted both the minimum and the maximum.

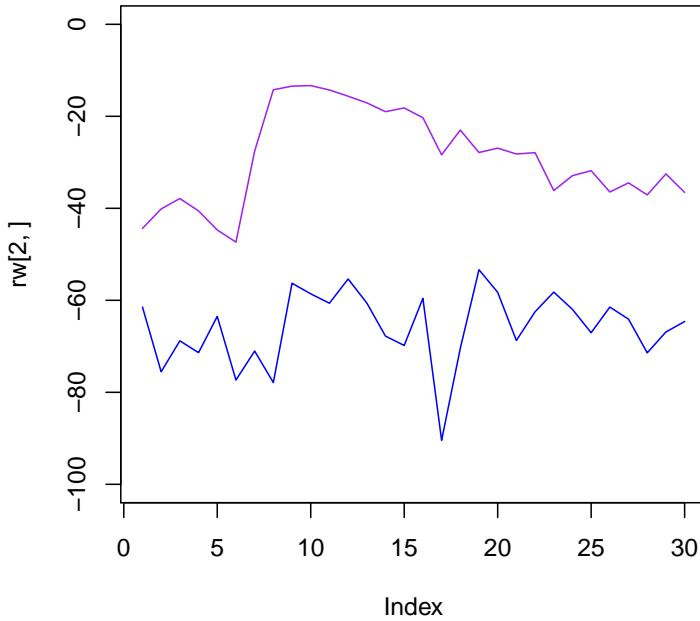
```

rw <- apply(warb, 2, range)
rw

#      t1.579   t1.6   t1.621   t1.643   t1.664   t1.685   t1.707   t1.728   t1.749
# [1,] -61.47 -75.53 -68.81 -71.38 -63.49 -77.33 -71.03 -77.87 -56.29
# [2,] -44.38 -40.13 -37.86 -40.58 -44.71 -47.35 -27.57 -14.23 -13.44
#      t1.771   t1.792   t1.813   t1.835   t1.856   t1.877   t1.899   t1.92   t1.941
# [1,] -58.59 -60.63 -55.36 -60.58 -67.79 -69.81 -59.57 -90.44 -70.45
# [2,] -13.32 -14.28 -15.63 -17.08 -18.99 -18.18 -20.31 -28.37 -22.99
#      t1.963   t1.984   t2.005   t2.027   t2.048   t2.069   t2.091   t2.112   t2.133
# [1,] -53.35 -58.27 -68.73 -62.49 -58.22 -61.97 -67.04 -61.47 -64.10
# [2,] -27.87 -26.92 -28.19 -27.90 -36.14 -32.90 -31.82 -36.45 -34.47
#      t2.155   t2.176   t2.197
# [1,] -71.42 -66.88 -64.61
# [2,] -37.09 -32.52 -36.55

plot(rw[2, ], type = "l", col = "purple", ylim = c(-100, 0))
lines(rw[1, ], col = "blue")

```



The second argument in the `apply()` function (`MARGIN`) lets you control whether the function is applied over rows (1) or columns (2). So, to find the minimum and maximum volumes for all *frequency* bins, we could use this.

```
apply(warb, 1, range)

#      f3.375 f3.422 f3.469 f3.516 f3.562 f3.609 f3.656 f3.703 f3.75
# [1,] -72.73 -64.10 -69.81 -63.65 -64.16 -62.49 -63.46 -58.43 -66.66
# [2,] -40.63 -41.66 -39.28 -32.99 -28.83 -27.57 -30.04 -28.92 -20.02
#      f3.797 f3.844 f3.891 f3.938 f3.984 f4.031 f4.078 f4.125 f4.172
# [1,] -66.05 -77.33 -90.44 -58.29 -65.96 -65.86 -65.73 -60.18 -75.16
# [2,] -15.20 -14.23 -15.91 -18.51 -20.83 -23.62 -21.70 -16.82 -13.44
#      f4.219 f4.266 f4.312 f4.359 f4.406 f4.453   f4.5 f4.547 f4.594
# [1,] -59.53 -77.87 -64.38 -71.38 -60.18 -71.03 -75.53 -66.88 -70.45
# [2,] -13.32 -16.91 -17.08 -15.63 -16.71 -25.79 -37.45 -40.30 -40.58
#      f4.641 f4.688 f4.734
# [1,] -65.06 -60.89 -65.53
# [2,] -42.87 -37.86 -38.35
```

The `plyr` alternative to this function is `aapply()`<sup>112</sup>.

```
plyr::aapply(warb, 2, range)
```

```
#
```

---

<sup>112</sup> I'm using the `::` operator to avoid loading `plyr`, which causes problems when using some `dplyr` functions because of common names. Typically this can be avoided by using `detach` but in this Rnw document, perhaps due to caching, problems are difficult to avoid.

```

# X1          1      2
#   t1.579 -61.47 -44.38
#   t1.6    -75.53 -40.13
#   t1.621 -68.81 -37.86
#   t1.643 -71.38 -40.58
#   t1.664 -63.49 -44.71
#   t1.685 -77.33 -47.35
#   t1.707 -71.03 -27.57
#   t1.728 -77.87 -14.23
#   t1.749 -56.29 -13.44
#   t1.771 -58.59 -13.32
#   t1.792 -60.63 -14.28
#   t1.813 -55.36 -15.63
#   t1.835 -60.58 -17.08
#   t1.856 -67.79 -18.99
#   t1.877 -69.81 -18.18
#   t1.899 -59.57 -20.31
#   t1.92   -90.44 -28.37
#   t1.941 -70.45 -22.99
#   t1.963 -53.35 -27.87
#   t1.984 -58.27 -26.92
#   t2.005 -68.73 -28.19
#   t2.027 -62.49 -27.90
#   t2.048 -58.22 -36.14
#   t2.069 -61.97 -32.90
#   t2.091 -67.04 -31.82
#   t2.112 -61.47 -36.45
#   t2.133 -64.10 -34.47
#   t2.155 -71.42 -37.09
#   t2.176 -66.88 -32.52
#   t2.197 -64.61 -36.55

```

We can transpose the result using the `t()` function to change its orientation. Note that while either of these functions could be used for simply summing values, there are two simpler functions for this task: `rowSums()` and `colSums()`, both of which accept a data frame or matrix as the only required argument.

This function can be used with any kind of atomic data structure, including matrices and arrays. Let's work with an array in the `datasets` package on admissions to UC Berkeley<sup>113</sup>.

```

ad <- UCBAdmissions
ad

# , , Dept = A
#
#           Gender
# Admit     Male Female
#   Admitted 512     89
#   Rejected 313     19
#
# , , Dept = B

```

---

<sup>113</sup> Actually a `table()`, but that just means an `array` with integer values.

```

#
#           Gender
# Admit      Male Female
#   Admitted 353     17
#   Rejected 207     8
#
# , , Dept = C
#
#           Gender
# Admit      Male Female
#   Admitted 120     202
#   Rejected 205     391
#
# , , Dept = D
#
#           Gender
# Admit      Male Female
#   Admitted 138     131
#   Rejected 279     244
#
# , , Dept = E
#
#           Gender
# Admit      Male Female
#   Admitted 53      94
#   Rejected 138     299
#
# , , Dept = F
#
#           Gender
# Admit      Male Female
#   Admitted 22      24
#   Rejected 351     317

```

We could find the total number of applicants admitted and rejected with this plyr function.

```

plyr::aaply(ad, 1, sum)

# Admitted Rejected
#       1755      2771

```

Or, if we were interested in how application and admission varied across departments, the next command would be helpful.

```

plyr::aaply(ad, c(1, 3), sum)

#           Dept
# Admit      A   B   C   D   E   F
#   Admitted 601 370 322 269 147  46
#   Rejected 332 215 596 523 437 668

```

Now for more complicated examples. A typical operation is this: you have a data frame with some grouping variable, and you need to carry out some operation on individual groups within the data. Take, for example, these data on the level of alpha-Chlordane in eagle blood.

```
eagles <- read.csv("../data/eagles.csv")
head(eagles)
```

```
#          site achlor
# 1      magee Marsh   1.25
# 2      Magee Marsh   1.55
# 3      Magee Marsh   1.74
# 4      Magee Marsh   1.77
# 5 Davis Besse-Toussaint   1.82
# 6 Carroll Twp-Camp Perry   2.33
```

```
dfsumm(eagles)
```

```
# 
# 147 rows and 2 columns
# 126 unique rows
#          site achlor
# Class      factor numeric
# Minimum    Ballville 0.375
# Maximum    Wills Ck  5.45
# Mean       Mercer   1.54
# Unique (excl. NA) 40     92
# Missing values 0      0
# Sorted      FALSE   FALSE
```

Let's say we want to know the mean value of achlor for each site. The function `ddply()` is the best bet for this task (functions within the base packages that could also be used are `tapply()` and `aggregate()`). The main arguments of `ddply()`, which are shared in part with the other `plyr` functions, are `.data`, `.variables`, and `.fun`, which are the data frame to be operated on, the variable(s) used for grouping, and the function to apply, respectively.

You might think you could use the following code to do this,

```
# 
# ddply(.data = eagles, .variables = "site", .fun = mean)
```

but you can't (or at least it won't work well). The `ddply` function operates on the entire data frame, and the mean function can't handle the site vector. So, there are two options that I'll explain. The first, and most flexible, approach is to write a new function that accepts a data frame.

```
plyr::ddply(.data = eagles, .variables = "site", .fun = function(x) mean(x$achlor))

#          site      V1
# 1      Ballville 2.7250000
# 2      Camp Perry 1.3250000
# 3 Carroll Twp-Camp Perry 2.3900000
```

```

# 4 Carroll Twp-Toussaint 2.3400000
# 5 Cedar Pt NWR 3.0580000
# 6 Cr 306 2.9900000
# 7 CR 306 2.2850000
# 8 Davis Besse-Toussaint 1.8200000
# 9 Ft Seneca 1.3414286
# 10 Garlo 0.3750000
# 11 Gibsonburg 1.8950000
# 12 Gonya 1.8550000
# 13 Indian Mill-Sycamore 1.3600000
# 14 Killdeer 0.4650000
# 15 Kinsman 0.3750000
# 16 Knobby 0.7825000
# 17 magee Marsh 1.2500000
# 18 Magee Marsh 1.8842857
# 19 Meander 1.2922222
# 20 Mercer 1.0300000
# 21 Metzger-Peach Is 1.4900000
# 22 Mosquito 0.3750000
# 23 Mud Ck 2.0757143
# 24 Old Womans Ck 1.1016667
# 25 Ottawa NWR 2.1412500
# 26 Ottawa SC 1.9200000
# 27 Peach Is 0.9725000
# 28 Pickeral Ck 1.2150000
# 29 Pymatuning 0.3750000
# 30 Rockwell 1.2556250
# 31 Rookery 0.9566667
# 32 Rossford 2.9900000
# 33 Sandusky Airport 1.6850000
# 34 Sass 2.7350000
# 35 Shandngo 0.3750000
# 36 Shendngo 0.4560000
# 37 Smithville-Sycamore 1.4450000
# 38 Snow Lk 0.9787500
# 39 Tri Valley-Wills Ck 3.8200000
# 40 Wills Ck 3.6225000

```

In the above code the actual `.fun` argument is a user-defined function, which is used to apply the `mean()` function to the `achlor` column. For some more complicated operations, you may need to do this, but usually, the second option is the better one. The `plyr` package includes a `summarise()` function, which works very well for just this type of problem. It applies a named function to subsets of a data frame, and returns a data frame with one row for each subset.

```
plyr:::ddply(eagles, "site", summarise, mean = mean(achlor))
```

```

#           site      mean
# 1     Ballville 2.7250000
# 2   Camp Perry 1.3250000
# 3 Carroll Twp-Camp Perry 2.3900000
# 4 Carroll Twp-Toussaint 2.3400000
# 5 Cedar Pt NWR 3.0580000
# 6       Cr 306 2.9900000

```

```

# 7          CR 306 2.2850000
# 8  Davis Besse-Toussaint 1.8200000
# 9          Ft Seneca 1.3414286
# 10         Garlo 0.3750000
# 11         Gibsonburg 1.8950000
# 12         Gonya 1.8550000
# 13  Indian Mill-Sycamore 1.3600000
# 14         Killdeer 0.4650000
# 15         Kinsman 0.3750000
# 16         Knobby 0.7825000
# 17         magee Marsh 1.2500000
# 18         Magee Marsh 1.8842857
# 19         Meander 1.2922222
# 20         Mercer 1.0300000
# 21  Metzger-Peach Is 1.4900000
# 22         Mosquito 0.3750000
# 23         Mud Ck 2.0757143
# 24  Old Womans Ck 1.1016667
# 25         Ottawa NWR 2.1412500
# 26         Ottawa SC 1.9200000
# 27         Peach Is 0.9725000
# 28         Pickeral Ck 1.2150000
# 29         Pymatuning 0.3750000
# 30         Rockwell 1.2556250
# 31         Rookery 0.9566667
# 32         Rossford 2.9900000
# 33  Sandusky Airport 1.6850000
# 34         Sass 2.7350000
# 35         Shandngo 0.3750000
# 36         Shendngo 0.4560000
# 37  Smithville-Sycamore 1.4450000
# 38         Snow Lk 0.9787500
# 39  Tri Valley-Wills Ck 3.8200000
# 40         Wills Ck 3.6225000

```

This call may make more sense if you take a look at the help file for `ddply()`. Any argument that is not a listed formal arguments will be passed onto the `.fun` function. Notice that we can name the new column right in the function call. And, `summarise()` can handle more than one operation (even if they use different vectors within the data frame, unlike the example given below).

```
plyr::ddply(.data = eagles, .variables = "site", .fun = summarise, mean = mean(achlor),
            sd = sd(achlor), n = length(achlor))
```

```

#           site      mean       sd n
# 1     Ballville 2.7250000 1.05809577 4
# 2     Camp Perry 1.3250000 0.84020236 6
# 3 Carroll Twp-Camp Perry 2.3900000 0.08485281 2
# 4 Carroll Twp-Toussaint 2.3400000        NA 1
# 5     Cedar Pt NWR 3.0580000 0.74563396 5
# 6          Cr 306 2.9900000        NA 1
# 7          CR 306 2.2850000 0.71512237 8
# 8  Davis Besse-Toussaint 1.8200000        NA 1
# 9     Ft Seneca 1.3414286 0.69694999 7

```

```

# 10           Garlo 0.3750000 0.00000000 3
# 11       Gibsonburg 1.8950000 0.95993489 5
# 12           Gonya 1.8550000 0.64346717 2
# 13 Indian Mill-Sycamore 1.3600000      NA 1
# 14       Killdeer 0.4650000 0.20551329 8
# 15       Kinsman 0.3750000 0.00000000 3
# 16       Knobby 0.7825000 0.57629203 2
# 17   magee Marsh 1.2500000      NA 1
# 18   Magee Marsh 1.8842857 0.48286841 7
# 19       Meander 1.2922222 0.76579253 9
# 20       Mercer 1.0300000 0.05656854 2
# 21 Metzger-Peach Is 1.4900000      NA 1
# 22       Mosquito 0.3750000 0.00000000 3
# 23       Mud Ck 2.0757143 0.42972306 7
# 24 Old Womans Ck 1.1016667 0.77656831 3
# 25       Ottawa NWR 2.1412500 1.20925855 4
# 26       Ottawa SC 1.9200000      NA 1
# 27       Peach Is 0.9725000 0.69685603 4
# 28     Pickeral Ck 1.2150000 0.40742075 7
# 29       Pymatuning 0.3750000 0.00000000 2
# 30       Rockwell 1.2556250 0.66600857 8
# 31       Rookery 0.9566667 0.49797256 6
# 32       Rossford 2.9900000 0.86267027 2
# 33 Sandusky Airport 1.6850000 0.36062446 2
# 34       Sass 2.7350000 0.85559921 2
# 35       Shandngo 0.3750000      NA 1
# 36       Shendngo 0.4560000 0.18112151 5
# 37 Smithville-Sycamore 1.4450000 0.02121320 2
# 38       Snow Lk 0.9787500 0.44317745 4
# 39 Tri Valley-Wills Ck 3.8200000      NA 1
# 40       Wills Ck 3.6225000 1.56991242 4

```

If this expression does not make sense yet, it is worth some effort to understand it. It may be helpful to see what `summarise` does on its own.

```

args(plyr::summarise)

# function (.data, ...)
# NULL

dat <- subset(eagles, site == "Magee Marsh")
plyr::summarise(dat, mean = mean(achlor), sd = sd(achlor), n = length(achlor))

#      mean      sd n
# 1 1.884286 0.4828684 7

```

In the `plyr` functions, the actual argument for `.variables` can be expressed using a quoted name of a variable within the data frame, e.g., `"site"`, by enclosing the name of the variable in `.()`, e.g., `.(site)`, or using a tilde, e.g., `site`. I recommend picking one option and sticking with it.

Because these types of manipulations are important and common, let's spend some time on a few variations. We'll work with some data on methane and carbon dioxide emission from Alaskan soils.

```

gas <- read.csv("../data/gas_emis.csv")
dfsumm(gas)

#
# 326 rows and 8 columns
# 326 unique rows
#      ecosystem      date    tmt   block morphology
# Class          factor     factor factor integer     factor
# Minimum        HEATH 1993/06/21      CT      1       I
# Maximum        TUSSOCK 1993/08/18     NP      4       T
# Mean           SEDGE 1993/07/21     GH      2       T
# Unique (excl. NA) 3          11      3      4       2
# Missing values 0          0      0      0       0
# Sorted          FALSE     TRUE   FALSE  FALSE  FALSE
#                  CH4      CO2    temp
# Class          numeric integer numeric
# Minimum        -14.7      16    1.98
# Maximum        648      12783   13.6
# Mean           44.7      1627   5.66
# Unique (excl. NA) 322      311    54
# Missing values 0          0    206
# Sorted          FALSE   FALSE  FALSE

```

In this experiment, three different treatments (control, greenhouse, or nitrogen and phosphorus fertilization) were applied in two different types of locations in two ecosystems. Let's say we want the mean and standard deviation of methane emission for all combinations of `ecosystem`, `tmt`, and `morphology`.

```

ch4summ <- plyr::ddply(gas, c("ecosystem", "morphology", "tmt"), plyr::summarise,
                        mean = mean(CH4), sd = sd(CH4), n = length(CH4))
ch4summ

#      ecosystem morphology tmt      mean        sd  n
# 1      HEATH         I  CT -0.7064000 0.7368068 15
# 2      HEATH         I  GH -1.8418750 0.7083178  8
# 3      HEATH         I  NP -0.6285333 1.2341800 15
# 4      HEATH         T  CT -0.8669333 0.7816677 15
# 5      HEATH         T  GH -1.3303889 0.7153461 18
# 6      HEATH         T  NP  0.0010000 1.4237865 15
# 7      SEDGE         I  CT  92.9698000 49.0933384 10
# 8      SEDGE         I  GH 247.4906000 185.8174467 10
# 9      SEDGE         I  NP  59.2837000 46.5053968 10
# 10     SEDGE         T  CT 103.5752000 64.4158832 20
# 11     SEDGE         T  GH 163.8704500 120.5914018 20
# 12     SEDGE         T  NP  54.4159500 60.0612206 20
# 13     TUSSOCK        I  CT  1.7238400 2.1647741 25
# 14     TUSSOCK        I  GH 15.7673200 68.8852950 25
# 15     TUSSOCK        I  NP  0.8027600 1.3133682 25
# 16     TUSSOCK        T  CT 72.0248800 72.1715944 25
# 17     TUSSOCK        T  GH 40.6221200 52.6456084 25
# 18     TUSSOCK        T  NP 37.2465600 56.1862917 25

```

The `plyr` package can be used with a function from the base packages called `transform()`, which is somewhat similar to `summarise()`, but adds new columns to a copy of the original data frame instead of collapsing it to only unique rows. Compare the above example to the output from the command below.

```
ch4summ2 <- plyr::ddply(gas, c("ecosystem", "morphology", "tmt"), transform,
                         mean = mean(CH4), sd = sd(CH4), n = length(CH4))
head(ch4summ2, 10)

#      ecosystem      date tmt block morphology     CH4    CO2 temp   mean
# 1      HEATH 1993/06/23  CT     1          I  0.434 1146   NA -0.7064
# 2      HEATH 1993/06/23  CT     2          I -0.720  904   NA -0.7064
# 3      HEATH 1993/06/23  CT     3          I  0.186  551   NA -0.7064
# 4      HEATH 1993/07/07  CT     1          I -1.703 1668   NA -0.7064
# 5      HEATH 1993/07/07  CT     2          I -0.326 1039   NA -0.7064
# 6      HEATH 1993/07/07  CT     3          I  0.351  206   NA -0.7064
# 7      HEATH 1993/07/23  CT     1          I -0.654  820   NA -0.7064
# 8      HEATH 1993/07/23  CT     2          I -0.823  325   NA -0.7064
# 9      HEATH 1993/07/23  CT     3          I -0.958  667   NA -0.7064
# 10     HEATH 1993/08/04  CT     1          I -2.168 1579   NA -0.7064
#
#           sd   n
# 1  0.7368068 15
# 2  0.7368068 15
# 3  0.7368068 15
# 4  0.7368068 15
# 5  0.7368068 15
# 6  0.7368068 15
# 7  0.7368068 15
# 8  0.7368068 15
# 9  0.7368068 15
# 10 0.7368068 15
```

The `transform()` function is handy for normalizing data, among other things. See the difference? If it isn't clear to you, try just comparing what `summarise()` and `transform()` do with a subset of the complete data frame.

What if we need to do something more complicated with a data frame, like fit a linear model to each group within a larger data set. In this case, we would need to store the output as a list, so `ddply` is a good choice (of the functions in the base packages, the `by` function does something similar to `ddply`). Let's say we want to fit an ANOVA to data from each ecosystem. Here we have to write our own simple function.

```
lm.lst <- plyr::ddply(gas, "ecosystem",
                      function(x) lm(CH4 ~ (tmt + morphology)^2, data = x))
```

The `lm.lst` object is a list with output from `lm` in each element. Say we want to apply some extractor function to each part of the list that this call returns, and save the output in a data frame. For example, let's extract the model coefficients. We could use either `ldply()` or `llply()`.

```
coefs <- plyr::ldply(lm.lst, .fun = coef)
coefs
```

```

#   ecosystem (Intercept)      tmtGH      tmtNP morphologyT
# 1     HEATH    -0.70640  -1.135475  0.07786667 -0.1605333
# 2     SEDGE    92.96980 154.520800 -33.68610000 10.6054000
# 3    TUSSOCK   1.72384  14.043480 -0.92108000 70.3010400
#   tmtGH:morphologyT tmtNP:morphologyT
# 1          0.6720194      0.7900667
# 2         -94.2255500     -15.4731500
# 3        -45.4462400     -33.8572400

```

But if we wanted output that couldn't be organized as a data frame, `l1ply()` would be the best choice.

```

anovas <- plyr::l1ply(lm.lst, .fun = anova)
length(anovas)

# [1] 3

anovas[1:3]

# $HEATH
# Analysis of Variance Table
#
# Response: CH4
#           Df Sum Sq Mean Sq F value    Pr(>F)
# tmt          2 19.299  9.6496  9.8881 0.0001454 ***
# morphology    1  1.963  1.9635  2.0120 0.1599434
# tmt:morphology 2  2.651  1.3256  1.3583 0.2629600
# Residuals     80 78.071  0.9759
# ---
# Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# $SEDGE
# Analysis of Variance Table
#
# Response: CH4
#           Df Sum Sq Mean Sq F value    Pr(>F)
# tmt          2 287616 143808 15.5751 1.764e-06 ***
# morphology    1 13479   13479  1.4599    0.2303
# tmt:morphology 2  34044   17022  1.8436    0.1646
# Residuals     84 775591   9233
# ---
# Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# $TUSSOCK
# Analysis of Variance Table
#
# Response: CH4
#           Df Sum Sq Mean Sq F value    Pr(>F)
# tmt          2   7967   3984  1.5043   0.22564
# morphology    1  72160  72160 27.2495 6.129e-07 ***
# tmt:morphology 2  13942   6971  2.6323   0.07537 .

```

```
# Residuals      144 381331     2648
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

If these examples don't make sense, here is a simpler one. First, with `l1ply()`,

```
x <- list(a = rnorm(10), b = rnorm(10), c = rnorm(10))
plyr::l1ply(.data = x, .fun = mean)
```

```
# $a
# [1] 0.0203616
#
# $b
# [1] -0.1425961
#
# $c
# [1] 0.09638351
```

Or, if we want a data frame for output, `ldply()`:

```
plyr::ldply(x, mean)

#   .id      V1
# 1   a  0.02036160
# 2   b -0.14259606
# 3   c  0.09638351
```

The `l1ply()` function, and the base package equivalent, `lapply()`, carry out the required operation on each element of the input list sequentially, starting with the first, moving to the second when the first is complete, and so on. Since the operation applied to the second element is independent of the first, this approach will waste processing power on any computer that has a multi-core processor. Alternatively, the `mclapply()` function, from the `parallel` package, works on as many elements as possible at one time, but sending each operation to a different core for processing<sup>114</sup>. Unfortunately, this approach only works in Unix-like operating systems (including Linux and Mac), although there are (less user-friendly) approaches available for Windows. This is an area of R that seems to be currently changing though, so see what you can find online if you are interested.

### 15.3 The dplyr package and the magrittr operator

The `dplyr` package excels at grouped operations on data frames. What am I referring to? Split a data frame by some variable(s), apply an operation, and then put the parts together again. This general approach is useful for many problems in data manipulation.

Let's load the package.

---

<sup>114</sup> Note that `mclapply()` will not be faster than `lapply()` or `l1ply()` in all cases, but only in those where the time required for processing each element, and the number of elements, are sufficiently high.

```
library(dplyr)
```

Looking at our presidents data frame, perhaps we want to calculate the mean time-in-office by party. First let's calculate this variable for each row.

```
pres19$in.office <- pres19$last - pres19$first
```

To avoid unrelated complexity, let's drop the last row which has an NA for our new column.

```
pres19 <- pres19[-nrow(pres19), ]
```

or

```
pres19 <- pres19[!is.na(pres19$in.office), ]
```

or

```
#pres19 <- select(pres19, -n())
```

To summarize these values by party, need to combine two functions in a single call: `group_by()` and `summarise()`.

```
dplyr::summarise(group_by(pres19, party), in.office.mn = mean(in.office))
```

```
# # A tibble: 2 x 2
#   party      in.office.mn
#   <fct>        <dbl>
# 1 Democrat     6.86
# 2 Republican   5.45
```

We could add as many new columns as we like.

```
dplyr::summarise(group_by(pres19, party),
                  in.office.mn = mean(in.office), in.office.min = min(in.office),
                  n.pres = length(unique(name)), first.yr = min(first)
                )

# # A tibble: 2 x 5
#   party      in.office.mn in.office.min n.pres first.yr
#   <fct>        <dbl>        <dbl>    <int>     <dbl>
# 1 Democrat     6.86        2.00       7     1913
# 2 Republican   5.45        2.00      11     1901
```

These examples shouldn't look too strange. We've nested code as is done throughout this book. But the dplyr package comes with an alternative based on a new operator `%>%`. With it, you write code from left to right instead of nesting from inside out. So the two calls below do the same thing.

```

dplyr::summarise(group_by(pres19, party), in.office.mn = mean(in.office))

# # A tibble: 2 x 2
#   party      in.office.mn
#   <fct>        <dbl>
# 1 Democrat     6.86
# 2 Republican   5.45

pres19 %>%
  group_by(party) %>%
  dplyr::summarise(in.office.mn = mean(in.office))

# # A tibble: 2 x 2
#   party      in.office.mn
#   <fct>        <dbl>
# 1 Democrat     6.86
# 2 Republican   5.45

```

Many people find this option easier than nesting. If you prefer nesting, feel free to stick with it.

The `summarise()` function is not the only one that can be used in grouped operations. Let's work with another data set to demonstrate.

```

h2 <- read.csv("../data/biohydrogen.csv")

dfsumm(h2)

#
# 135 rows and 5 columns
# 135 unique rows
#          reactor      date    time      vol conc.h2
# Class       factor      factor factor numeric numeric
# Minimum      G171  9/18/2006 11:12       0      0
# Maximum      G185  9/21/2006  9:26    26.6    35.2
# Mean         G178  9/19/2006 14:00    7.72    17.1
# Unique (excl. NA)    15        4       9      71      59
# Missing values    0        0       0       1      68
# Sorted        TRUE     FALSE  FALSE FALSE FALSE

```

To calculate mean `conc.h2`, we could use `summarise`.

```

dplyr::summarise(group_by(h2, reactor), conc.h2.mn = mean(conc.h2, na.rm = TRUE))

# # A tibble: 15 x 2
#   reactor conc.h2.mn
#   <fct>        <dbl>
# 1 G171        31.9
# 2 G172        16.4
# 3 G173        17.9

```

```

# 4 G174      17.6
# 5 G175      14.8
# 6 G176      13.0
# 7 G177      NaN
# 8 G178      NaN
# 9 G179      NaN
# 10 G180     15.2
# 11 G181     18.2
# 12 G182     16.1
# 13 G183      NaN
# 14 G184      NaN
# 15 G185      NaN

```

or

```

h2 %>%
  group_by(reactor) %>%
  dplyr::summarise(conc.h2.mn = mean(conc.h2, na.rm = TRUE))

# # A tibble: 15 x 2
#   reactor conc.h2.mn
#   <fct>     <dbl>
# 1 G171      31.9
# 2 G172      16.4
# 3 G173      17.9
# 4 G174      17.6
# 5 G175      14.8
# 6 G176      13.0
# 7 G177      NaN
# 8 G178      NaN
# 9 G179      NaN
# 10 G180     15.2
# 11 G181     18.2
# 12 G182     16.1
# 13 G183      NaN
# 14 G184      NaN
# 15 G185      NaN

```

By what if we wanted to calculate something that should be returned with all rows of data? In this case, use `mutate`, which adds columns to a data frame.

```
mutate(group_by(h2, reactor), cumvol = cumsum(vol))
```

```

# # A tibble: 135 x 6
# # Groups:   reactor [15]
#   reactor date      time    vol conc.h2 cumvol
#   <fct>   <fct>    <fct> <dbl>  <dbl>   <dbl>
# 1 G171    9/18/2006 11:12    0      NA      0
# 2 G171    9/18/2006 14:00    0      31.2    0
# 3 G171    9/19/2006 9:26    11.4   35.2   11.4
# 4 G171    9/19/2006 12:51    0      NA      11.4

```

```

# 5 G171 9/19/2006 16:00 0 NA 11.4
# 6 G171 9/19/2006 22:52 0 NA 11.4
# 7 G171 9/20/2006 8:52 0 NA 11.4
# 8 G171 9/20/2006 13:41 2.00 30.9 13.4
# 9 G171 9/21/2006 12:40 0 30.3 13.4
# 10 G172 9/18/2006 11:12 0 NA 0
# # ... with 125 more rows

```

or

```

h2 %>%
  group_by(reactor) %>%
  dplyr::mutate(cumvol = cumsum(vol))

# # A tibble: 135 x 6
# # Groups:   reactor [15]
#   reactor date      time    vol conc.h2 cumvol
#   <fct>   <fct>    <fct> <dbl>   <dbl>   <dbl>
# 1 G171   9/18/2006 11:12  0     NA     0
# 2 G171   9/18/2006 14:00  0     31.2   0
# 3 G171   9/19/2006 9:26   11.4   35.2   11.4
# 4 G171   9/19/2006 12:51  0     NA     11.4
# 5 G171   9/19/2006 16:00  0     NA     11.4
# 6 G171   9/19/2006 22:52  0     NA     11.4
# 7 G171   9/20/2006 8:52   0     NA     11.4
# 8 G171   9/20/2006 13:41  2.00   30.9   13.4
# 9 G171   9/21/2006 12:40  0     30.3   13.4
# 10 G172  9/18/2006 11:12  0     NA     0
# # ... with 125 more rows

```

You may notice that the output from these calls is not displayed like ordinary data frames. This is because they convert output to data tables. If you find this format annoying here is an easy way to switch back.

```

h2 %>%
  group_by(reactor) %>%
  dplyr::mutate(cumvol = cumsum(vol)) %>%
  as.data.frame()

```

The `transmute()` function is similar, but only returns the column(s) used for grouping.

The operations we've covered in this section could all be implemented with the use of `for` loops. Why not just use loops instead of the functions above? There are at least two arguments against loops, but they do not always apply: loops in R are slow, and can require more or more complicated code. But ultimately, loops are probably the most flexible approach to aggregation and summarization problems. And in some cases, they are the best possible approach. The rules for using loops are more flexible.

For example, to carry out an operation like this:

```

eagles <- read.csv("../data/eagles.csv")

dplyr:::summarise(group_by(eagles, site), mean = mean(achlor), sd = sd(achlor),
                  n = length(achlor))

# # A tibble: 40 x 4
#   site           mean      sd     n
#   <fct>        <dbl>    <dbl> <int>
# 1 Ballville     2.72    1.06     4
# 2 Camp Perry   1.32    0.840    6
# 3 Carroll Twp-Camp Perry 2.39    0.0849    2
# 4 Carroll Twp-Toussaint 2.34    NA       1
# 5 Cedar Pt NWR  3.06    0.746    5
# 6 Cr 306       2.99    NA       1
# 7 CR 306       2.28    0.715    8
# 8 Davis Besse-Toussaint 1.82    NA       1
# 9 Ft Seneca    1.34    0.697    7
# 10 Garlo       0.375   0         3
# # ... with 30 more rows

```

we could use this loop:

```

summ <- data.frame(site = sites <- unique(eagles$site), mean = NA, sd = NA, n = NA)

for(s in unique(eagles$site)) {
  eagsub <- subset(eagles, site == s)
  summ[summ$site == s, "mean"] <- mean(eagsub$achlor)
  summ[summ$site == s, "sd"] <- sd(eagsub$achlor)
  summ[summ$site == s, "n"] <- length(eagsub$achlor)
}

summ

#               site      mean      sd n
# 1      Magee Marsh 1.2500000  NA 1
# 2      Magee Marsh 1.8842857 0.48286841 7
# 3  Davis Besse-Toussaint 1.8200000  NA 1
# 4 Carroll Twp-Camp Perry 2.3900000 0.08485281 2
# 5      Ottawa NWR 2.1412500 1.20925855 4
# 6      Camp Perry 1.3250000 0.84020236 6
# 7      Cedar Pt NWR 3.0580000 0.74563396 5
# 8 Carroll Twp-Toussaint 2.3400000  NA 1
# 9      Rossford 2.9900000 0.86267027 2
# 10      Rookery 0.9566667 0.49797256 6
# 11      Mud Ck  2.0757143 0.42972306 7
# 12      Peach Is 0.9725000 0.69685603 4
# 13      CR 306  2.2850000 0.71512237 8
# 14      Gibsonburg 1.8950000 0.95993489 5
# 15      Cr 306  2.9900000  NA 1
# 16      Pickeral Ck 1.2150000 0.40742075 7
# 17 Old Womans Ck 1.1016667 0.77656831 3

```

```

# 18      Ottawa SC 1.9200000      NA 1
# 19      Sandusky Airport 1.6850000 0.36062446 2
# 20      Gonya 1.8550000 0.64346717 2
# 21      Ballville 2.7250000 1.05809577 4
# 22      Metzger-Peach Is 1.4900000      NA 1
# 23      Sass 2.7350000 0.85559921 2
# 24      Ft Seneca 1.3414286 0.69694999 7
# 25      Killdeer 0.4650000 0.20551329 8
# 26      Knobby 0.7825000 0.57629203 2
# 27      Indian Mill-Sycamore 1.3600000      NA 1
# 28      Smithville-Sycamore 1.4450000 0.02121320 2
# 29      Garlo 0.3750000 0.00000000 3
# 30      Mercer 1.0300000 0.05656854 2
# 31      Shendngo 0.4560000 0.18112151 5
# 32      Rockwell 1.2556250 0.66600857 8
# 33      Meander 1.2922222 0.76579253 9
# 34      Mosquito 0.3750000 0.00000000 3
# 35      Kinsman 0.3750000 0.00000000 3
# 36      Shandngo 0.3750000      NA 1
# 37      Snow Lk 0.9787500 0.44317745 4
# 38      Pymatuning 0.3750000 0.00000000 2
# 39      Tri Valley-Wills Ck 3.8200000      NA 1
# 40      Wills Ck 3.6225000 1.56991242 4

```

## 15.4 Reshaping data

The functions in the `reshape2` package are also useful for aggregating and summarizing data. There are two steps involved in data aggregation with these functions—first, data are “melted” using `melt()` (converted to a “long” format), and then `dcast()` is used to aggregate. To demonstrate, let’s continue with the gas emission data.

```

names(gas)

# [1] "ecosystem"    "date"          "tmt"           "block"         "morphology"
# [6] "CH4"           "CO2"           "temp"

```

Let’s use both response variables, `CH4` and `CO2`. To use `melt()`, we can specify either `id.vars` or `measure.vars`. We’ll use `measure.vars` here, since it is fewer columns and less typing.

```

library(reshape2)

#
# Attaching package:  'reshape2'

# The following object is masked from 'package:tidyverse':
#
#     smiths

```

```

gasm <- melt(data = gas, measure.vars = c("CH4", "CO2"))
head(gasm)

#   ecosystem      date tmt block morphology temp variable  value
# 1  TUSSOCK 1993/06/21  CT     1          T  2.56    CH4 -0.204
# 2  TUSSOCK 1993/06/21  CT     1          I   NA    CH4 -0.396
# 3  TUSSOCK 1993/06/21  CT     3          T   NA    CH4 14.049
# 4  TUSSOCK 1993/06/21  CT     3          I   NA    CH4  0.139
# 5  TUSSOCK 1993/06/21  CT     3          T   NA    CH4 21.235
# 6  TUSSOCK 1993/06/21  CT     3          I   NA    CH4 -0.282

dfsumm(gasm)

#
# 652 rows and 8 columns
# 652 unique rows
#           ecosystem      date     tmt   block morphology
# Class      factor       factor factor integer   factor
# Minimum    HEATH 1993/06/21      CT     1          I
# Maximum    TUSSOCK 1993/08/18     NP     4          T
# Mean       SEDGE 1993/07/21     GH     2          T
# Unique (excl. NA) 3          11      3          4          2
# Missing values 0          0      0          0          0
# Sorted      FALSE      FALSE FALSE FALSE FALSE
#           temp variable  value
# Class      numeric   factor numeric
# Minimum    1.98      CH4   -14.7
# Maximum    13.6      CO2   12800
# Mean       5.66      CO2   1040
# Unique (excl. NA) 54          2      633
# Missing values 412         0      0
# Sorted      FALSE      TRUE FALSE

```

Notice the variable and value columns. Now we can use `cast` for aggregation. Say we want mean CO<sub>2</sub> and CH<sub>4</sub> emission for each `ecosystem`, `morphology`, `tmt` combination. The `dcast()` function uses a formula to specify how to organize the output, but it is pretty simple to master. Variables on the left of `~` make up rows, and those to the right of `~` make up columns. The response variable is taken as the vector named `value` by default, but this can be overridden by specifying a `value.var` argument.

```
dcast(data = gasm, formula = variable + ecosystem + morphology ~ tmt, fun.aggregate = mean)
```

```

#   variable ecosystem morphology      CT      GH
# 1      CH4    HEATH          I -0.7064000 -1.841875
# 2      CH4    HEATH          T -0.8669333 -1.330389
# 3      CH4    SEDGE          I 92.9698000 247.490600
# 4      CH4    SEDGE          T 103.5752000 163.870450
# 5      CH4  TUSSOCK          I  1.7238400 15.767320
# 6      CH4  TUSSOCK          T  72.0248800 40.622120
# 7      CO2    HEATH          I 839.4000000 691.375000
# 8      CO2    HEATH          T 741.2666667 527.055556

```

```

# 9      CO2      SEDGE        I  753.100000 961.800000
# 10     CO2      SEDGE        T  852.900000 1073.050000
# 11     CO2    TUSSOCK        I 1820.400000 2103.520000
# 12     CO2    TUSSOCK        T 2235.560000 2623.000000
#
#          NP
# 1   -0.6285333
# 2   0.0010000
# 3   59.2837000
# 4   54.4159500
# 5   0.8027600
# 6   37.2465600
# 7   5320.7333333
# 8   3448.4000000
# 9   1449.3000000
# 10  2165.6000000
# 11  2643.7200000
# 12  3824.1600000

dcast(data = gasm, formula = variable ~ ecosystem + tmt, fun.aggregate = mean)

#   variable    HEATH_CT    HEATH_GH      HEATH_NP SEDGE_CT    SEDGE_GH
# 1     CH4   -0.7866667  -1.487769   -0.3137667 100.0401  191.7438
# 2     CO2  790.3333333 577.615385 4384.5666667 819.6333 1035.9667
#   SEDGE_NP TUSSOCK_CT TUSSOCK_GH TUSSOCK_NP
# 1   56.03853   36.87436  28.19472   19.02466
# 2 1926.83333 2027.98000 2363.26000 3233.94000

```

The `dcast()` function can handle more than two response variables at a time. Unfortunately, it cannot handle multiple functions at once—for these types of operations, `ddply()` is the function of choice.

The `tidyr` package is also used for reshaping data frames. It overlaps with the `reshape2` package, but `tidyr` is simpler and less flexible. Both packages can be used for various types of operations, but here we'll just look at switching from wide to long data frame formats. The purpose of `tidyr` is to make it easy to “tidy” your data, where tidy data are defined by Wickham as having a unique variable in each column and a unique observation in each row. Let's look at some wind speed data for cities in New York State.

```

library(tidyr)

w <- read.csv("../data/ave_wind_us.csv")

summary(w)

#           location       no.yr       jan
# 03017DENVER_ CO    : 1  Min.   : 3.0  Min.   : 2.900
# 03103FLAGSTAFF_ AZ    : 1  1st Qu.:44.5  1st Qu.: 7.800
# 03812ASHEVILLE_ NC    : 1  Median :57.0  Median : 9.500
# 03813MACON_ GA    : 1  Mean    :52.2  Mean    : 9.662
# 03816PADUCAH KY    : 1  3rd Qu.:61.5  3rd Qu.:11.300

```

```

# 03820AUGUSTA_GA : 1 Max. :81.0 Max. :46.200
# (Other) :269
#   feb       mar       apr       may
# Min. : 3.900 Min. : 5.00 Min. : 4.30 Min. : 4.200
# 1st Qu.: 8.200 1st Qu.: 8.60 1st Qu.: 8.50 1st Qu.: 7.900
# Median : 9.600 Median :10.10 Median :10.20 Median : 9.300
# Mean   : 9.849 Mean   :10.35 Mean   :10.32 Mean   : 9.451
# 3rd Qu.:11.200 3rd Qu.:11.70 3rd Qu.:11.70 3rd Qu.:10.700
# Max.   :44.400 Max.   :41.40 Max.   :35.90 Max.   :29.500
#
#   jun       jul       aug       sep
# Min. : 3.900 Min. : 3.600 Min. : 3.3 Min. : 3.200
# 1st Qu.: 7.200 1st Qu.: 6.700 1st Qu.: 6.4 1st Qu.: 6.650
# Median : 8.500 Median : 7.900 Median : 7.5 Median : 8.000
# Mean   : 8.835 Mean   : 8.196 Mean   : 7.9 Mean   : 8.227
# 3rd Qu.:10.100 3rd Qu.: 9.300 3rd Qu.: 9.1 3rd Qu.: 9.400
# Max.   :27.400 Max.   :25.100 Max.   :24.7 Max.   :28.800
#
#   oct       nov       dec       ann
# Min. : 3.200 Min. : 3.600 Min. : 3.000 Min. : 4.100
# 1st Qu.: 6.700 1st Qu.: 7.300 1st Qu.: 7.600 1st Qu.: 7.500
# Median : 8.400 Median : 9.000 Median : 9.300 Median : 8.900
# Mean   : 8.608 Mean   : 9.215 Mean   : 9.377 Mean   : 9.176
# 3rd Qu.:10.000 3rd Qu.:10.750 3rd Qu.:10.900 3rd Qu.:10.450
# Max.   :33.800 Max.   :39.500 Max.   :44.700 Max.   :35.100
#

```

`head(w)`

```

#           location no.yr jan feb mar apr may jun jul aug sep
# 1 13876BIRMINGHAM_AP_AL    65  8.1  8.7  9.0  8.2  6.8  6.0  5.7  5.4  6.3
# 2 03856HUNTSVILLE_AL      41  9.0  9.4  9.7  9.2  7.9  6.8  5.9  5.8  6.7
# 3 13894MOBILE_AL         60 10.1 10.3 10.5 10.1  8.7  7.5  6.9  6.7  7.7
# 4 13895MONTGOMERY_AL     64  7.7  8.2  8.3  7.3  6.1  5.8  5.7  5.2  5.9
# 5 26451ANCHORAGE_AK      55  6.4  6.8  7.1  7.3  8.5  8.4  7.3  6.9  6.7
# 6 25308ANNETTE_AK        44 11.2 11.2 10.4 10.2  8.6  8.3  7.5  7.6  8.3
#
#   oct  nov  dec ann
# 1  6.2  7.2  7.7 7.1
# 2  7.2  8.0  8.9 7.9
# 3  8.0  8.9  9.6 8.8
# 4  5.7  6.5  7.1 6.6
# 5  6.7  6.4  6.3 7.1
# 6 10.8 11.2 11.6 9.8

```

We will work with the first 5 rows only.

```
w <- w[1:5, ]
```

The format of these data is not tidy by Wickham's definition. We have one variable (mean wind speed) spread out over multiple columns. It is easy to change the structure with `gather`.

```
gather(w, key = month, value = wind, jan:dec)
```

```
#          location no.yr ann month wind
# 1 13876BIRMINGHAM AP_AL   65 7.1 jan  8.1
# 2 03856HUNTSVILLE_ AL   41 7.9 jan  9.0
# 3 13894MOBILE_ AL    60 8.8 jan 10.1
# 4 13895MONTGOMERY_ AL   64 6.6 jan  7.7
# 5 26451ANCHORAGE_ AK   55 7.1 jan  6.4
# 6 13876BIRMINGHAM AP_AL   65 7.1 feb  8.7
# 7 03856HUNTSVILLE_ AL   41 7.9 feb  9.4
# 8 13894MOBILE_ AL    60 8.8 feb 10.3
# 9 13895MONTGOMERY_ AL   64 6.6 feb  8.2
# 10 26451ANCHORAGE_ AK   55 7.1 feb  6.8
# 11 13876BIRMINGHAM AP_AL   65 7.1 mar  9.0
# 12 03856HUNTSVILLE_ AL   41 7.9 mar  9.7
# 13 13894MOBILE_ AL    60 8.8 mar 10.5
# 14 13895MONTGOMERY_ AL   64 6.6 mar  8.3
# 15 26451ANCHORAGE_ AK   55 7.1 mar  7.1
# 16 13876BIRMINGHAM AP_AL   65 7.1 apr  8.2
# 17 03856HUNTSVILLE_ AL   41 7.9 apr  9.2
# 18 13894MOBILE_ AL    60 8.8 apr 10.1
# 19 13895MONTGOMERY_ AL   64 6.6 apr  7.3
# 20 26451ANCHORAGE_ AK   55 7.1 apr  7.3
# 21 13876BIRMINGHAM AP_AL   65 7.1 may  6.8
# 22 03856HUNTSVILLE_ AL   41 7.9 may  7.9
# 23 13894MOBILE_ AL    60 8.8 may  8.7
# 24 13895MONTGOMERY_ AL   64 6.6 may  6.1
# 25 26451ANCHORAGE_ AK   55 7.1 may  8.5
# 26 13876BIRMINGHAM AP_AL   65 7.1 jun  6.0
# 27 03856HUNTSVILLE_ AL   41 7.9 jun  6.8
# 28 13894MOBILE_ AL    60 8.8 jun  7.5
# 29 13895MONTGOMERY_ AL   64 6.6 jun  5.8
# 30 26451ANCHORAGE_ AK   55 7.1 jun  8.4
# 31 13876BIRMINGHAM AP_AL   65 7.1 jul  5.7
# 32 03856HUNTSVILLE_ AL   41 7.9 jul  5.9
# 33 13894MOBILE_ AL    60 8.8 jul  6.9
# 34 13895MONTGOMERY_ AL   64 6.6 jul  5.7
# 35 26451ANCHORAGE_ AK   55 7.1 jul  7.3
# 36 13876BIRMINGHAM AP_AL   65 7.1 aug  5.4
# 37 03856HUNTSVILLE_ AL   41 7.9 aug  5.8
# 38 13894MOBILE_ AL    60 8.8 aug  6.7
# 39 13895MONTGOMERY_ AL   64 6.6 aug  5.2
# 40 26451ANCHORAGE_ AK   55 7.1 aug  6.9
# 41 13876BIRMINGHAM AP_AL   65 7.1 sep  6.3
# 42 03856HUNTSVILLE_ AL   41 7.9 sep  6.7
# 43 13894MOBILE_ AL    60 8.8 sep  7.7
# 44 13895MONTGOMERY_ AL   64 6.6 sep  5.9
# 45 26451ANCHORAGE_ AK   55 7.1 sep  6.7
# 46 13876BIRMINGHAM AP_AL   65 7.1 oct  6.2
# 47 03856HUNTSVILLE_ AL   41 7.9 oct  7.2
# 48 13894MOBILE_ AL    60 8.8 oct  8.0
# 49 13895MONTGOMERY_ AL   64 6.6 oct  5.7
# 50 26451ANCHORAGE_ AK   55 7.1 oct  6.7
```

```

# 51 13876BIRMINGHAM AP_AL    65 7.1    nov  7.2
# 52  03856HUNTSVILLE_ AL    41 7.9    nov  8.0
# 53      13894MOBILE_ AL    60 8.8    nov  8.9
# 54  13895MONTGOMERY_ AL    64 6.6    nov  6.5
# 55  26451ANCHORAGE_ AK    55 7.1    nov  6.4
# 56 13876BIRMINGHAM AP_AL    65 7.1    dec   7.7
# 57  03856HUNTSVILLE_ AL    41 7.9    dec   8.9
# 58      13894MOBILE_ AL    60 8.8    dec   9.6
# 59  13895MONTGOMERY_ AL    64 6.6    dec   7.1
# 60  26451ANCHORAGE_ AK    55 7.1    dec   6.3

```

Or, if you prefer, the following gives the same result.

```
w %>%
  gather(key = month, value = wind, jan:dec)
```

The last argument is used to select columns for “gathering”. It has the same flexibility as the `select` function in the `dplyr` package.

To go from long to wide, see the functions `separate()` and `spread()`.

Let’s save a long version to work with.

```
wl <- w %>%
  gather(key = month, value = wind, jan:dec)
```

```
head(wl)
```

```

#           location no.yr ann month wind
# 1 13876BIRMINGHAM AP_AL    65 7.1  jan  8.1
# 2 03856HUNTSVILLE_ AL    41 7.9  jan  9.0
# 3 13894MOBILE_ AL     60 8.8  jan 10.1
# 4 13895MONTGOMERY_ AL    64 6.6  jan  7.7
# 5 26451ANCHORAGE_ AK    55 7.1  jan  6.4
# 6 13876BIRMINGHAM AP_AL    65 7.1  feb  8.7

```

```
spread(wl, month, wind)
```

```

#           location no.yr ann  apr aug dec  feb  jan  jul jun  mar
# 1 03856HUNTSVILLE_ AL    41 7.9 9.2 5.8 8.9  9.4  9.0 5.9 6.8 9.7
# 2 13876BIRMINGHAM AP_AL    65 7.1 8.2 5.4 7.7  8.7  8.1 5.7 6.0 9.0
# 3 13894MOBILE_ AL     60 8.8 10.1 6.7 9.6 10.3 10.1 6.9 7.5 10.5
# 4 13895MONTGOMERY_ AL    64 6.6 7.3 5.2 7.1  8.2  7.7 5.7 5.8 8.3
# 5 26451ANCHORAGE_ AK    55 7.1 7.3 6.9 6.3  6.8  6.4 7.3 8.4 7.1
#   may nov oct sep
# 1 7.9 8.0 7.2 6.7
# 2 6.8 7.2 6.2 6.3
# 3 8.7 8.9 8.0 7.7
# 4 6.1 6.5 5.7 5.9
# 5 8.5 6.4 6.7 6.7

```

The output is nearly the same as our original data frame. But note that the column order is alphabetical, and we have lost a single column that we did not select in the `gather` call.

## 16 Introduction to base graphics

R can be used to produce many different types of plots, and it is possible to customize plots in many different ways. Furthermore, plots can be exported in several different types of formats. If you already have a selection-and-response program that you use for making plots (like Excel or SigmaPlot) and you don't think it is worth the trouble to switch to something new, here are three reasons to consider it: plotting data in the same program that you use for data analysis can save time and effort; plots in R are very nice, and can be customized; and producing plots from code becomes much more efficient than using a sequence of mouse clicks as soon as you need to produce more than one plot. Unfortunately, it takes an investment of time and effort to learn how to use R plotting capabilities well. This introduction should get you started, but if you want to go further, I recommend you check out Paul Murrell's book [11]. Additionally, you should be aware that there is a popular alternative to the base graphics functions in a package called `ggplot2`. Once you understand the syntax of the `ggplot2` functions, they can be used to create sophisticated plots without a lot of effort or code. You can find more information in the next section below. Personally, I generally prefer base graphics, but I find myself using `ggplot2` functions more and more in the past year or so.

### 16.1 Introduction to the `plot` function

The `plot` function is one of the most useful and used graphics function in the base packages. Here, I'll use the `plot` function with a single common type of plot to provide an introduction.

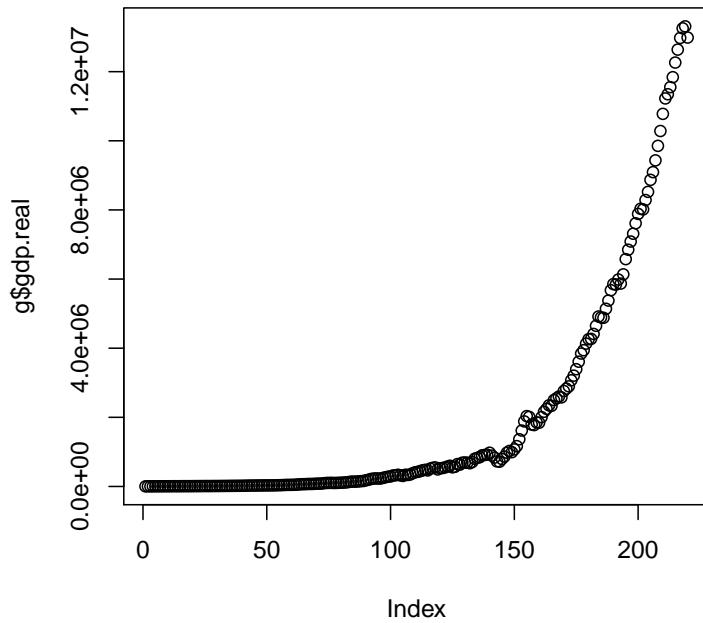
The `plot` function is a generic function, meaning what it does depends on the class of the first argument submitted to it. Let's work with the data in the file `us_gdp.csv`.

```
g <- read.csv("../data/us_gdp.csv")
dfsumm(g)

#
# 220 rows and 3 columns
# 220 unique rows
#           year  gdp.nom gdp.real
# Class      integer  integer   integer
# Minimum    1790      187     4027
# Maximum    2009 14441400 13312200
# Mean       1899      19506    412475
# Unique (excl. NA)  220      220     220
# Missing values  0        0       0
# Sorted      TRUE     FALSE    FALSE
```

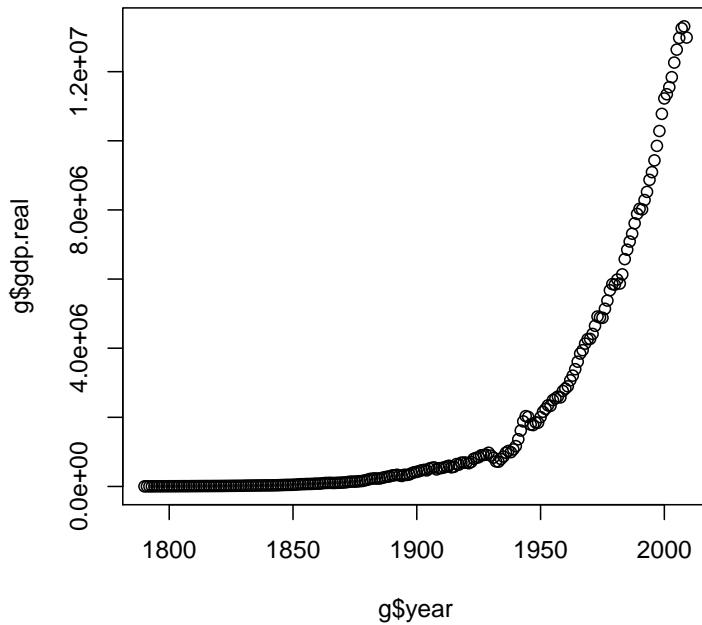
To plot just a single vector, you can use the `plot` function with just the vector for an argument.

```
plot(g$gdp.real)
```



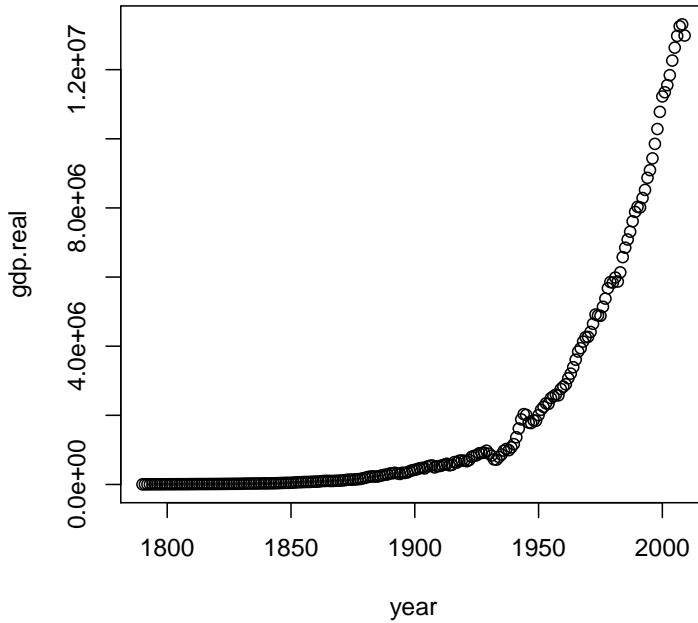
By default, R will plot the values in a vector versus their position (index), as points. To plot one vector versus another, there are two options. The first is more flexible, since the vectors do not have to be within the same data frame.

```
plot(g$year, g$gdp.real)
```



But I recommend the second one when working with data within a single data frame, since it can make specification of other arguments easier. When other arguments are set to columns within the same data frame, you can refer to them directly without using the dollar sign notation.

```
plot(gdp.real ~ year, data = g)
```



The `plot` function is generic, and so it is actually two different *methods* that are being used to produce the two previous plots. The first uses `plot.default` and the second `plot.formula`.

Plots with the default options do not look that great. Arguments you might want to change can be found in the help file for `plot.default`, and can be listed with the `args` function. Although it isn't clear from the help file for `plot.formula`, you can use (almost) all of the optional arguments listed for `plot.default` with either approach.

```
args(plot.default)

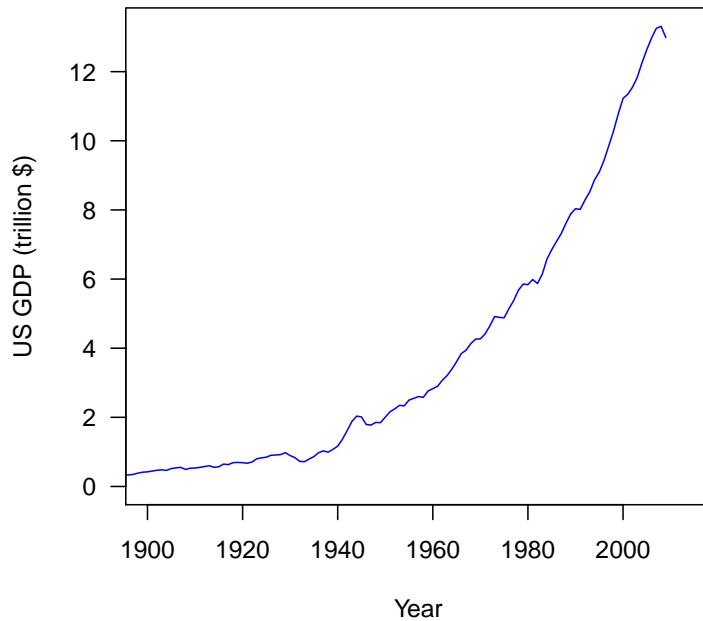
# function (x, y = NULL, type = "p", xlim = NULL, ylim = NULL,
#        log = "", main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
#        ann = par("ann"), axes = TRUE, frame.plot = axes, panel.first = NULL,
#        panel.last = NULL, asp = NA, ...)
# NULL
```

Some of the more useful arguments are listed in the table below.

Argument	Example values	Additional information
col	"red", "blue" 1 through 257	Plotting symbol color. . . .
pch	1 through 25	Plotting symbols. See below for symbols. Or any character.
type	"p" for points "l" for line "b" for both "o" for over "n" for none	What type of plot? Use "n" for blank plot.
xlab and ylab	Any character string, e.g., "Year"	Axis labels.
xlim and ylim	c(0, 100)	Axis limits. Any length-two numeric vector. Larger value first to reverse axis.
log	"x" "y"	Logarithmic axis.
las	0 1	Rotation of axis labels. 0 is default, 1 is a good alternative.

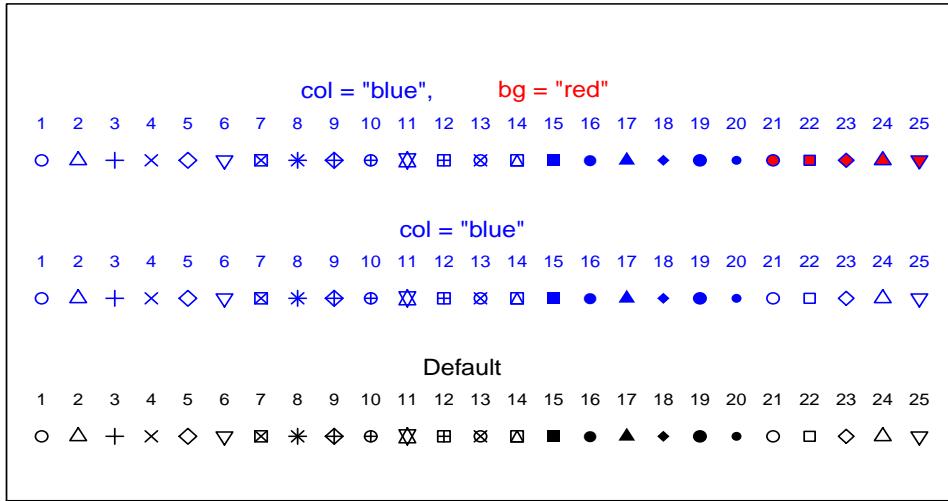
The following call demonstrates the use of some of these arguments.

```
plot(gdp.real/1E6 ~ year, data = g, type = "l", col = "blue", xlab = "Year",
      ylab = "US GDP (trillion $)", xlim = c(1900, 2014), las = 1)
```



And the following code produces a figure that is handy for selecting values for `pch`, `col`, and `bg` (which sets the “background” color for some plotting symbols).

```
plot(1:25, rep(1, 25), pch = 1:25, ylim = c(0, 10), xlab = "", ylab = "", axes = F)
text(1:25, 1.8, as.character(1:25), cex = 0.7)
text(12.5, 2.5, "Default", cex = 0.9)
points(1:25, rep(4, 25), pch = 1:25, col = "blue")
text(1:25, 4.8, as.character(1:25), cex = 0.7, col = "blue")
text(12.5, 5.5, 'col = "blue"', cex = 0.9, col = "blue")
points(1:25, rep(7, 25), pch = 1:25, col = "blue", bg = "red")
text(1:25, 7.8, as.character(1:25), cex = 0.7, col = "blue")
text(10, 8.5, 'col = "blue", ', cex = 0.9, col = "blue")
text(15, 8.5, 'bg = "red"', cex = 0.9, col = "red")
box()
```



There is a large number of graphical parameters that can be set both “globally”, for all plots produced in a session<sup>115</sup>, or “locally”, for an individual plot. To set them “globally”, use the `par` function. The help file for this function lists all the parameters that can be set in this way.

?par

Not every parameter can be set using both approaches, but many can.

## 16.2 Adding data to plots

Several R functions can be used to add data to existing plots, but `points` and `lines` may be the most useful. Let's continue with the GDP data.

```
g <- read.csv("../data/us_gdp.csv")
names(g)

# [1] "year"      "gdp.nom"    "gdp.real"
```

Often, we would like to plot different observations of the same variable with different colors or symbols. With these data, for example, we might like to separate data from before, during, and after the Great Depression. A call to the `plot` function, followed by a series of calls to the `subset` and then `points` or `lines` functions is a good approach for plotting the same variables from several different groups on the same plot. For many groups, or a variable number, a `for` loop can be

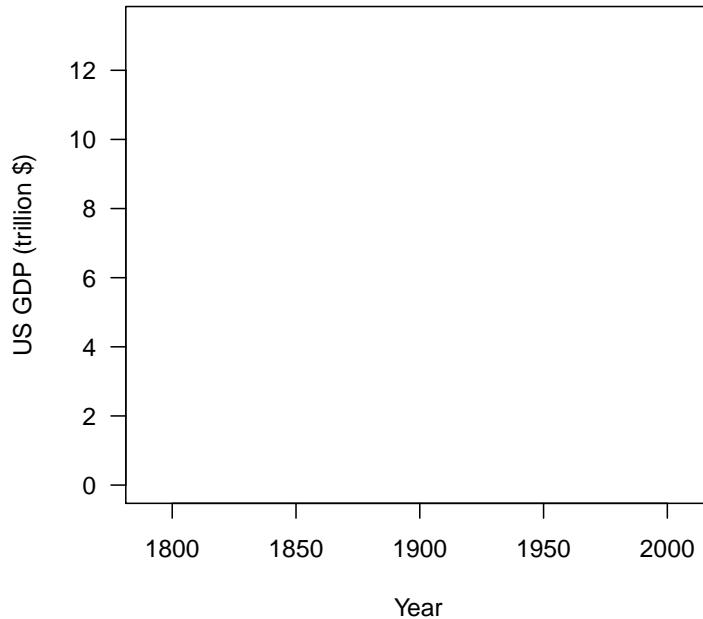
---

<sup>115</sup> Or sent to a single graphics device. More on this topic below.

used. (And, it is important to point out that the ggplot2 package ([ggplot2.org](#)) provides a more automatic alternative to the base graphics functions.)

It is convenient here to first create a blank plot. Doing this with the actual data we want to plot will ensure that axis limits are appropriate.

```
plot(gdp.real/1E6 ~ year, data = g, type = "n", xlab = "Year",
      ylab = "US GDP (trillion $)", las = 1)
```

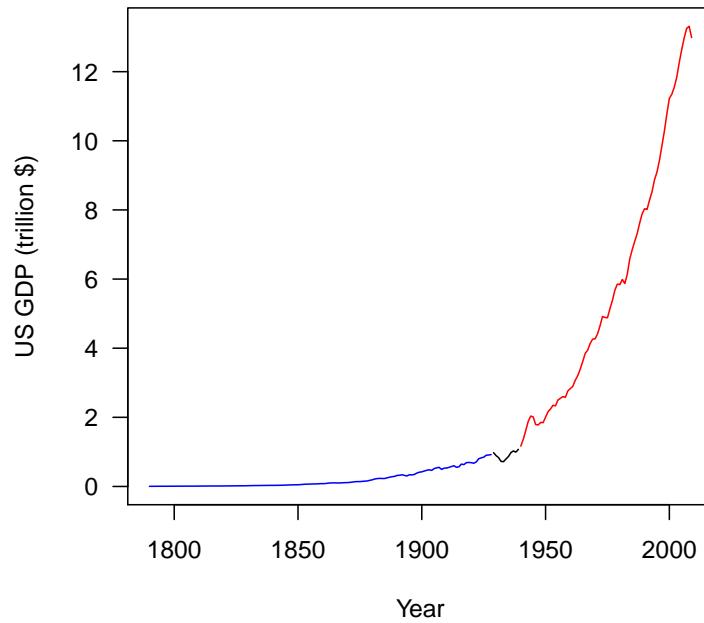


Now let's make the subsets.

```
gpre <- subset(g, year < 1929)
gdep <- subset(g, year > 1928 & year < 1940)
gpost <- subset(g, year > 1939)
```

And plot them using the `lines()` function.

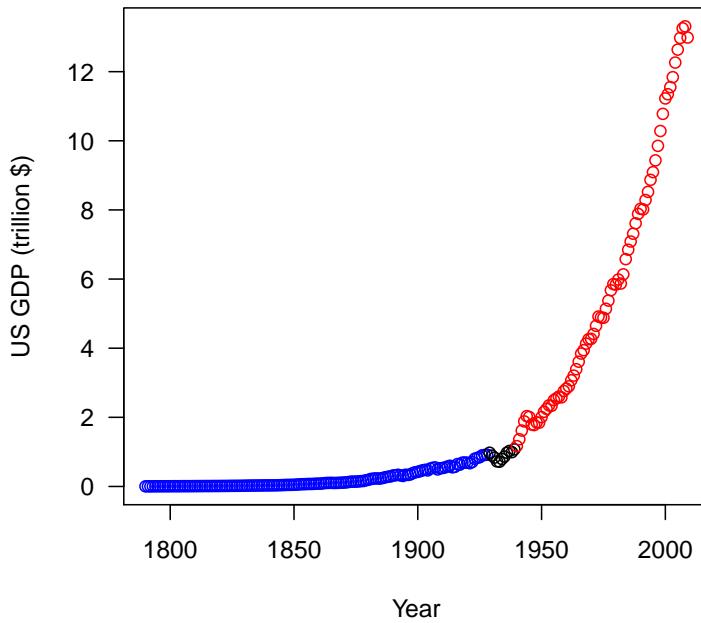
```
lines(gdp.real/1E6 ~ year, data = gpre, col = "blue")
lines(gdp.real/1E6 ~ year, data = gdep, col = "black")
lines(gdp.real/1E6 ~ year, data = gpost, col = "red")
```



Alternatively, `points()` could be used to add points instead of lines.

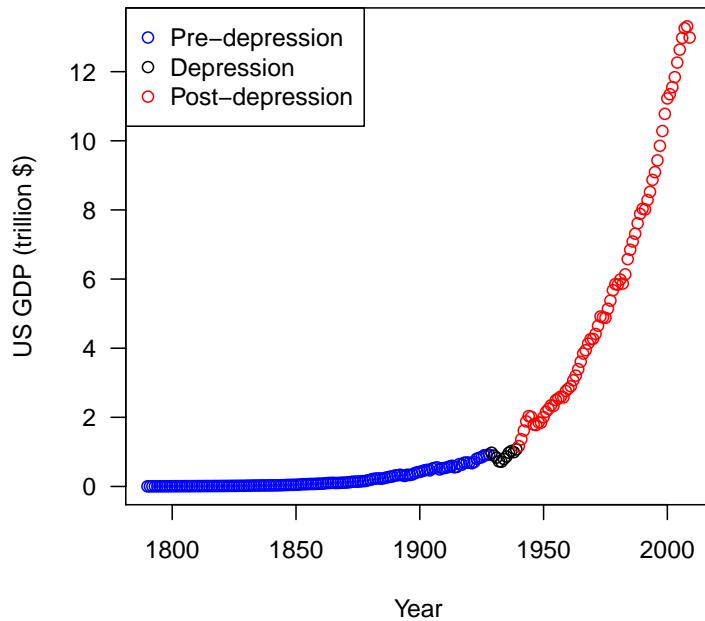
```
plot(gdp.real/1E6 ~ year, data = g, type = "n", xlab = "Year",
      ylab = "US GDP (trillion $)", las = 1)

points(gdp.real/1E6 ~ year, data = gpre, col = "blue")
points(gdp.real/1E6 ~ year, data = gdep, col = "black")
points(gdp.real/1E6 ~ year, data = gpost, col = "red")
```



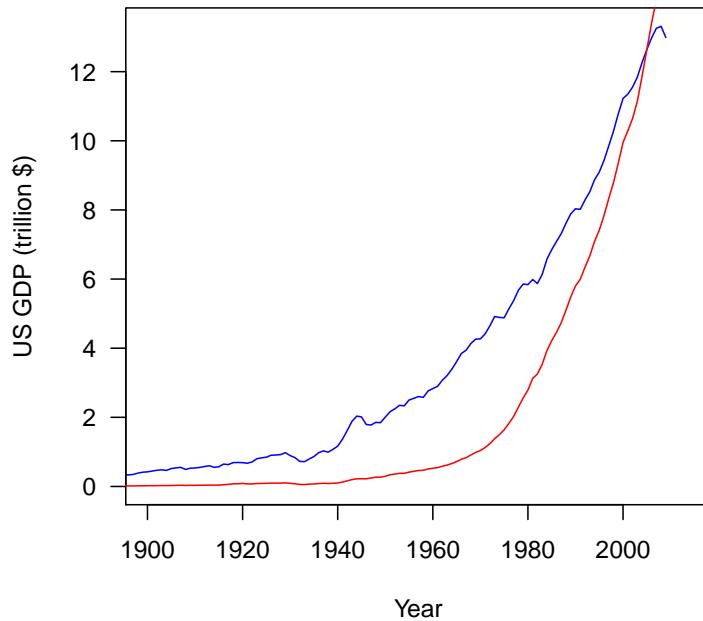
For a plot like this, a legend would be useful. They can be produced (manually) using the `legend()` function. Location can be specified using x, y coordinates through the first two arguments, or by a location keyword, as below.

```
legend("topleft", c("Pre-depression", "Depression", "Post-depression"), pch = 1,  
      col = c("blue", "black", "red"))
```



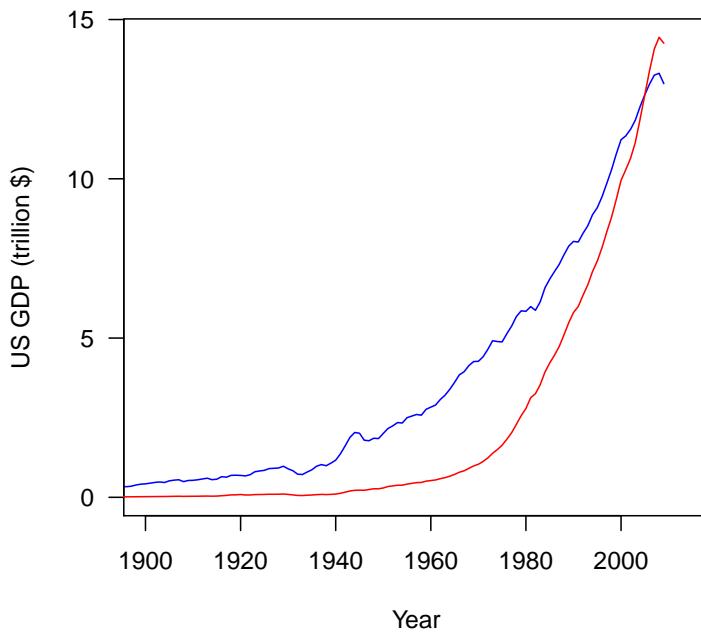
Plotting different types of data on the same plot can be more difficult.

```
plot(gdp.real/1E6 ~ year, data = g, type = "l", col = "blue", xlab = "Year",
      ylab = "US GDP (trillion $)", xlim = c(1900, 2014), las = 1)
lines(gdp.nom/1E6 ~ year, data = g, col = "red")
```



That doesn't quite work, because of an axis limit problem. Here is one approach for fixing it.

```
plot(gdp.real/1E6 ~ year, data = g, type = "l", col = "blue", xlab = "Year",
      ylab = "US GDP (trillion $)", xlim = c(1900, 2014),
      ylim = range(gdp.real, gdp.nom)/1E6, las = 1)
lines(gdp.nom/1E6 ~ year, data = g, col = "red")
```

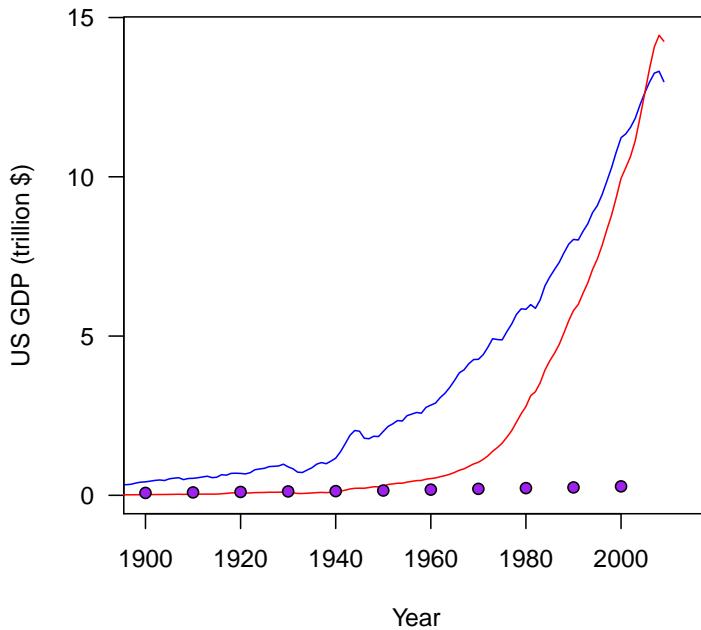


The `points` function is similar. Here, we'll use it to add population data from a different data frame.

```
p <- read.csv("../data/us_pop.csv")
dfsumm(p)

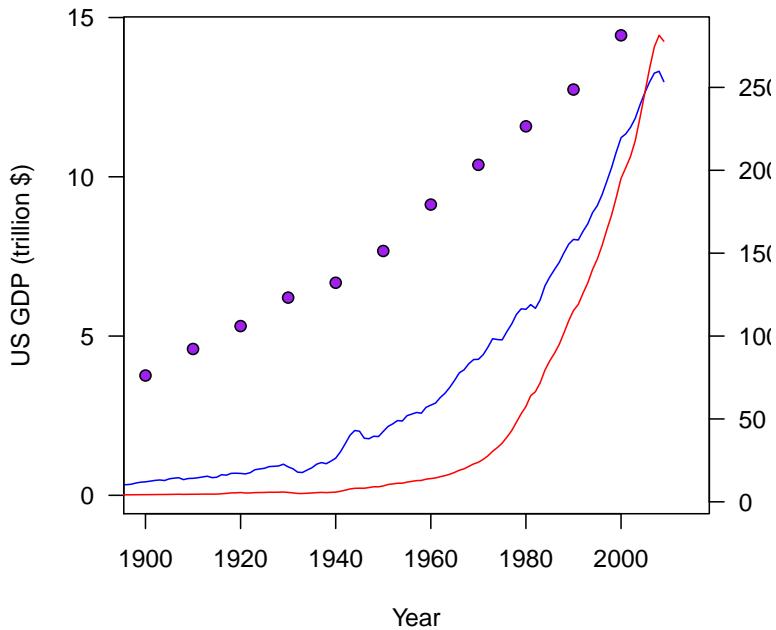
#
# 22 rows and 2 columns
# 22 unique rows
#           year      pop
# Class       integer   integer
# Minimum     1790    3929214
# Maximum     2000  281421906
# Mean        1890   62979766
# Unique (excl. NA)  22      22
# Missing values  0       0
# Sorted       TRUE     TRUE

points(pop/1E9 ~ year, data = p, pch = 21, bg = "purple")
```



It might make more sense to show a different axis for population in this example. There are at least two ways to do this with the base graphics functions in R, and both are slightly tricky. I'll show the easier way, which is to tell R that it has moved onto a new blank plotting page when it really hasn't.

```
plot(gdp.real/1E6 ~ year, data = g, type = "l", col = "blue", xlab = "Year",
      ylab = "US GDP (trillion $)", xlim = c(1900, 2014),
      ylim = range(gdp.real, gdp.nom)/1E6, las = 1)
lines(gdp.nom/1E6 ~ year, data = g, col = "red")
par(new = TRUE)
plot(NULL, type = "n", xlab = "", ylab = "", xlim = c(1900, 2014), ylim = range(p$pop/1E6),
      axes = FALSE)
points(pop/1E6 ~ year, data = p, pch = 21, bg = "purple")
axis(side = 4, las = 1)
```

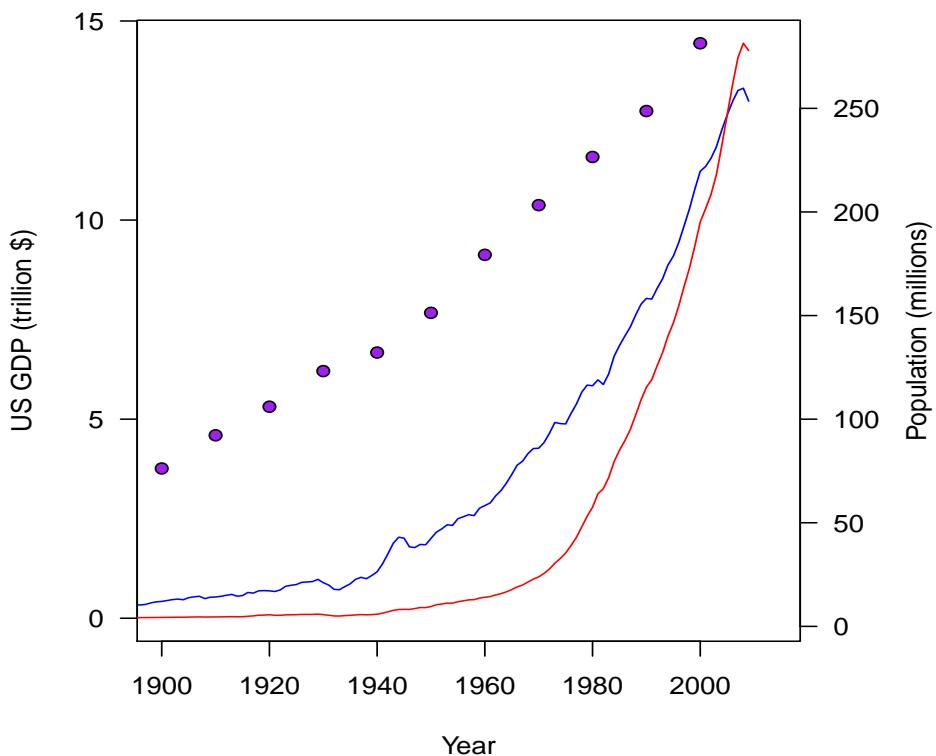


I used the `axis` function to add an axis to the side 4, which is the right “y” axis. Here, there is no reason to use points anymore. The following code would produce an identical plot.

```
plot(gdp.real/1E6 ~ year, data = g, type = "l", col = "blue", xlab = "Year",
      ylab = "US GDP (trillion $)", xlim = c(1900, 2014),
      ylim = range(gdp.real, gdp.nom)/1E6, las = 1)
lines(gdp.nom/1E6 ~ year, data = g, col = "red")
par(new = TRUE)
plot(pop/1E6 ~ year, data = p, type = "p", pch = 21, bg = "purple", xlab = "", ylab = "",
      xlim = c(1900, 2014), ylim = range(p$pop/1E6), axes = FALSE)
axis(4, las = 1)
```

But now we see another problem—there isn’t enough room for the new axis labels. This can be fixed by changing the size of the margins around the plot. And we can use `mtext` to easily add a label.

```
par(mar = c(4, 4, 1, 5))
plot(gdp.real/1E6 ~ year, data = g, type = "l", col = "blue", xlab = "Year",
      ylab = "US GDP (trillion $)", xlim = c(1900, 2014),
      ylim = range(gdp.real, gdp.nom)/1E6, las = 1)
lines(gdp.nom/1E6 ~ year, data = g, col = "red")
par(new = TRUE)
plot(pop/1E6 ~ year, data = p, type = "p", pch = 21, bg = "purple", xlab = "", ylab = "",
      xlim = c(1900, 2014), ylim = range(p$pop/1E6), axes = FALSE)
axis(4, las = 1)
mtext("Population (millions)", side = 4, line = 3)
```



### 16.3 Exporting graphics

The basic approach to producing graphics files in R is this:

1. Open a “graphics device”
2. Produce plot(s)
3. Close graphics device

The characteristics of the file you will produce are set when you open the appropriate graphics device. For example, to put our GDP plot in a 5 inch by 5 inch pdf file, we could use the following commands.

```
pdf("gdp_plot.pdf", height = 5, width = 5)
plot(gdp.real/1E6 ~ year, data = g, type = "l", col = "blue", xlab = "Year",
      ylab = "US GDP (trillion $)", xlim = c(1900, 2014), las = 1)
dev.off()
```

Really, it is pretty simple. Several other types of graphics files can be created with R, including postscript files (use the `postscript` function) and a few bitmap formats, including jpeg (`jpeg` function) and png (`png` function)<sup>116</sup>. For Microsoft Office products (Word and PowerPoint), the bitmap formats work the best, and for L<sup>A</sup>T<sub>E</sub>X, pdf or postscript files are best (and generally look nicer than bitmap files). There are differences among these functions—see the help files for each for more information. Most of the arguments can be left at their default values in most cases. The

<sup>116</sup> See the help file for `Devices` for a complete list.

`pointsize` argument is one exception. Trial-and-error adjustment is useful for sizing plot elements. For bitmap formats, the number of pixels or the resolution can be set. If you use Microsoft Word or other “what you see is what you get” (WYSIWYG) word processing software, a fixed resolution of 300 dots per inch (dpi) is generally sufficient. For PowerPoint or similar programs, the best approach is to set the size of the image as number of height and width pixels to a fraction of the resolution of the projector that will ultimately be used, usually 1024 by 768 or 1920 by 1080. Any resolution higher than that of the projector only increases file size, and does not improve the quality of the displayed image.

It is common to set graphical parameters using `par` after opening a graphics device. The settings are applied to only the file(s) created, and not all following plots. For example, to produce a pdf with four plots on one page, the following commands could be used.

```
pdf("plots.pdf", height = 8.5, width = 11)
par(mfrow = c(2, 2))
plot(gdp.real/1E6 ~ year, data = g, type = "l", col = "blue", xlab = "Year",
      ylab = "US real GDP (trillion $)", xlim = c(1900, 2014), las = 1)
plot(gdp.real/1E6 ~ year, data = g, type = "l", col = "red", xlab = "Year",
      ylab = "US nominal GDP (trillion $)", xlim = c(1900, 2014), las = 1)
plot(pop/1E6 ~ year, data = p, type = "l", col = "black", xlab = "Year",
      ylab = "US population (millions)", las = 1)
plot(pop/1E6 ~ year, data = p, type = "l", col = "black", xlab = "Year",
      ylab = "US population (millions)", xlim = c(1900, 2014), las = 1)
dev.off()
```

## 17 ggplot2 graphics

The `ggplot2` package was written to address shortcomings in the base graphic functions and the `lattice` package. It is one currently of the most popular add-on R packages, and is worth learning. In this section I will present a brief summary only. For more details and some newer information, I strongly recommend the documentation available through the tidyverse website (<https://ggplot2.tidyverse.org/>) or the “Data visualization” chapter (3) in [8] (<https://r4ds.had.co.nz/data-visualisation.html>). From <https://ggplot2.tidyverse.org/> you can download a compact “cheatsheet” that should be helpful at the start and even after you become more familiar with the package (or download directly here: <https://github.com/rstudio/cheatsheets/blob/master/data-visualization-2.1.pdf>).

### 17.1 Components of ggplot2 graphics

Hadley Wickham, the author of `ggplot2`, attempted to retain the good parts of earlier plotting functions, including features of the `lattice` package. The package follows a consistent approach to describing and building graphics called “the grammar of graphics”. As described in Wickham [26], this leads to the use of seven graphics components in `ggplot2`:

1. Data, and the “mapping” that relates data to visual (or aesthetic) attributes that you can see. (The `data` and `mapping` argument and `aes` function.)
2. Geometric objects, which include points, lines, bars, and polygons. (The `geom` argument and `geom_xxxx` functions)
3. Statistical transformations, such as binning or counting. (The `stat` argument and `stat_xxxx` functions.)
4. Scales, including axes and legends, which are the inverse of mapping, and let the viewer determine data values based on aesthetic attributes. (The `scale_xxxx` functions.)
5. A coordinate system, which is usually Cartesian. (The `coord_xxxx` function.)
6. A faceting system, which can be used to plot subsets of data using multiple plots (called conditioning, latticing, or trellising). (The `facets` argument and `facet_xxxx` functions.)

Overwhelmed? It isn’t really that bad. The most important parts (i.e., in most cases, the only parts you have to think about) are data and geometry. However, the design of `ggplot2` functions does seem a bit less intuitive than the base R plotting functions. The advantage is that the `ggplot2` functions will produce refined and sophisticated plots with less code (and, ideally, with less work for the users). With the `plot` function, anything other than a simple one-panel plot with a single data series will probably take multiple lines of code and sometimes a loop. With the `ggplot2` functions, something as complex as a multi-panel plot with multiple data series, observations, model predictions, and a legend could be produced with a single command. The `ggplot2` functions take some of the programming away from the user (the legend design is a good example of this). If the `ggplot2` functions get it wrong, the defaults can be changed, but perhaps with more effort than it takes with the base functions.

As mentioned above, two different functions in `ggplot2` can be used for producing plots: `qplot` and `ggplot`. The `qplot` function, which stands for quick plot, is simpler than `ggplot`, but for the most control and flexibility, the `ggplot` function is the only choice. The `qplot` function actually calls `ggplot` internally. I recommend going straight for `ggplot`.

## 17.2 An example

Let's use `ggplot` for a single example.

```
library(ggplot2)

crabs <- read.csv("../data/crabs.csv")

summary(crabs)

#      sp      temp     sex      resp
# Min.   :1   high:24   F:36   Min.   :1.000
# 1st Qu.:1   low :24   M:36   1st Qu.:1.900
# Median :2   med :24           Median :2.300
# Mean   :2                   Mean   :2.325
# 3rd Qu.:3                   3rd Qu.:2.900
# Max.   :3                   Max.   :3.600

dfsumm(crabs)

#
# 72 rows and 4 columns
# 69 unique rows
#          sp      temp     sex      resp
# Class      integer factor factor numeric
# Minimum        1    high      F      1
# Maximum        3    med       M     3.6
# Mean          2    low       M     2.33
# Unique (excl. NA) 3      3      2     24
# Missing values 0      0      0      0
# Sorted         TRUE FALSE FALSE FALSE

table(crabs$sp, crabs$sex, crabs$temp)

# , , = high
#
#
#      F M
# 1 4 4
# 2 4 4
# 3 4 4
#
# , , = low
#
#
#      F M
# 1 4 4
# 2 4 4
# 3 4 4
```

```

# , , = med
#
#
# F M
# 1 4 4
# 2 4 4
# 3 4 4

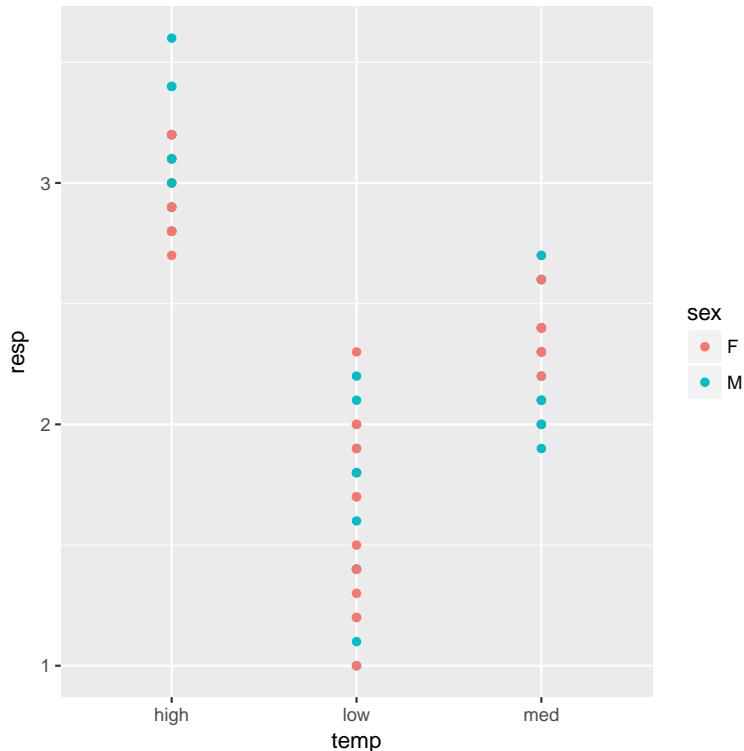
```

We should make `sp` a factor.

```
crabs$sp <- factor(crabs$sp)
```

To start out, let's plot respiration rate `resp` versus temperature `temp`, and use a different color for the two sexes.

```
ggplot(data = crabs, aes(x = temp, y = resp, colour = sex)) + geom_point()
```



We have “mapped” `temp` to the x position, `resp` to the y position, and color to `sex`. We need to fix the order of the levels of `temp`.

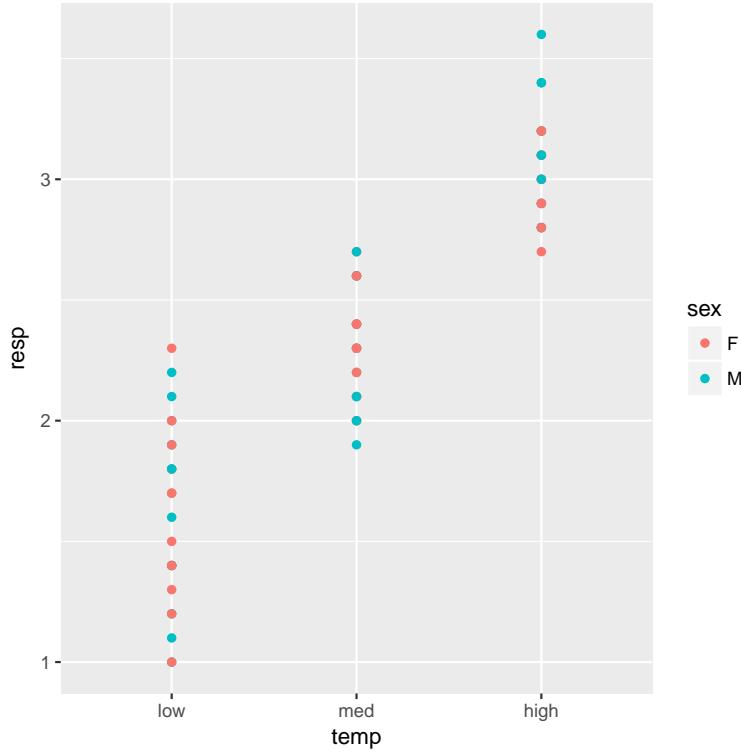
```
levels(crabs$temp)
```

```
# [1] "high" "low"  "med"
```

```
crabs$temp <- factor(crabs$temp, levels = c("low", "med", "high"))
levels(crabs$temp)
```

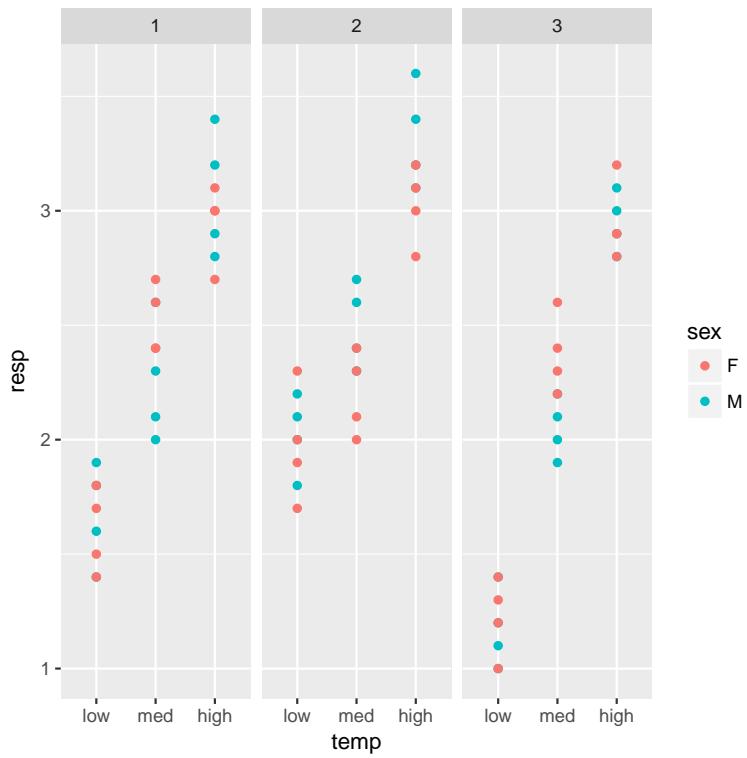
```
# [1] "low"  "med"  "high"
```

```
ggplot(data = crabs, aes(x = temp, y = resp, colour = sex)) + geom_point()
```



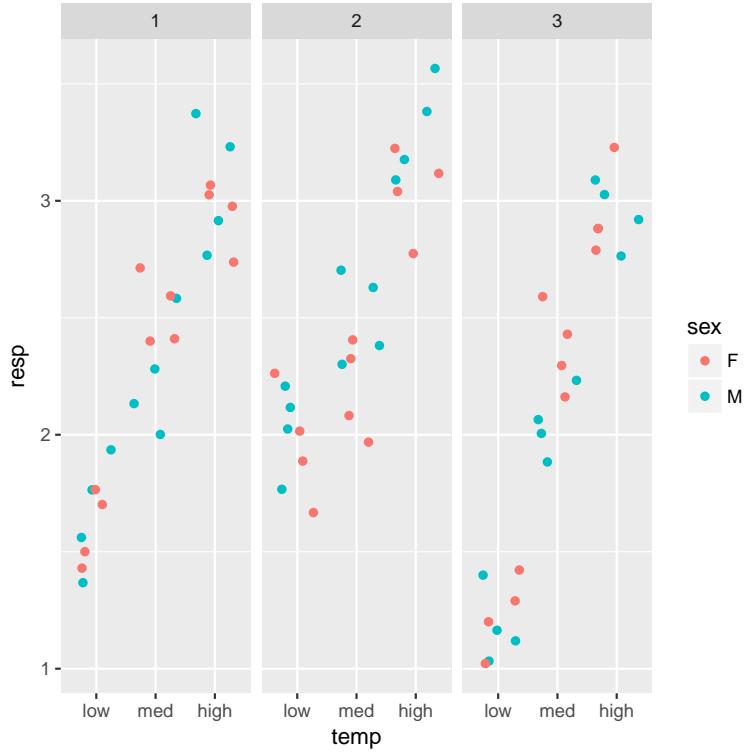
And we are ignoring the `sp` variable. We could add it using facetting.

```
ggplot(crabs, aes(temp, resp, colour = sex)) + geom_point() + facet_wrap(~ sp)
```

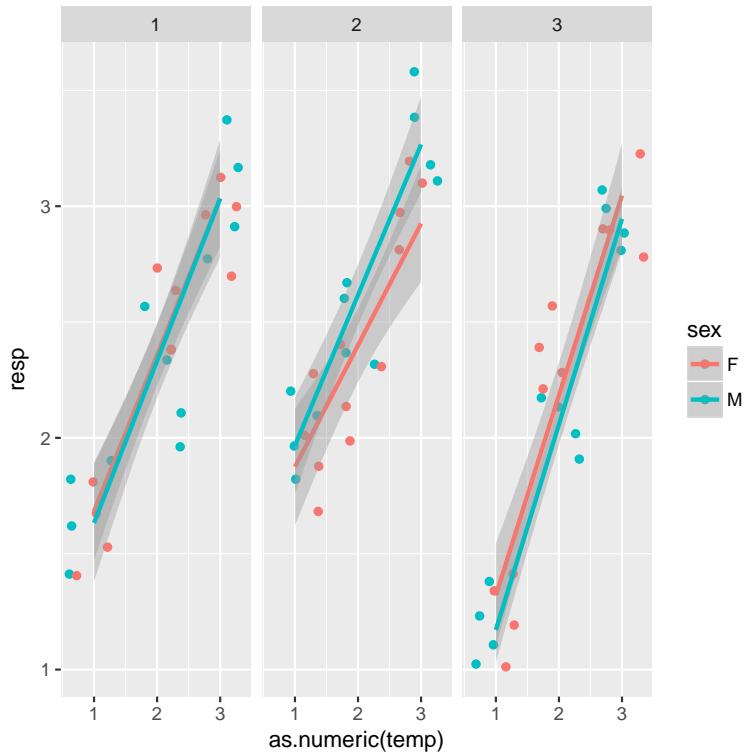


All that remains are a few tweaks. There must be some overlapping points. We can address this by replacing `geom_point` with `geom_jitter`.

```
ggplot(crabs, aes(temp, resp, colour = sex)) + geom_jitter() + facet_wrap(~ sp)
```

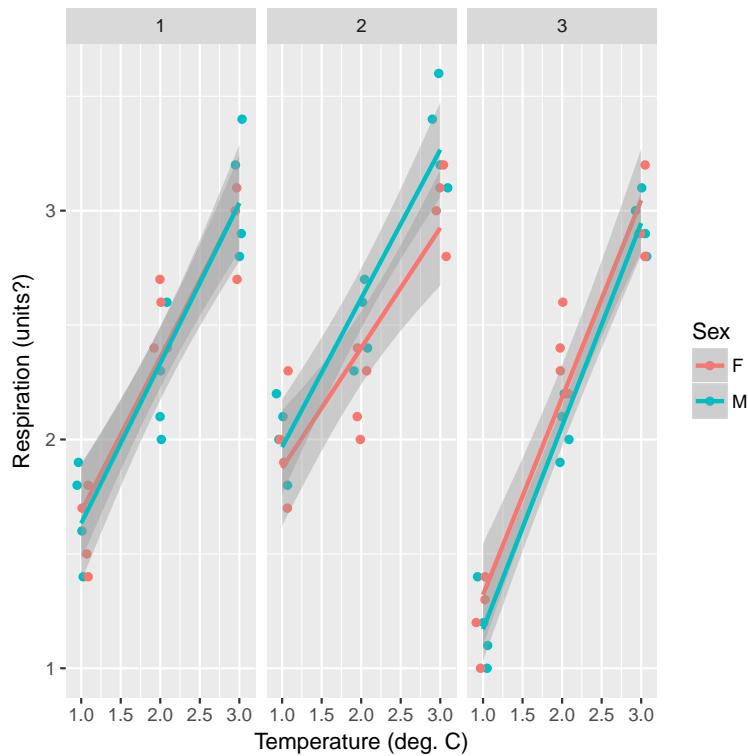


```
ggplot(crabs, aes(as.numeric(temp), resp, colour = sex)) +
  geom_jitter() + facet_wrap(~ sp) + geom_smooth(method = "lm")
```



Lastly, let's improve the labels a bit.

```
ggplot(crabs, aes(as.numeric(temp), resp, colour = sex)) +
  geom_jitter(width = 0.1, height = 0) + facet_wrap(~ sp) +
  geom_smooth(method = "lm") +
  labs(x = "Temperature (deg. C)", y = "Respiration (units?)",
       colour = "Sex")
```



### 17.3 Exporting ggplot2 graphics

The approach described above (Section 16) can be used to export ggplots plots as well. However, the `ggsave` function is easier. By default, it will save the last plot you created.

```
ggsave("../plots/crabs.pdf")
```

```
# Saving 5.2 x 5.2 in image
```

There are several arguments that can be used to tweak the output. But the function does a decent job in setting the values on its own.

```
ggsave("../plots/crabs.png")
```

```
# Saving 5.2 x 5.2 in image
```

## 18 $t$ tests

R can be used for one-sample, two-sample, and paired  $t$  tests.

### 18.1 One-sample

To demonstrate one-sample  $t$  tests, we will use some data from Wilcock et al. [27] on the measurement of dissolved oxygen (DO) in reference waters using a chemical method called the Winkler method. The objective here is to see if there is a bias in the laboratories' determinations.

```
dox <- read.csv("../data/do_methods_1.csv")
summary(dox)

#      lab          method       ref        result
# Min.   : 1.00   winkler:36   Min.   :1.2   Min.   :1.000
# 1st Qu.:11.75           1st Qu.:1.2   1st Qu.:1.200
# Median :23.00           Median :1.2   Median :1.310
# Mean   :22.58           Mean   :1.2   Mean   :1.473
# 3rd Qu.:33.25           3rd Qu.:1.2   3rd Qu.:1.692
# Max.   :45.00           Max.   :1.2   Max.   :2.800
#      X
# Mode:logical
# NA's:36
#
#
#
#
```

```
t.test(dox$result, mu = 1.2)

#
# One Sample t-test
#
# data: dox$result
# t = 3.8052, df = 35, p-value = 0.0005463
# alternative hypothesis: true mean is not equal to 1.2
# 95 percent confidence interval:
#  1.327248 1.618307
# sample estimates:
# mean of x
# 1.472778
```

This looks like a pretty clear bias.

### 18.2 Two-sample

To demonstrate two-sample tests, let's use a data set included with the ISwR package on energy expenditure in women as a function of their body mass.

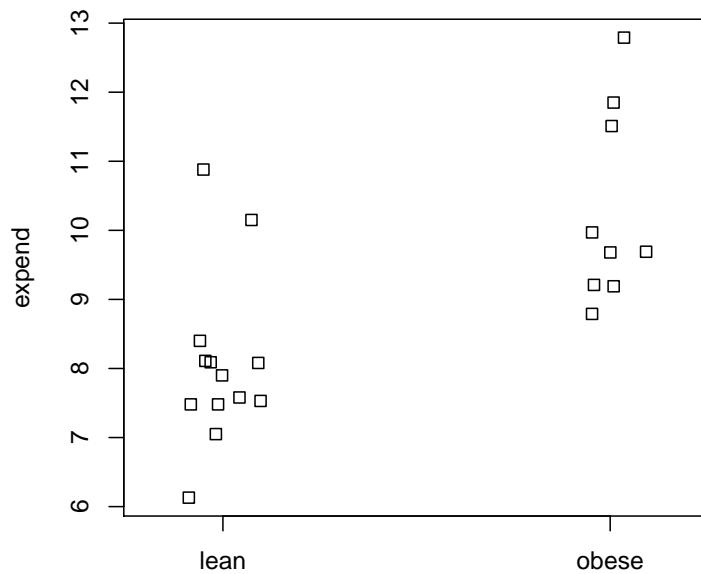
```
library(ISwR)
data(energy)
```

```
?energy
```

```
#energy <- energy
summary(energy)

#      expend      stature
#  Min.   : 6.130   lean  :13
#  1st Qu.: 7.660   obese: 9
#  Median  : 8.595
#  Mean   : 8.979
#  3rd Qu.: 9.900
#  Max.   :12.790

stripchart(expend ~ stature, data = energy, method = "jitter", vertical = TRUE)
```



```
t.test(expend ~ stature, data = energy)
```

```
#
#  Welch Two Sample t-test
#
# data:  expend by stature
```

```

# t = -3.8555, df = 15.919, p-value = 0.001411
# alternative hypothesis: true difference in means is not equal to 0
# 95 percent confidence interval:
# -3.459167 -1.004081
# sample estimates:
# mean in group lean mean in group obese
# 8.066154 10.297778

```

Note that R uses the Welch procedure to calculate the standard error of the difference. You can use the classical approach as well.

```
t.test(expend ~ stature, data = energy, var.equal = TRUE)
```

```

#
# Two Sample t-test
#
# data: expend by stature
# t = -3.9456, df = 20, p-value = 0.000799
# alternative hypothesis: true difference in means is not equal to 0
# 95 percent confidence interval:
# -3.411451 -1.051796
# sample estimates:
# mean in group lean mean in group obese
# 8.066154 10.297778

```

For a paired *t* test, the same function can be used. For this example, let's use some additional DO measurement data. In this data set, we have electrode and Winkler results for several laboratories [27].

```

dox2 <- read.csv("../data/do_methods_2.csv")
summary(dox2)

#      lab          ref         wink        elect
# Min.   : 4.00   Min.   :1.2   Min.   :1.000   Min.   :1.300
# 1st Qu.:21.50  1st Qu.:1.2   1st Qu.:1.125  1st Qu.:1.425
# Median :28.00  Median :1.2   Median :1.300  Median :1.700
# Mean   :25.73  Mean   :1.2   Mean   :1.400  Mean   :1.695
# 3rd Qu.:33.50  3rd Qu.:1.2   3rd Qu.:1.375  3rd Qu.:1.850
# Max.   :42.00  Max.   :1.2   Max.   :2.300  Max.   :2.300

t.test(dox2$wink, dox2$elect, paired = T)

#
# Paired t-test
#
# data: dox2$wink and dox2$elect
# t = -3.4924, df = 10, p-value = 0.0058
# alternative hypothesis: true difference in means is not equal to 0
# 95 percent confidence interval:
# -0.4839533 -0.1069558

```

```
# sample estimates:  
# mean of the differences  
# -0.2954545
```

The output from statistical tests in R are contained in objects with specific classes that depend on the type of test carried out. For example,

```
tt1 <- t.test(dox$result, mu = 1.2)  
class(tt1)  
  
# [1] "htest"
```

Objects of class `htest` are lists<sup>117</sup> that contain information on the input and output of a t test. The output that can be extracted from `htest` objects can be found in the help file associated with `t.test`:

<code>statistic</code>	the value of the t-statistic
<code>parameter</code>	the degrees of freedom for the t-statistic
<code>p.value</code>	the p-value for the test
<code>conf.int</code>	a confidence interval for the mean appropriate
<code>estimate</code>	the estimated mean or difference in means
<code>null.value</code>	the specified hypothesized value of the mean or mean difference
<code>alternative</code>	a character string describing the alternative hypothesis
<code>method</code>	a character string indicating what type of t-test was performed
<code>data.name</code>	a character string giving the name(s) of the data

To demonstrate, say we want *P*-value and the confidence interval:

```
tt1$conf.int
```

```
# [1] 1.327248 1.618307  
# attr("conf.level")  
# [1] 0.95
```

```
tt1$p.value
```

```
# [1] 0.0005463255
```

To find out what elements are present in a statistical object, you can use the `names` function, for example:

```
names(tt1)  
  
# [1] "statistic"    "parameter"    "p.value"      "conf.int"  
# [5] "estimate"     "null.value"   "alternative"  "method"  
# [9] "data.name"
```

<sup>117</sup> As are the output from at least most and maybe all statistical functions in R.

Results can be extracted from other statistical tests in a similar way, although the `summary` function must be used in some cases. Examples are shown in following sections.

## 19 Linear regression

### 19.1 The lm function, model formulas, and statistical output

In R, several classical statistical models can be implemented using one function: `lm` (for linear model). The `lm` function can be used for simple and multiple linear regression, analysis of variance (ANOVA), and analysis of covariance (ANCOVA).

The arguments for `lm` are

```
args(lm)

# function (formula, data, subset, weights, na.action, method = "qr",
#   model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
#   contrasts = NULL, offset, ...)
# NULL
```

The first argument, `formula`, is where you specify the basic structure of the statistical model. This approach is used in other R functions as well, such as `glm`, `gam`, and others. Venables et al. [25, pp. 54–55] has a useful list of example formulas—some examples are repeated below. In these examples, the variables `x`, `y`, and `z` are continuous, and `A`, `B`, and `C` are factors.

<code>y ~ x</code>	Simple linear regression of <code>y</code> on <code>x</code>
<code>y ~ x + z</code>	Multiple regression of <code>y</code> on <code>x</code> and <code>z</code>
<code>y ~ poly(x, 2)</code>	Second order orthogonal polynomial regression
<code>y ~ x + I(x^2)</code>	Second order polynomial regression
<code>y ~ A</code>	Single factor ANOVA
<code>y ~ A + B</code>	Two-factor ANOVA
<code>y ~ A + B + A:B</code>	Two-factor ANOVA with interaction
<code>y ~ A*B</code>	Two-factor ANOVA with interaction
<code>y ~ (A + B + C)^2</code>	Three-factor ANOVA with all first-order interactions
<code>y ~ (A + B + C)^2 - B:C</code>	As above but without <code>B:C</code> interaction
<code>y ~ A + x</code>	ANCOVA

There is some similarity between the statistical output in R and in other statistical software programs. However, by default, R usually gives only basic output. More detailed output can be retrieved with the `summary` function. For specific statistics, you can use “extractor” functions, such as `coef` or `deviance`. An additional difference is that statistical output from R is usually saved as an object that can later be queried or printed.

Output from the `lm` function is of the class `lm`, and both default output and specialized output from extractor functions can be assigned to objects (this is of course true for other model objects as well). This quality is very handy when writing code that uses the results of statistical models in further calculations or in compiling summaries.

### 19.2 Linear regression

To demonstrate simple linear regression in R, let’s read in a data set on the hardness of some Australian woods.

```

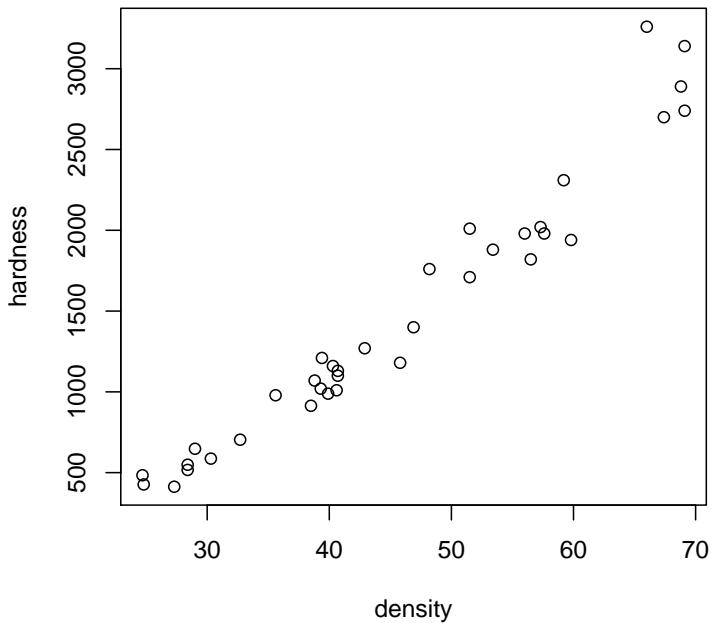
hard <- read.csv("../data/janka.csv")
dfsumm(hard)

#
# 36 rows and 2 columns
# 36 unique rows
#          density hardness
# Class      numeric   integer
# Minimum       24.7      413
# Maximum       69.1     3260
# Mean         45.7     1180
# Unique (excl. NA)    32      35
# Missing values      0      0
# Sorted        TRUE     FALSE

```

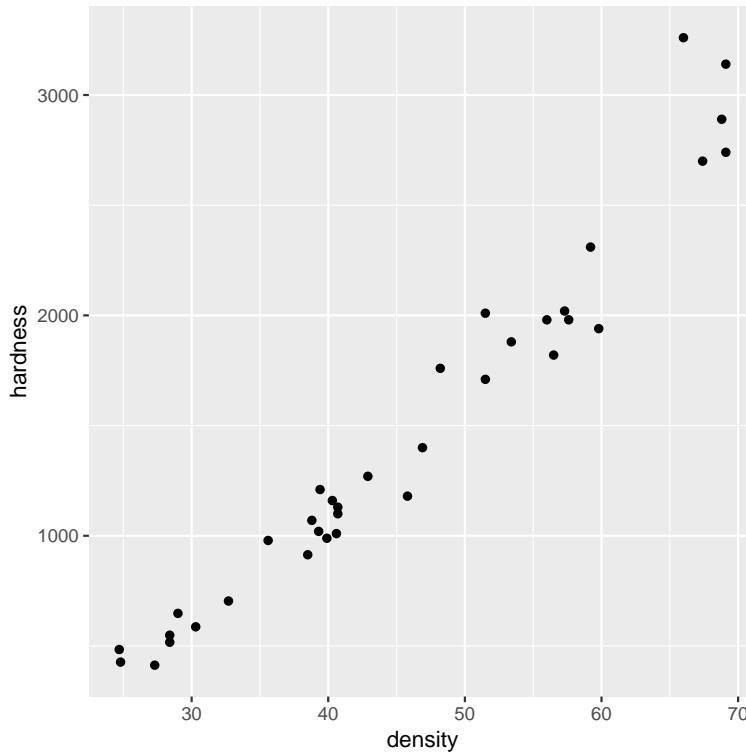
Let's start out by seeing what the data look like.

```
plot(hardness ~ density, data = hard)
```



For practice, we could use `ggplot` as well.

```
ggplot(hard, aes(density, hardness)) + geom_point()
```



Here `plot` is easier.

This looks like a pretty clear relationship. We might be interested in doing two things with these data: determining if wood hardness (difficult to measure) is related to wood density (easy to measure), and, if so, predicting hardness from the density. For the first task, our (alternative) hypothesis here is that density is a good predictor of hardness. It is clear in this case from simply looking at a plot that this hypothesis is correct, and so the hypothesis test itself is not so interesting here. But the model can still be useful. And the general approach applies in cases where we are more interested in assessing the “statistical significance” of a relationship. And we may still be interested in hypothesis tests about the nature of the relationship between hardness and density. We want to build this model:

$$y_i = \beta_0 + \beta_1 x_i + e_i \quad (1)$$

where  $y_i$  is the hardness for the  $j^{th}$  observation, and  $e_i$  is the (random, normally distributed) error for this observation.

To fit a linear model, we can use the `lm()` function.

```
mod <- lm(hardness ~ density, data = hard)
mod
```

```
#  
# Call:
```

```
# lm(formula = hardness ~ density, data = hard)
#
# Coefficients:
# (Intercept)      density
# -1160.50        57.51
```

R returns only the call and coefficients by default. You can get a lot more information using the `summary()` function, which we'll use below.

As mentioned above, the output from the `lm()` function is an object of class `lm()`. These objects are lists that contain at least the following elements (you can find a complete list in the help file for `lm()`):

Element	What it returns
<code>coefficients</code>	a named vector of coefficients
<code>residuals</code>	the residuals, that is response minus fitted values
<code>fitted.values</code>	the fitted mean values
<code>df.residual</code>	the residual degrees of freedom
<code>call</code>	the matched call
<code>terms</code>	the terms object used
<code>y</code>	if requested, the response used
<code>x</code>	if requested, the model matrix used
<code>model</code>	if requested (the default), the model frame used

```
class(mod)
```

```
# [1] "lm"
```

Let's take a look at some of the options R has for dealing with linear model output. To get at elements listed above, you can simply index the `lm()` object, i.e., call up part of the list.

```
mod$coefficients

# (Intercept)      density
# -1160.49970    57.50667
```

However, R has several extractor functions designed precisely for pulling data out of statistical model output. Venables et al. [25] gives a list of extractor functions and a brief description of the most commonly used ones: `add1()`, `alias()`, `anova()`, `coef()`, `deviance()`, `drop1()`, `effects()`, `family()`, `formula()`, `kappa()`, `labels()`, `plot()`, `predict()`, `print()`, `proj()`, `residuals()`, `step()`, `summary()`, and `vcov()`.

The `coef()` function returns model coefficients—in this case  $\beta_0$  and  $\beta_1$  given in Eq. 1.

```
coef(mod)

# (Intercept)      density
# -1160.49970    57.50667
```

The `resid()` function returns residuals, i.e.,

$$r = y_i - \hat{y} \quad (2)$$

```
resid(mod)
```

```
#          1         2         3         4         5
# 224.0848370 161.3341695 3.5674826 44.3101404 76.3101404
#       6         7         8         9        10
# 140.8061355 5.0474583 -15.9685611 92.2620821 -139.5072748
#      11        12        13        14        15
# -0.7592772 -79.5126146 104.7367180 -145.0166194 2.9807107
#      16        17        18        19        20
# -164.2712918 -80.0219592 -50.0219592 -36.5366437 -293.3060005
#      21        22        23        24        25
# -136.5633428 148.6779800 -91.0940467 208.9059533 -30.3567287
#      26        27        28        29        30
# -79.8740831 -268.6274205 -114.6327603 -171.8847628 66.1045576
#      31        32        33        34        35
# -338.3994472 625.0591692 -15.4501754 94.0404799 -73.2115225
#      36
# 326.7884775
```

One of the most useful extractor functions is `summary()`.

```
summary(mod)
```

```
#
# Call:
# lm(formula = hardness ~ density, data = hard)
#
# Residuals:
#   Min     1Q   Median     3Q    Max
# -338.40 -96.98 -15.71  92.71 625.06
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept) -1160.500   108.580 -10.69 2.07e-12 ***
# density      57.507     2.279   25.24 < 2e-16 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 183.1 on 34 degrees of freedom
# Multiple R-squared:  0.9493, Adjusted R-squared:  0.9478
# F-statistic:  637 on 1 and 34 DF, p-value: < 2.2e-16
```

Let's pick apart this output. The "Call" shows the code you used to create the model. "Residuals" shows a summary of model residuals. The "Coefficients" section gives least-squares estimates for model coefficients, standard error estimates, and the result of a Wald-type hypothesis test on whether or not they are different from zero. The results of this test will be dependent on how you specify the

model—especially when working with factors. Finally, the last three lines show information on the fit of the whole model. Residual standard error is the estimate of standard deviation of the population. It, and all the hypothesis test given in this output, are calculated from various sums of squares (SS). The multiple  $R^2$  value provides an indication of model fit—it can be interpreted as the fraction of total variation in the data explained by the model. The adjusted value is penalized for the number of parameters included. And finally, the last line shows the results of an  $F$  test for the model—this is the estimate of the probability that a *null* model could have produced the observed data.

Much of the behind-the-scenes operations carried out by statistical functions in R are done by code written in R<sup>118</sup>, and there is nothing unique about the objects returned by these functions. So it is relatively easy to check or modify results from statistical functions, and not that difficult to build new ones<sup>119</sup>.

Here is a list of what's available from the `summary` function for this model<sup>120</sup>:

```
names(summary(mod))

# [1] "call"          "terms"         "residuals"      "coefficients"
# [5] "aliased"       "sigma"         "df"            "r.squared"
# [9] "adj.r.squared" "fstatistic"    "cov.unscaled"

summary(mod) [[4]]

#           Estimate Std. Error   t value   Pr(>|t|) 
# (Intercept) -1160.49970 108.579605 -10.68801 2.065919e-12
# density      57.50667   2.278534  25.23845 1.332735e-23
```

This flexibility is useful, but it makes for some redundancy in R. For some model statistics, there are three ways to get your data: an extractor function (such as `coef()`), indexing the `lm` object, and indexing output from the `summary()` function. The best approach is to use an extractor function whenever you can<sup>121</sup>. In some cases, the `summary` function will return results that you cannot get by indexing or using other extractor functions.

Let's talk a little about hypothesis testing. The Wald-type results shown from `summary.lm` are equivalent to an  $F$  test based on the effect of adding a term to the model that includes all the other terms in the model. So in this summary:

```
summary(mod)$coefficients

#           Estimate Std. Error   t value   Pr(>|t|) 
# (Intercept) -1160.49970 108.579605 -10.68801 2.065919e-12
# density      57.50667   2.278534  25.23845 1.332735e-23
```

the  $P$ -value  $1.33 \times 10^{-23}$  corresponds to the probability that the reduction in SS caused by adding `density` to a model that included only an intercept could have been produced by that original model. So it is a very low probability in this case.

<sup>118</sup> R functions can also include C and Fortran code.

<sup>119</sup> Or at least that would seem to be the case—I've never tried.

<sup>120</sup> The `str()` function could also be useful here.

<sup>121</sup> These should remain the same in future versions of R, while it is possible that names or order within the `lm` object will change, so indexing may not always work the same.

In this simple model, with only one predictor, this Wald-type test is equivalent to the overall  $F$ -test for the model. Notice that if you square the  $t$  value for the Wald-type test, it is the same as the  $F$  value for the overall test.

Let's talk more about  $F$ -tests for `lm` objects. The powerhouse function here is `anova()`. This function will calculate an analysis of variance table, which can be used to evaluate the significance of the terms in single models or to compare two nested models. The ANOVA table returned with `anova()` show the results of successive deletion tests, i.e., tests are based on Type I sum of squares (SS)<sup>122</sup>.

```
anova(mod)

# Analysis of Variance Table
#
# Response: hardness
#           Df  Sum Sq Mean Sq F value    Pr(>F)
# density     1 21345674 21345674 636.98 < 2.2e-16 ***
# Residuals  34 1139366   33511
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

This result isn't particularly informative for this simple model—it duplicates what we've already seen in the output from `summary`<sup>123</sup>. The `anova()` function can also be used to compare nested models, i.e., models that are all `subsets` of one larger model. For example, we already know that `density` is a significant predictor, but how about a quadratic term?

```
mod2 <- lm(hardness ~ density + I(density^2), data = hard)
```

We can compare this new model to the original like this:

```
anova(mod, mod2)

# Analysis of Variance Table
#
# Model 1: hardness ~ density
# Model 2: hardness ~ density + I(density^2)
#   Res.Df   RSS Df Sum of Sq    F    Pr(>F)
# 1     34 1139366
# 2     33  863325  1   276041 10.552 0.002669 **
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

But this  $F$ -test result is identical to the Wald-type  $t$ -test given in the output from `summary()`. In fact, the  $F$  statistic is exactly equal to the square of the  $t$  statistic. When possible, just use the output from `summary()`.

---

<sup>122</sup> This is different from the default approach used in some other statistical software, notably SAS. More discussion on this topic follows in Section 20.

<sup>123</sup> Two locations in this output.

```

summary(mod2)

#
# Call:
# lm(formula = hardness ~ density + I(density^2), data = hard)
#
# Residuals:
#      Min      1Q Median      3Q     Max
# -326.63 -81.35 -15.27  59.87 537.82
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept) -118.0074   334.9669 -0.352 0.72686
# density      9.4340    14.9356  0.632 0.53197
# I(density^2)  0.5091     0.1567  3.248 0.00267 **
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 161.7 on 33 degrees of freedom
# Multiple R-squared:  0.9616, Adjusted R-squared:  0.9593
# F-statistic: 413.2 on 2 and 33 DF, p-value: < 2.2e-16

```

But, what if we wanted to compare a model with both linear and quadratic terms to a null model? We can make a null model, with only an intercept term, like this:

```

modn <- lm(hardness ~ 1, data = hard)
summary(modn)

#
# Call:
# lm(formula = hardness ~ 1, data = hard)
#
# Residuals:
#      Min      1Q Median      3Q     Max
# -1056.5 -506.7 -274.5  510.5 1790.5
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept) 1469.5       133.6     11 6.61e-13 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 801.5 on 35 degrees of freedom

```

Then for our test:

```

anova(modn, mod2)

# Analysis of Variance Table
#

```

```

# Model 1: hardness ~ 1
# Model 2: hardness ~ density + I(density^2)
#   Res.Df      RSS Df Sum of Sq    F    Pr(>F)
# 1     35 22485041
# 2     33  863325  2  21621716 413.24 < 2.2e-16 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

Interestingly, it looks like there is good reason to include the quadratic term in our model. In general, a better way to carry out polynomial regression is with the `poly` function, which is used for orthogonal polynomial regression.

Once we have fit a model in R, we can generate predicted values using the `predict()` function, which creates an  $X$  matrix (predictor matrix) of orthogonal polynomials. One advantage of this approach over the use of explicit polynomial terms using `I()` above is that coefficient estimates are independent on the terms included in the model.

```

modp <- lm(hardness ~ poly(density, 3), data = hard)
summary(modp)

#
# Call:
# lm(formula = hardness ~ poly(density, 3), data = hard)
#
# Residuals:
#   Min     1Q Median     3Q    Max
# -310.98 -92.52 -14.94  61.41 537.92
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept) 1469.47    27.29  53.841 < 2e-16 ***
# poly(density, 3)1 4620.14   163.76  28.213 < 2e-16 ***
# poly(density, 3)2  525.40   163.76   3.208 0.00303 **
# poly(density, 3)3   72.14   163.76   0.441  0.66252
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 163.8 on 32 degrees of freedom
# Multiple R-squared:  0.9618, Adjusted R-squared:  0.9583
# F-statistic: 268.8 on 3 and 32 DF,  p-value: < 2.2e-16

```

In truth though, polynomials above degree 2 are not particularly useful—terms are difficult to interpret, predictions oscillate, and extrapolation is very poor. Let's move on from hypothesis tests.

Another useful extractor function is `predict()`, which can be used to produce model predictions.

```

predict(mod2)

#
#       1        2        3        4        5        6        7
# 425.5960 429.0594 518.9517 560.5203 560.5203 583.7134 635.2224
#       8        9       10       11       12       13       14

```

```

#  734.8366  863.0282  999.7825 1014.4182 1039.0147 1043.9646 1068.8665
#      15       16       17       18       19       20       21
# 1088.9713 1104.1568 1109.2390 1109.2390 1223.6234 1381.9321 1444.2201
#      22       23       24       25       26       27       28
# 1519.4216 1718.0454 1718.0454 1837.4343 2006.7647 2040.1174 2094.0110
#      29       30       31       32       33       34       35
# 2114.3891 2224.6199 2266.6285 2722.1795 2830.4624 2940.7409 2964.6317
#      36
# 2964.6317

```

This is a generic function, and it works for a whole range of statistical models in R—not just `lm` objects. Of course, we can treat these predictions as we would any vector, e.g., we can add them to the above plot or put them in the original data frame. The `predict()` function can also give confidence and prediction intervals.

```

predict(mod, int = "conf")

#          fit      lwr      upr
# 1    259.9152 144.4580 375.3724
# 2    265.6658 150.5990 380.7327
# 3    409.4325 303.9330 514.9320
# 4    472.6899 371.2673 574.1125
# 5    472.6899 371.2673 574.1125
# 6    507.1939 407.9554 606.4323
# 7    581.9525 487.3394 676.5657
# 8    719.9686 633.4426 806.4945
# 9    886.7379 808.9806 964.4952
# 10   1053.5073 983.0352 1123.9793
# 11   1070.7593 1000.9368 1140.5818
# 12   1099.5126 1030.7240 1168.3013
# 13   1105.2633 1036.6739 1173.8527
# 14   1134.0166 1066.3848 1201.6485
# 15   1157.0193 1090.1056 1223.9329
# 16   1174.2713 1107.8675 1240.6751
# 17   1180.0220 1113.7825 1246.2614
# 18   1180.0220 1113.7825 1246.2614
# 19   1306.5366 1243.1602 1369.9131
# 20   1473.3060 1411.3016 1535.3104
# 21   1536.5633 1474.3248 1598.8019
# 22   1611.3220 1548.2751 1674.3689
# 23   1801.0940 1733.5849 1868.6032
# 24   1801.0940 1733.5849 1868.6032
# 25   1910.3567 1838.9092 1981.8043
# 26   2059.8741 1981.7427 2138.0055
# 27   2088.6274 2009.0660 2168.1888
# 28   2134.6328 2052.6992 2216.5663
# 29   2151.8848 2069.0364 2234.7331
# 30   2243.8954 2155.9582 2331.8327
# 31   2278.3994 2188.4707 2368.3282
# 32   2634.9408 2522.4622 2747.4195
# 33   2715.4502 2597.5086 2833.3918
# 34   2795.9595 2672.4562 2919.4628
# 35   2813.2115 2688.5049 2937.9182

```

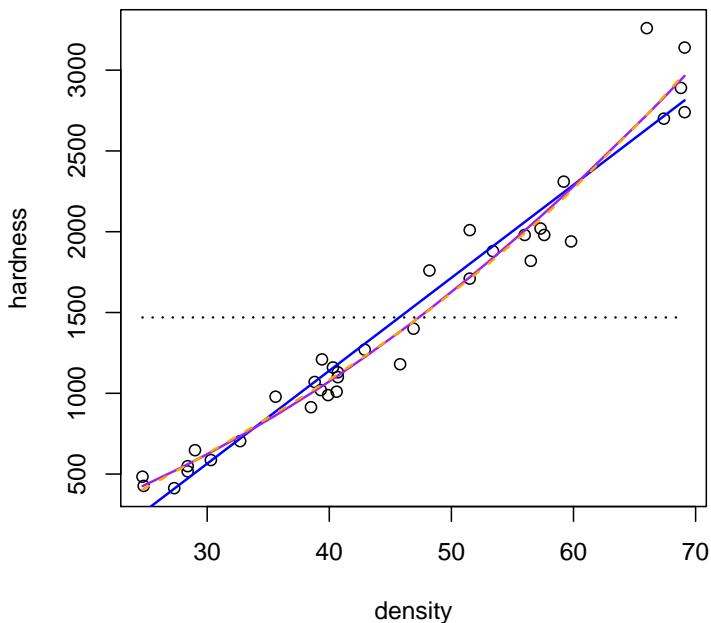
```
# 36 2813.2115 2688.5049 2937.9182
```

Let's plot some of our models. First we'll add predictions to the original data frame.

```
hard$pred <- predict(mod)
hard$pred2 <- predict(mod2)
hard$predp <- predict(modp)
hard$predn <- predict(modn)
```

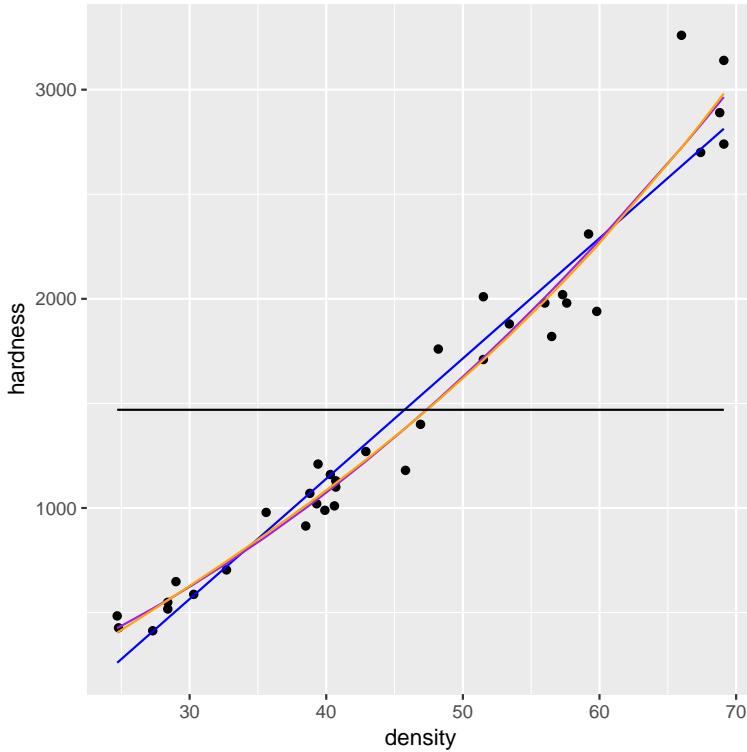
And put these predictions on a plot.

```
plot(hardness ~ density, data = hard)
lines(pred ~ density, data = hard, col = "blue", lwd = 1.6)
lines(pred2 ~ density, data = hard, col = "purple", lwd = 1.6)
lines(predp ~ density, data = hard, col = "orange", lty = 2, lwd = 1.6)
lines(predn ~ density, data = hard, col = "black", lty = 3, lwd = 1.6)
```



How could we do this with ggplot instead?

```
ggplot(hard) + geom_point(aes(density, hardness)) +
  geom_line(aes(density, pred), colour = 'blue') +
  geom_line(aes(density, pred2), colour = 'purple') +
  geom_line(aes(density, predp), colour = 'orange') +
  geom_line(aes(density, predn), colour = 'black')
```



But this may not be the most efficient approach. Let's try reshaping the data first.

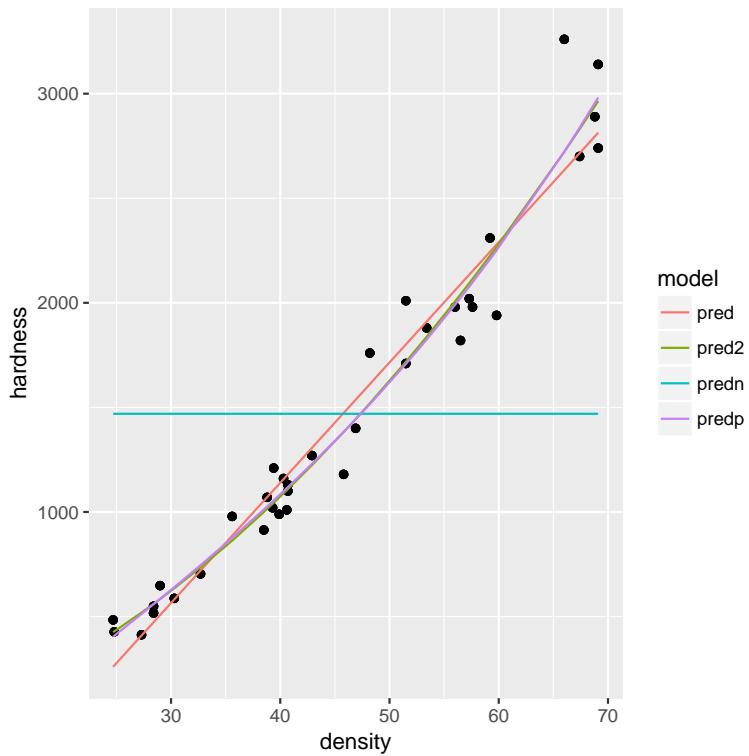
```
library(tidyr)
head(hard)

#   density hardness      pred     pred2     predp     predn
# 1    24.7      484 259.9152 425.5960 403.8437 1469.472
# 2    24.8      427 265.6658 429.0594 407.9858 1469.472
# 3    27.3      413 409.4325 518.9517 512.3600 1469.472
# 4    28.4      517 472.6899 560.5203 558.8727 1469.472
# 5    28.4      549 472.6899 560.5203 558.8727 1469.472
# 6    29.0      648 507.1939 583.7134 584.4203 1469.472

hardl <- gather(hard, model, value, -density:-hardness)
head(hardl)

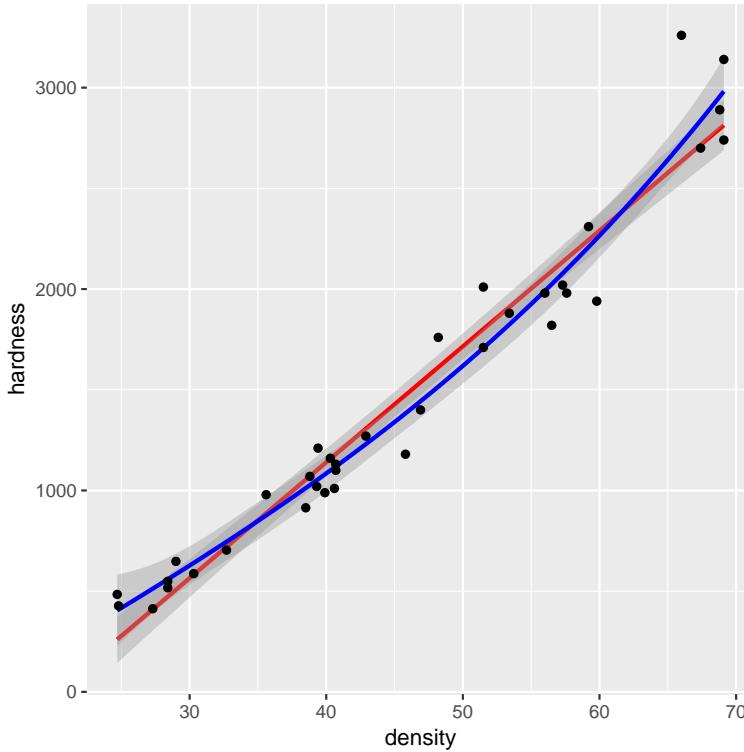
#   density hardness model      value
# 1    24.7      484  pred 259.9152
# 2    24.8      427  pred 265.6658
# 3    27.3      413  pred 409.4325
# 4    28.4      517  pred 472.6899
# 5    28.4      549  pred 472.6899
# 6    29.0      648  pred 507.1939

ggplot(hardl) + geom_point(aes(density, hardness)) +
  geom_line(data = hardl, aes(density, value, colour = model))
```



And a completely different approach would be to have `ggplot()` fit the models.

```
ggplot(hard, aes(density, hardness)) +
  geom_smooth(method = 'lm', formula = y ~ x, colour = 'red') +
  geom_smooth(method = 'lm', formula = y ~ poly(x, 3), colour = 'blue') +
  geom_point()
```



And so on. One disadvantage here is that we only see the model predictions.

The approach we used for plotting model predictions before the `geom_smooth()` bit—adding predictions to our original data frame, then plotting them—is actually not the most flexible approach. One problem with is that the data will not be sorted, and (except in the case of a straight line) we will end up with a line that jumps around. A second problem (again, really only when you are not plotting a straight line) is that some areas may not have high enough point density to make a smooth line. So, you will often want to set up a new data frame just for predictions. Such a data frame can be used with the `predict()` function to generate predictions for new values of predictors. To use new data to create predictions from an `lm` object, the data frame names need to match the names in the original data frame used to fit the model.

We'll demonstrate with a nearly saturated model. And let's “unsort” the `density` column in our data frame, since it won't always be sorted.

```

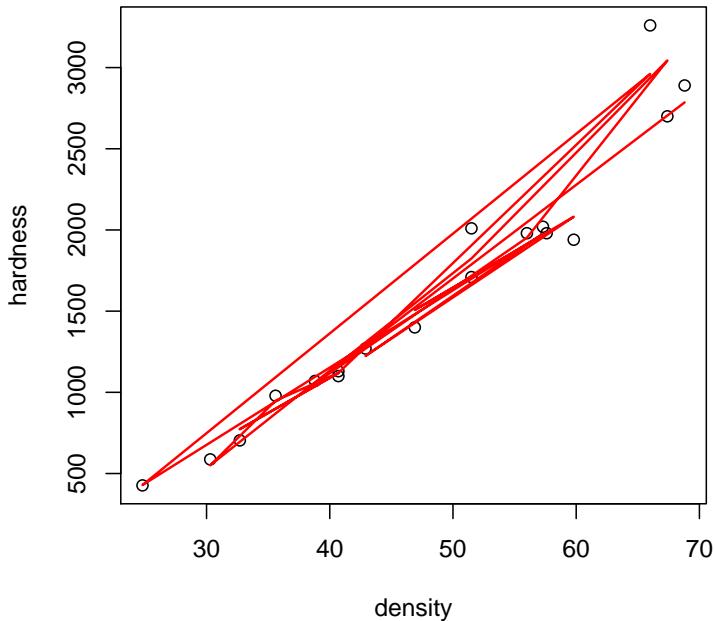
h <- hard[sample(nrow(hard), nrow(hard)), ]
dfsumm(h)

#
# 36 rows and 6 columns
# 36 unique rows
#
#          density hardness     pred    pred2    predp    predn
# Class      numeric   integer  numeric  numeric  numeric  numeric
# Minimum      24.7      413      260      426      404     1470
# Maximum      69.1     3260     2810     2960     2980     1470
# Mean        45.7     1180     1470     1470     1470     1470
# Unique (excl. NA)  32       35       32       32       32       1
# Missing values  0        0        0        0        0        0
# Sorted       FALSE     FALSE     FALSE     FALSE     FALSE     TRUE

```

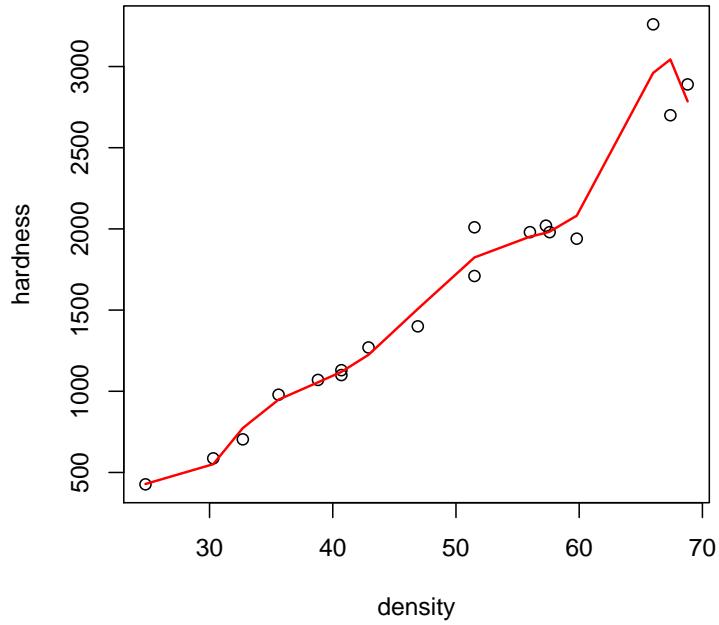
And we'll remove half the points.

```
h <- hard[sample(nrow(hard), nrow(hard)/2), ]
modsat <- lm(hardness ~ poly(density, 8), data = h)
h$predsat <- predict(modsat)
plot(hardness ~ density, data = h)
lines(predsat ~ density, data = h, col = "red", lwd = 1.6)
```



That's a mess. We can improve things a bit by sorting the data frame.

```
h <- h[order(h$density), ]
plot(hardness ~ density, data = h)
lines(predsat ~ density, data = h, col = "red", lwd = 1.6)
```

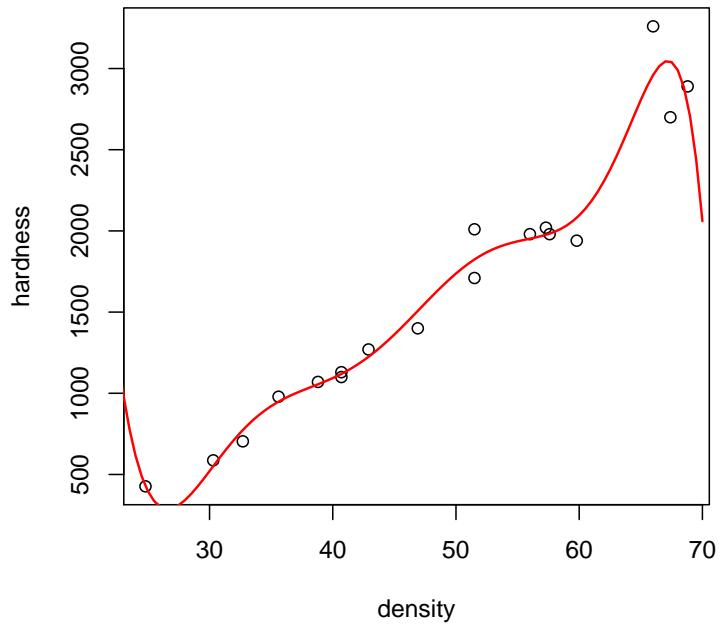


But our polynomial model predicts a smooth response to `hardness`, and that is not what we see. We'll need to add more points to change this. Here is a general solution to this problem.

```

h.pred <- data.frame(density = seq(10, 70, 0.5))
h.pred$hardness <- predict(modsat, newdata = h.pred)
plot(hardness ~ density, data = h)
lines(hardness ~ density, data = h.pred, col = "red", lwd = 1.6)

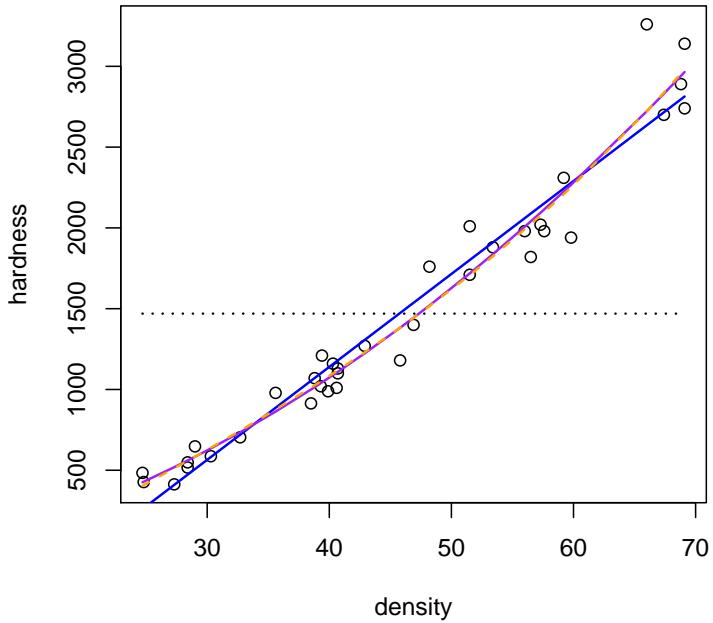
```



This model is probably not useful, but at least it is accurately plotted!

Back to our more reasonable models.

```
plot(hardness ~ density, data = hard)
lines(pred ~ density, data = hard, col = "blue", lwd = 1.6)
lines(pred2 ~ density, data = hard, col = "purple", lwd = 1.6)
lines(predp ~ density, data = hard, col = "orange", lty = 2, lwd = 1.6)
lines(predn ~ density, data = hard, col = "black", lty = 3, lwd = 1.6)
```



Do these predictions match the model summaries? Can you see a relationship with model statistics and their apparent fit? Let's look at some statistics for model fit. First, multiple  $R^2$ .

```
summary(mod)$r.squared
```

```
# [1] 0.9493278
```

```
summary(mod2)$r.squared
```

```
# [1] 0.9616045
```

```
summary(modp)$r.squared
```

```
# [1] 0.9618359
```

```
summary(modn)$r.squared
```

```
# [1] 0
```

So the best fit is with the degree 3 polynomial model `modp`—no surprise there, and, as we can see in the plot, there is very little difference between it and the degree 2 model. We can also look at the AIC (Akaike's “An Information Criterion”), which is based on the model likelihood, and includes a penalty for the number of parameters.

```

AIC(mod)

# [1] 481.2123

AIC(mod2)

# [1] 473.2246

AIC(modp)

# [1] 475.0069

AIC(modn)

# [1] 586.5779

```

By this criterion, we can conclude that the degree 2 polynomial model `mod2` is the best one. That is actually the same conclusion we drew from the hypothesis tests above.

An important part of model evaluation is evaluating the assumptions of the model. Let's take a look at the general equation for our best model.

$$y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + e_i \quad (3)$$

The equations involved in fitting this model and hypothesis testing are based on a few important assumptions:

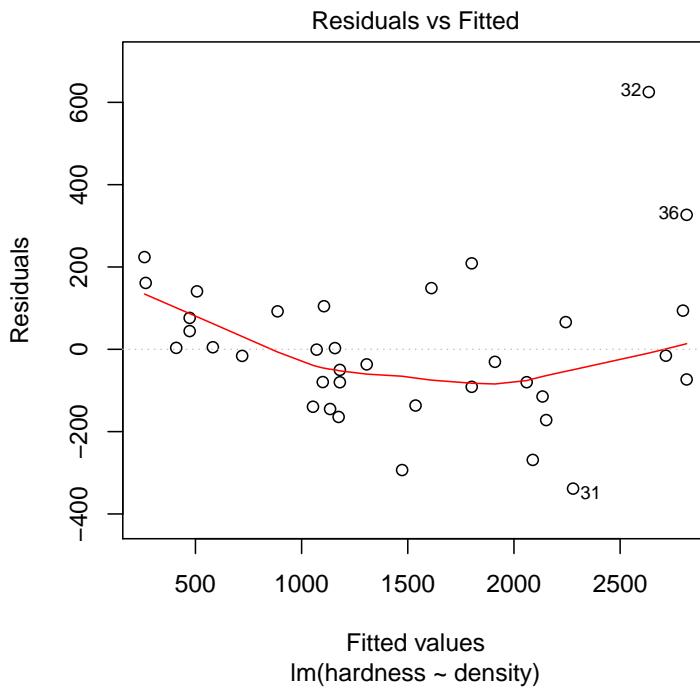
1. The response is a linear function of model parameters (not necessarily model predictors)
2. Variance is constant
3. Errors ( $e$ ) are normally distributed
4. Effects are additive

We can only evaluate these assumptions after we've fit a model. Model diagnostics are based on residuals, which we can take as an estimate of model error  $e$ . We can use the `plot` function<sup>124</sup> for model diagnostics. The help file for `plot.lm` indicates that this function will generate six different plots. Let's start with the first one.

```
plot(mod, 1)
```

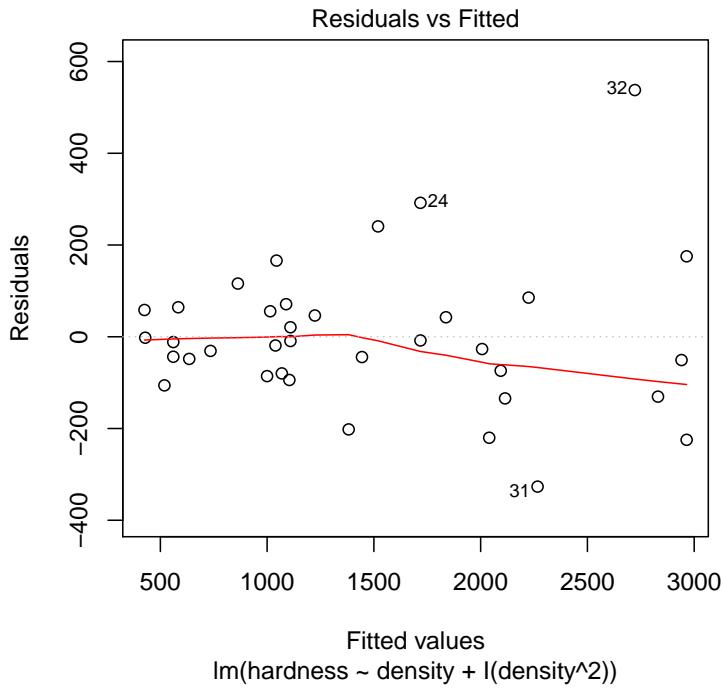
---

<sup>124</sup> Remember it is generic, so it does something different with an `lm()` object than, say, a vector.



This plot shows residuals  $r$  versus predicted values  $\hat{y}$ —it is handy for checking the model structure. It doesn't look that great here—there seems to be underprediction at low and high values, and overprediction near the middle. This response is a clear indication that a linear relationship between our predictor and response is not appropriate. Let's try out quadratic model.

```
plot(mod2, 1)
```



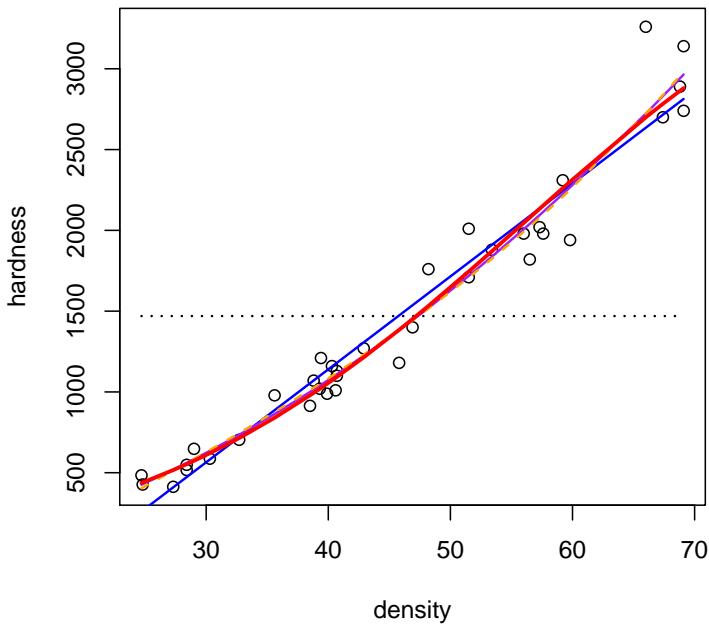
This looks much better, but now there is another issue that is easy to see—our variance doesn't look constant, but it seems to increase with  $\hat{y}$ . A transformation of the response is one way to handle something like this. And a common transformation that stabilizes variance is the log transformation. Let's try it.

```
modlog <- lm(log10(hardness) ~ poly(density, 2), data = hard)
summary(modlog)
```

```
# 
# Call:
# lm(formula = log10(hardness) ~ poly(density, 2), data = hard)
#
# Residuals:
#       Min     1Q     Median      3Q     Max
# -0.096983 -0.024792 -0.004795  0.032573  0.081955
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)    
# (Intercept) 3.099195  0.007294 424.896 < 2e-16 ***
# poly(density, 2)1 1.470617  0.043764 33.603 < 2e-16 ***
# poly(density, 2)2 -0.234322  0.043764 -5.354 6.49e-06 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 0.04376 on 33 degrees of freedom
# Multiple R-squared:  0.9723, Adjusted R-squared:  0.9706 
# F-statistic: 578.9 on 2 and 33 DF,  p-value: < 2.2e-16
```

With a transformed response, it is difficult to compare this model to the other ones—we can no longer use `anova`, and it doesn't make sense to compare multiple  $R^2$  or AIC values. It would be a good idea to repeat the model building process we carried out above with this transformed response, but we won't go through that here. But let's at least take a look at these predictions.

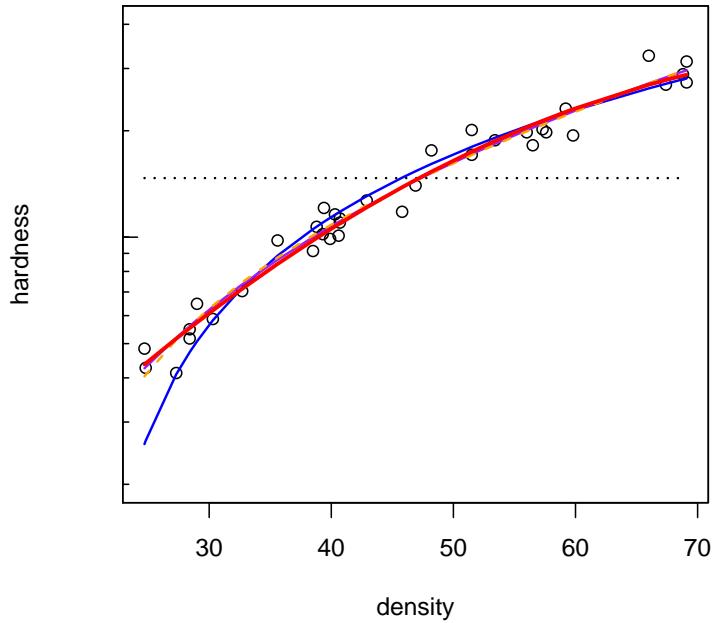
```
hard$predlog <- 10^predict(modlog)
plot(hardness ~ density, data = hard)
lines(pred ~ density, data = hard, col = "blue", lwd = 1.6)
lines(pred2 ~ density, data = hard, col = "purple", lwd = 1.6)
lines(predp ~ density, data = hard, col = "orange", lty = 2, lwd = 1.6)
lines(predn ~ density, data = hard, col = "black", lty = 3, lwd = 1.6)
lines(predlog ~ density, data = hard, col = "red", lty = 1, lwd = 2.5)
```



It fits differently to be sure. With the transformation, (untransformed) residuals above the prediction line have less of an effect than those below. A logarithmic axis can show us why<sup>125</sup>.

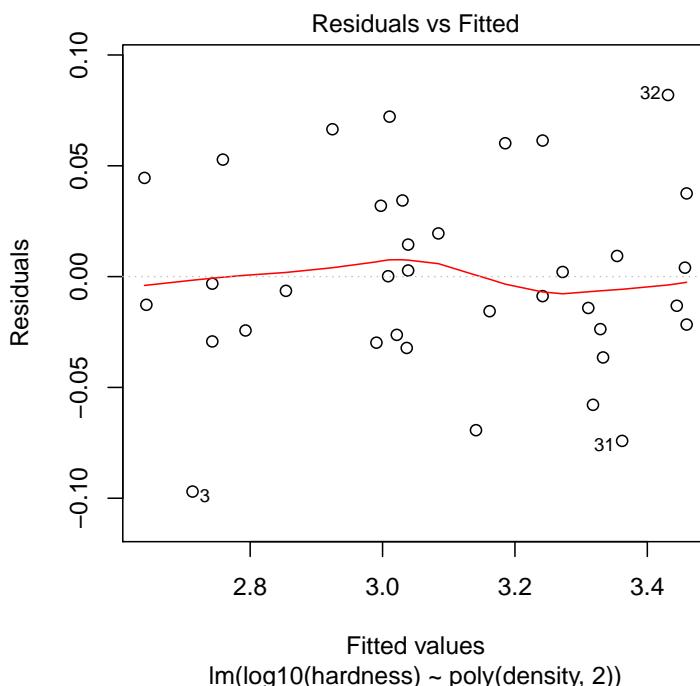
```
source("../functions/logaxis.R")
plot(hardness ~ density, data = hard, log = "y", yaxt = "n", ylim = c(200, 4000))
logaxis(2)
lines(pred ~ density, data = hard, col = "blue", lwd = 1.6)
lines(pred2 ~ density, data = hard, col = "purple", lwd = 1.6)
lines(predp ~ density, data = hard, col = "orange", lty = 2, lwd = 1.6)
lines(predn ~ density, data = hard, col = "black", lty = 3, lwd = 1.6)
lines(predlog ~ density, data = hard, col = "red", lty = 1, lwd = 2.5)
```

<sup>125</sup> The `logaxis` function is in the script file `logaxis.R`. I think it is the first R function I wrote—the code could definitely be improved.



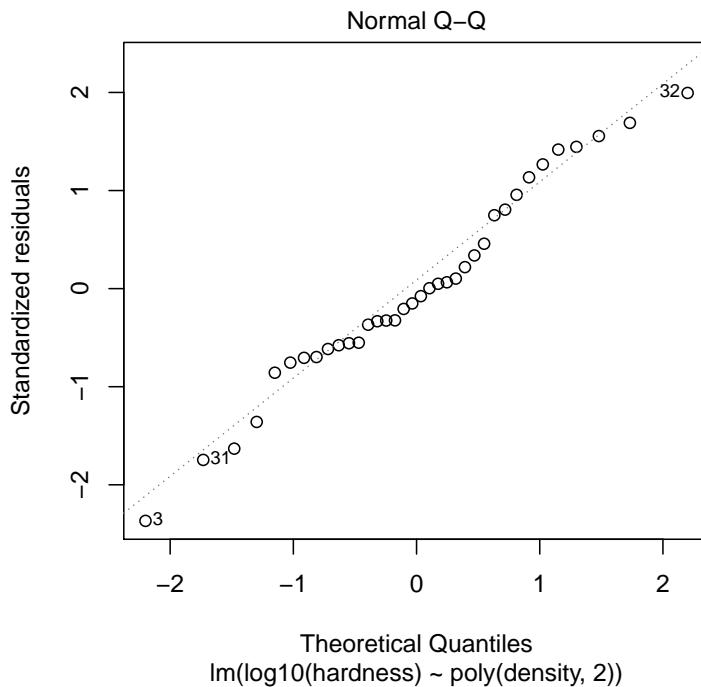
OK, I'm convinced that this new model `modlog` is reasonable. Perhaps our assessment should include a consideration of the purpose of the model. But let's get back to model diagnostics.

```
plot(modlog, 1)
```



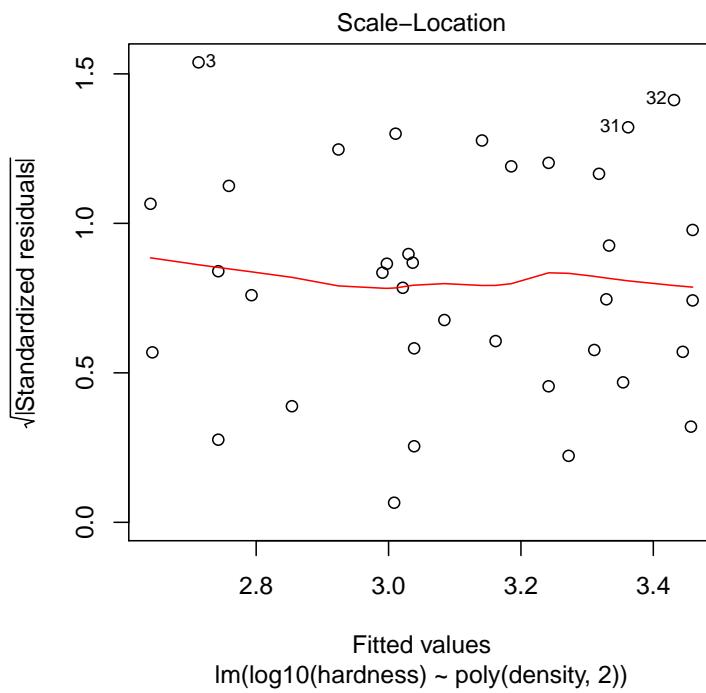
These look great! Moving on.

```
plot(modlog, 2)
```



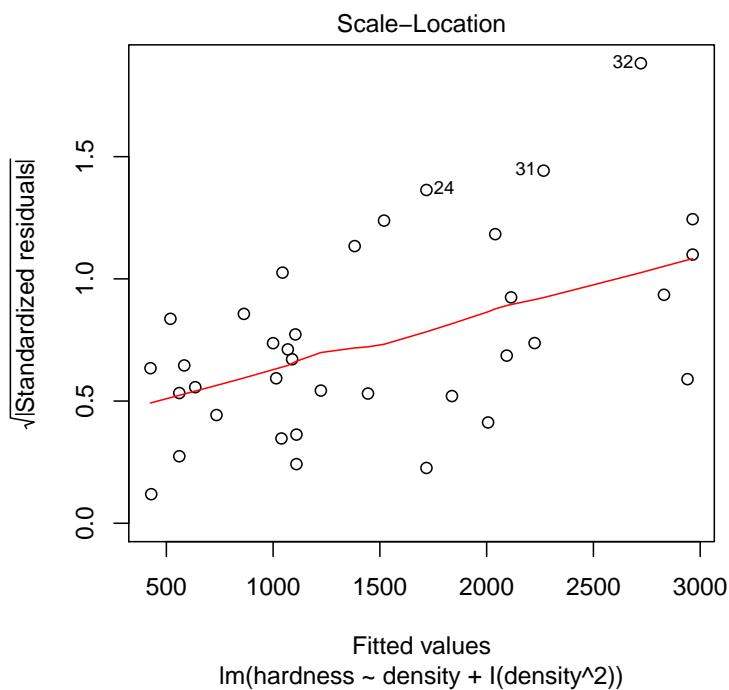
This quantile-quantile plot can be used to assess the assumption of normally distributed errors. The distribution doesn't look perfectly normal, but it isn't bad at all.

```
plot(modlog, 3)
```



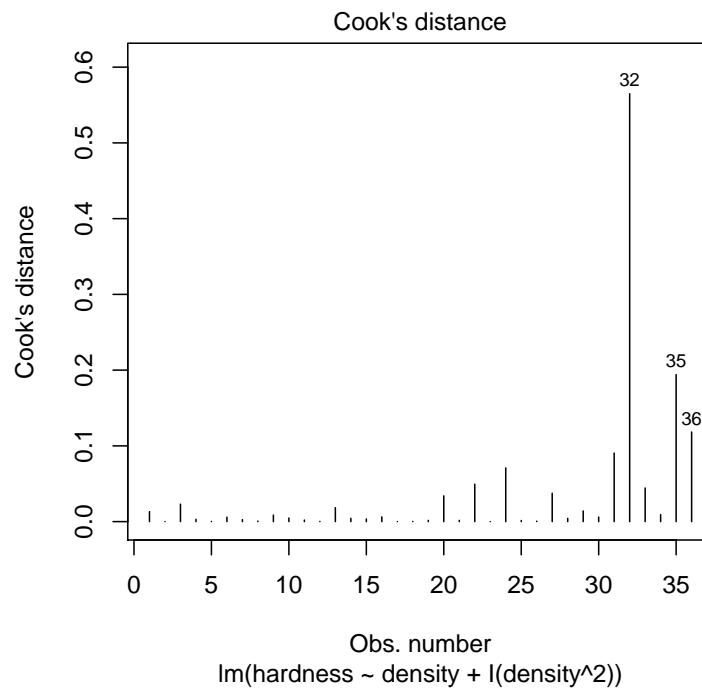
This plot is particularly useful for checking for nonconstant variance. This is what it looks like when variance isn't constant.

```
plot(mod2, 3)
```

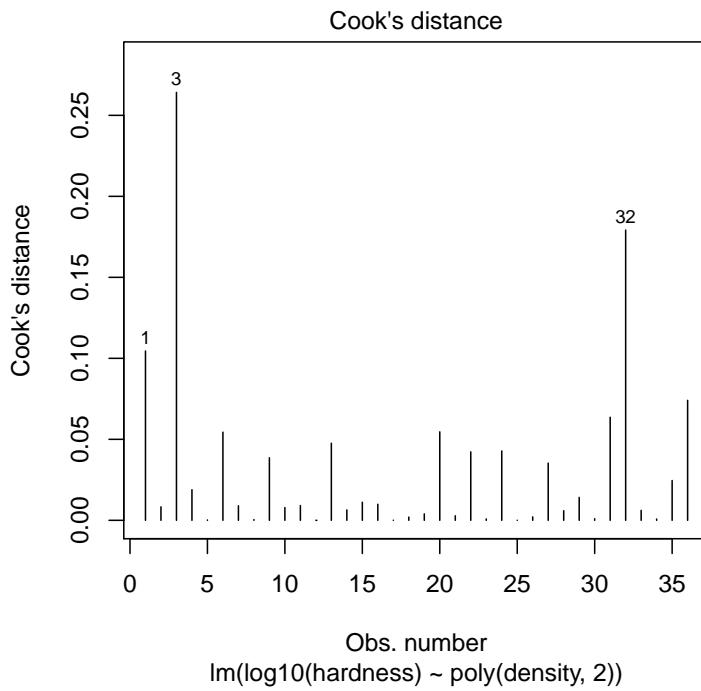


The fourth diagnostic plot returned from `plot.lm` shows Cook's distance, which indicates the influence each point has on model parameters. A rule-of-thumb is that values greater than  $4/n$  should be examined.

```
plot(mod2, 4)
```

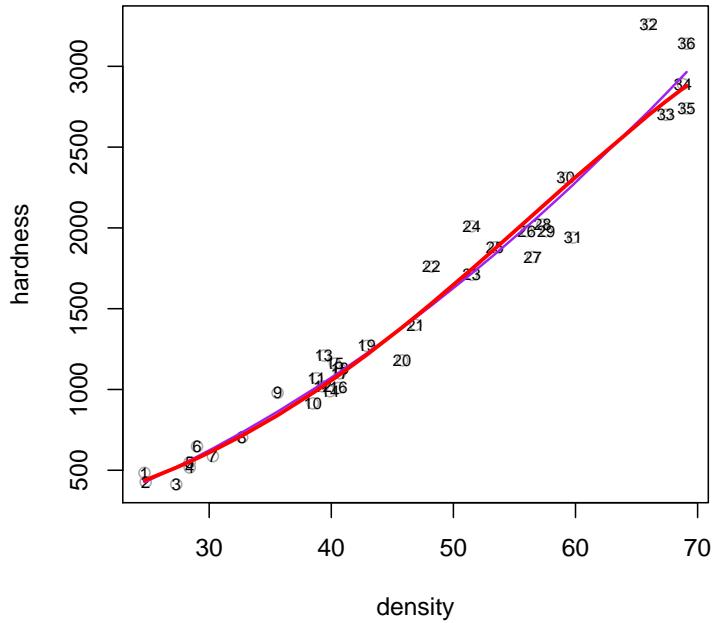


```
plot(modlog, 4)
```



Let's take a closer look at the third and 32<sup>nd</sup> observations.

```
plot(hardness ~ density, data = hard, col = "darkgray")
text(hard$density, hard$hardness, rownames(hard), cex = 0.7)
lines(pred2 ~ density, data = hard, col = "purple", lwd = 1.6)
lines(predlog ~ density, data = hard, col = "red", lty = 1, lwd = 2.5)
```



It would be a stretch to call either of them outliers. The remaining two plots are also useful for evaluating influence. And there are many more tools for `lm` object diagnostics—for example, see the help files for `influence()` and `cooks.distance`.

Let's move on to multiple linear regression. To demonstrate multiple linear regression in R, let's use a data set on ground-level ozone in the atmosphere. We might want to use this observational data set to determine how ozone formation is related to weather.

```

ozone <- read.csv("../data/ozone.csv")
dfsumm(ozone)

#
# 111 rows and 4 columns
# 111 unique rows
#
#          rad      temp     wind    ozone
# Class       integer   integer  numeric  integer
# Minimum        7       57      2.3       1
# Maximum       334      97     20.7     168
# Mean         203      79     9.94      30
# Unique (excl. NA)  93      39      28       66
# Missing values  0       0       0       0
# Sorted        FALSE    FALSE   FALSE   FALSE

```

A quick way to look for relationships between variables in a data frame is with the `cor()` function.

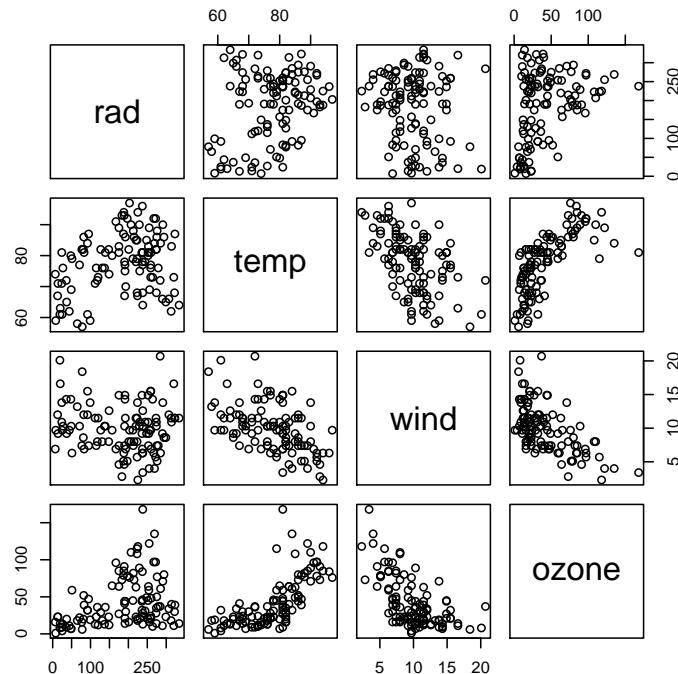
```
cor(ozone)
```

#	rad	temp	wind	ozone
---	-----	------	------	-------

```
# rad      1.0000000  0.2940876 -0.1273656  0.3483417
# temp     0.2940876  1.0000000 -0.4971459  0.6985414
# wind    -0.1273656 -0.4971459  1.0000000 -0.6129508
# ozone    0.3483417  0.6985414 -0.6129508  1.0000000
```

To visualize these relationships, we can use `pairs()`.

```
pairs(ozone)
```



We can see that ozone concentration is positively correlated with temperature (`temp`), negatively correlated with wind speed (`wind`), and less clearly positively correlated with radiation (`rad`). There isn't so much collinearity that we should be concerned. Let's fit a model.

```
mod1 <- lm(ozone ~ rad + temp + wind, data = ozone)
```

Because we are using all columns except the response `ozone` as model predictors, we could use a shortcut syntax.

```
mod1 <- lm(ozone ~ ., data = ozone)
summary(mod1)

#
# Call:
# lm(formula = ozone ~ ., data = ozone)
#
# Residuals:
```

```

#      Min     1Q Median     3Q    Max
# -40.485 -14.210 -3.556 10.124 95.600
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept) -64.23208   23.04204 -2.788 0.00628 **
# rad          0.05980    0.02318  2.580 0.01124 *
# temp         1.65121    0.25341  6.516 2.43e-09 ***
# wind         -3.33760    0.65384 -5.105 1.45e-06 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 21.17 on 107 degrees of freedom
# Multiple R-squared:  0.6062, Adjusted R-squared:  0.5952
# F-statistic: 54.91 on 3 and 107 DF,  p-value: < 2.2e-16

```

Let's drop radiation (although it would probably be significant by most standards).

```

mod2 <- lm(ozone~temp + wind, data = ozone)
summary(mod2)

#
# Call:
# lm(formula = ozone ~ temp + wind, data = ozone)
#
# Residuals:
#      Min     1Q Median     3Q    Max
# -42.160 -13.209 -3.089 10.588 98.470
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept) -67.2008   23.6083 -2.846 0.00529 **
# temp         1.8265    0.2504  7.293 5.32e-11 ***
# wind         -3.2993    0.6706 -4.920 3.12e-06 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 21.72 on 108 degrees of freedom
# Multiple R-squared:  0.5817, Adjusted R-squared:  0.574
# F-statistic: 75.1 on 2 and 108 DF,  p-value: < 2.2e-16

```

We could also use the `update()` function to modify an existing `lm()` object—this is especially handy for dealing with large model formulas.

```

mod2 <- update(mod1, ~. -rad)
summary(mod2)

#
# Call:
# lm(formula = ozone ~ temp + wind, data = ozone)
#

```

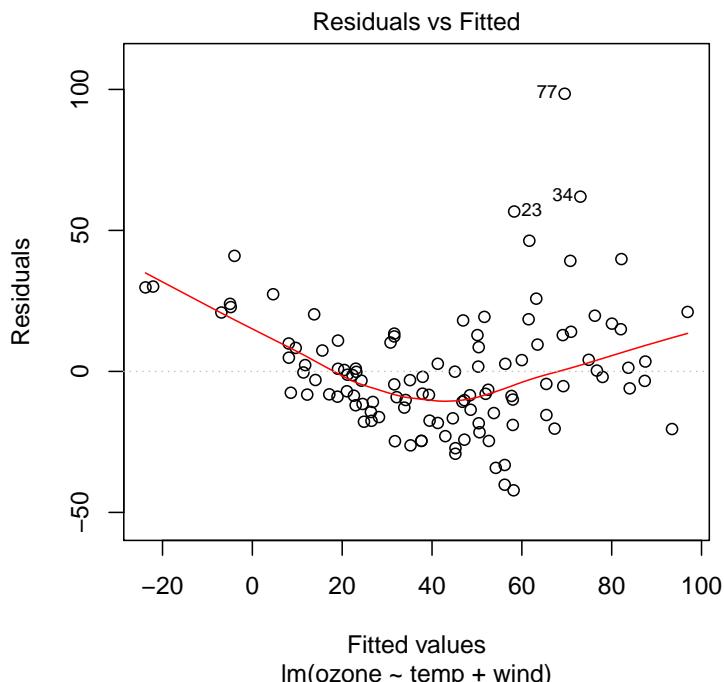
```

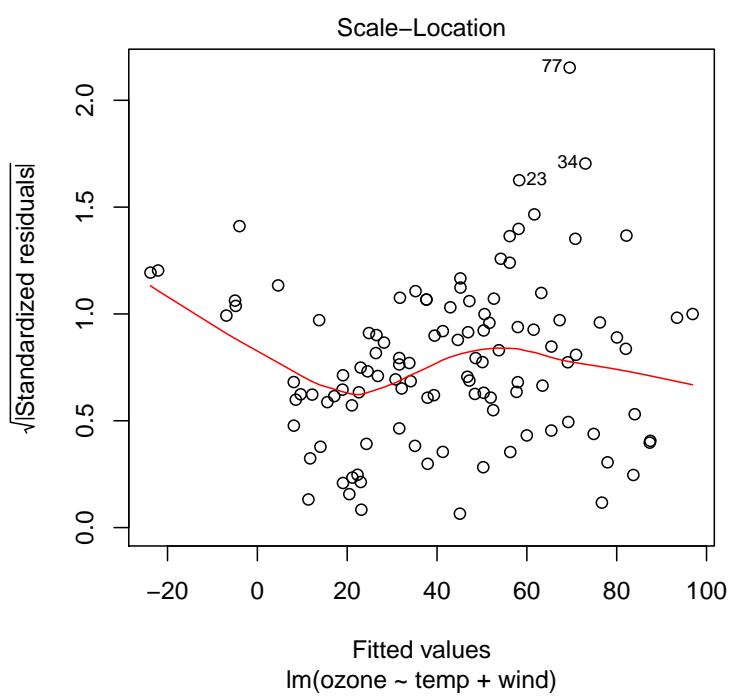
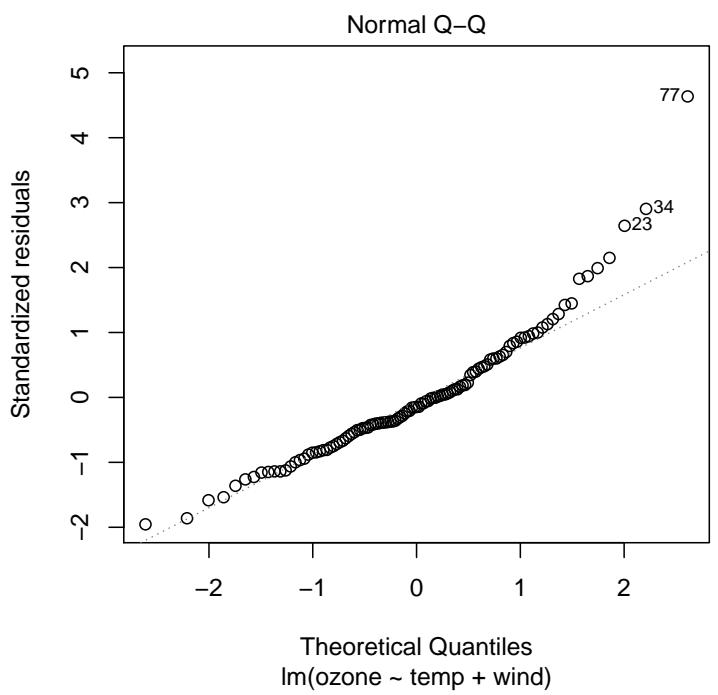
# Residuals:
#      Min     1Q Median     3Q    Max
# -42.160 -13.209 -3.089 10.588 98.470
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept) -67.2008   23.6083 -2.846  0.00529 **
# temp         1.8265    0.2504  7.293 5.32e-11 ***
# wind        -3.2993    0.6706 -4.920 3.12e-06 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 21.72 on 108 degrees of freedom
# Multiple R-squared:  0.5817, Adjusted R-squared:  0.574
# F-statistic: 75.1 on 2 and 108 DF,  p-value: < 2.2e-16

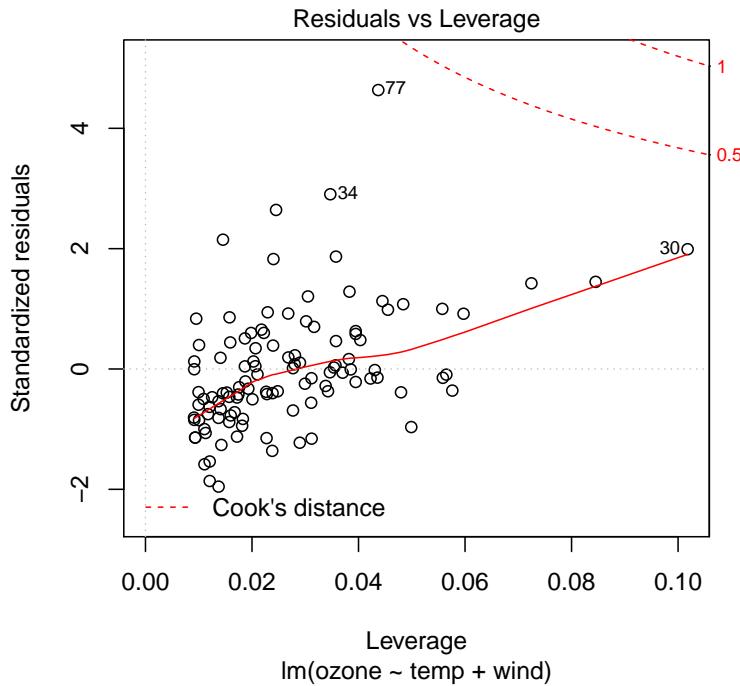
```

Let's take a look at the results.

```
plot(mod2)
```







It looks like the residuals are not normally distributed, and the pattern seen in the residuals plot suggests that a linear response isn't appropriate. Since we are attempting to model concentration data, we might consider log-transforming the values<sup>126</sup>.

```

mod3 <- lm(log10(ozone) ~ rad + temp + wind, data = ozone)
summary(mod3)

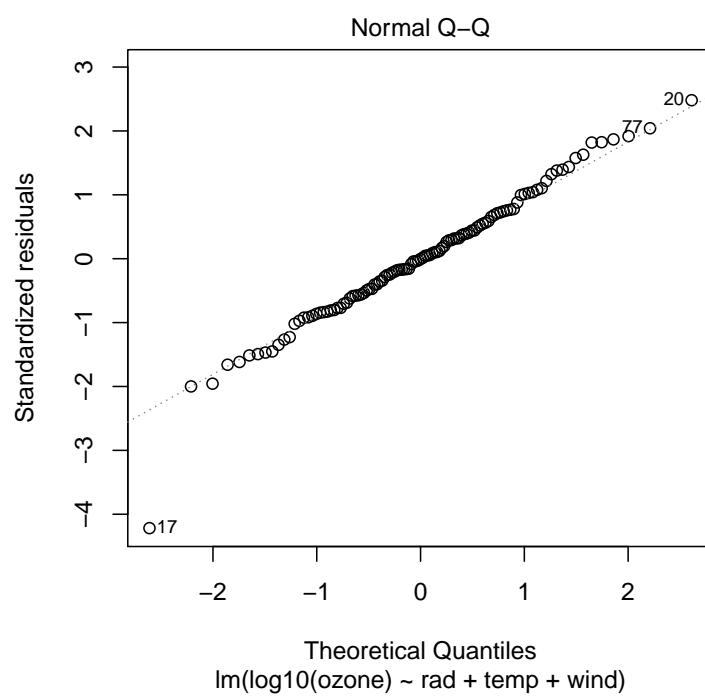
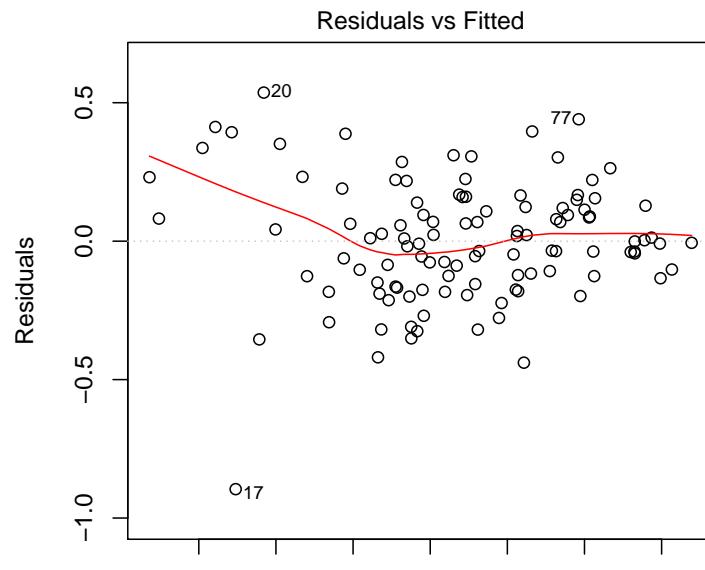
#
# Call:
# lm(formula = log10(ozone) ~ rad + temp + wind, data = ozone)
#
# Residuals:
#      Min        1Q     Median        3Q       Max
# -0.89557 -0.13015 -0.00097  0.13362  0.53667
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept) -0.1134264  0.2403430 -0.472 0.637934
# rad          0.0010921  0.0002418  4.518 1.62e-05 ***
# temp         0.0213512  0.0026433  8.078 1.07e-12 ***
# wind         -0.0267493  0.0068200 -3.922 0.000155 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 0.2208 on 107 degrees of freedom
# Multiple R-squared:  0.6645, Adjusted R-squared:  0.6551
# F-statistic: 70.65 on 3 and 107 DF,  p-value: < 2.2e-16

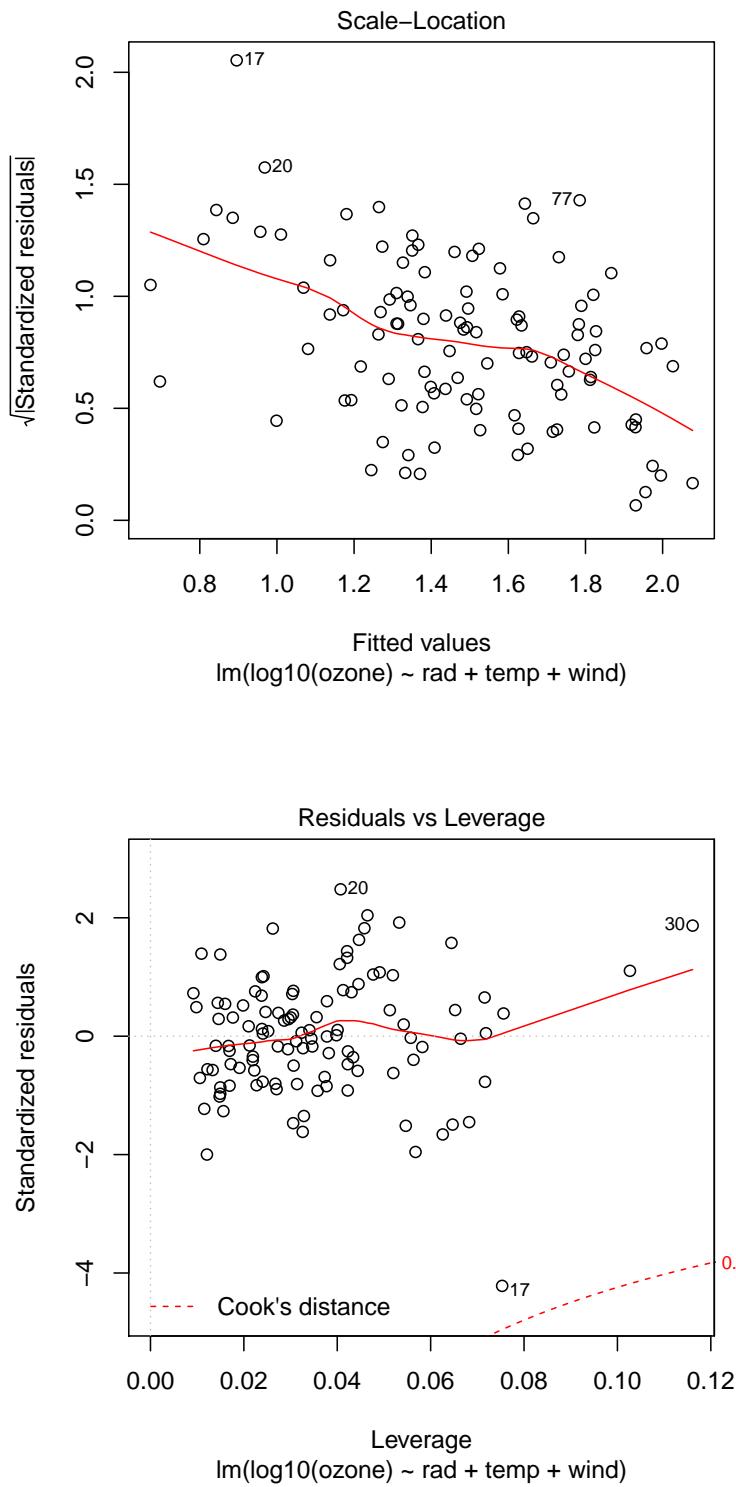
```

<sup>126</sup> R also has a `transform()` function, for transforming variables and maintaining the same variable name.

The first obvious difference is that the  $t$  value has increased for radiation.

```
plot(mod3)
```





Not everything looks better though. We might also consider predictor interactions.

```
mod4 <- lm(log10(ozone) ~ (rad + temp + wind)^3 + 1, data = ozone)
summary(mod4)
```

```

#
# Call:
# lm(formula = log10(ozone) ~ (rad + temp + wind)^3 + 1, data = ozone)
#
# Residuals:
#      Min      1Q  Median      3Q     Max 
# -0.81654 -0.13634 -0.02682  0.11326  0.52866 
#
# Coefficients:
#              Estimate Std. Error t value Pr(>|t|)    
# (Intercept) -2.502e+00 1.222e+00 -2.047 0.04320 *  
# rad          1.074e-02 6.653e-03  1.615 0.10934    
# temp         5.235e-02 1.671e-02  3.133 0.00226 ** 
# wind          1.979e-01 1.006e-01  1.967 0.05190 .  
# rad:temp    -1.235e-04 8.818e-05 -1.401 0.16433    
# rad:wind    -9.196e-04 5.869e-04 -1.567 0.12019    
# temp:wind   -2.972e-03 1.421e-03 -2.091 0.03897 *  
# rad:temp:wind 1.196e-05 7.930e-06  1.509 0.13445  
# ---        
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 0.2174 on 103 degrees of freedom
# Multiple R-squared:  0.687, Adjusted R-squared:  0.6657 
# F-statistic: 32.3 on 7 and 103 DF,  p-value: < 2.2e-16

```

With observational data that contain multiple potential predictors, such as this data set on ozone, variable selection is important. In the example above we used a manual backward selection approach for variable selection. But R has functions for automated model selection based on various criteria. The MASS package has a function that uses an AIC-based approach, called `stepAIC()`.

```

library(MASS)
args(stepAIC)

# function (object, scope, scale = 0, direction = c("both", "backward",
#       "forward"), trace = 1, keep = NULL, steps = 1000, use.start = FALSE,
#       k = 2, ...)
# NULL

stp <- stepAIC(mod4, scope = list(upper= ~ (rad+temp+wind)^3 + 1, lower= ~1), trace = 0)
stp$anova

# Stepwise Model Path
# Analysis of Deviance Table
#
# Initial Model:
# log10(ozone) ~ (rad + temp + wind)^3 + 1
#
# Final Model:
# log10(ozone) ~ (rad + temp + wind)^3 + 1
#
#
#   Step Df Deviance Resid. Df Resid. Dev      AIC
# 1                   103    4.868512 -331.0684

```

Our all the possible nested models, our initial one has the lowest AIC.

The `leaps` package provides functions for evaluating all model subsets of a certain size.

```
library(leaps)
```

We can use use the `regsubsets()` function to find the best models for a given number of predictor variables. Let's use a dataset with more variables from the `faraway` package.

```
library(faraway)
```

```
#  
# Attaching package:  'faraway'  
  
# The following object is masked _by_ '.GlobalEnv':  
#  
#     ozone  
  
# The following object is masked from 'package:plyr':  
#  
#     ozone  
  
ms <- mammalsleep  
detach("package:faraway")  
dfsumm(ms)  
  
#  
# 62 rows and 10 columns  
# 62 unique rows  
#  
#          body    brain nondream    dream    sleep lifespan  
# Class      numeric   numeric   numeric   numeric   numeric   numeric  
# Minimum      0.005     0.14      2.1       0      2.6       2  
# Maximum     6650      5710     17.9      6.6     19.9     100  
# Mean        199       283      8.67      1.97     10.5     19.9  
# Unique (excl. NA)    60       59       39       30       44       47  
# Missing values    0        0       14       12       4        4  
# Sorted        FALSE      FALSE      FALSE      FALSE      FALSE      FALSE  
#  
#          gestation predation exposure danger  
# Class      numeric   integer   integer   integer  
# Minimum      12        1         1         1  
# Maximum     645        5         5         5  
# Mean        142        3         2         2  
# Unique (excl. NA)    49        5         5         5  
# Missing values    4        0         0         0  
# Sorted        FALSE      FALSE      FALSE      FALSE
```

There are three potential response variables here and they are the time spent sleeping for different mammal species: `nondream`, `dream`, and `sleep`. We'll just focus on the total sleep time given in `sleep`. And we'll try to see what other characteristics of mammals are related to the time spent sleeping<sup>127</sup>. Let's find the best models<sup>128</sup>.

<sup>127</sup> Of course we can't say that these characteristics actually affect the time spent sleeping, but just that there is a correlation.

<sup>128</sup> If our data frame did not include two variables we want to exclude, we could have written the formula like this:

```
lms <- summary(regsubsets(sleep ~ body + brain + lifespan + gestation + predation + exposure + danger
```

We can see what variables are included in the best model for each size by

```
lms
```

```
# Subset selection object
# Call: regsubsets.formula(sleep ~ body + brain + lifespan + gestation +
#   predation + exposure + danger, data = ms)
# 7 Variables (and intercept)
#       Forced in Forced out
# body      FALSE      FALSE
# brain     FALSE      FALSE
# lifespan  FALSE      FALSE
# gestation FALSE      FALSE
# predation FALSE      FALSE
# exposure  FALSE      FALSE
# danger    FALSE      FALSE
# 1 subsets of each size up to 7
# Selection Algorithm: exhaustive
#       body brain lifespan gestation predation exposure danger
# 1  ( 1 ) " " " " " " "*" " "
# 2  ( 1 ) " " " " " "*" " " " "
# 3  ( 1 ) " " " " " "*" " *" " "
# 4  ( 1 ) " " "*" " " "*" " *" " "
# 5  ( 1 ) " " "*" " " "*" " *" " "
# 6  ( 1 ) " " "*" "*" " " "*" " *" " "
# 7  ( 1 ) "*" "*" "*" " " "*" " *" " "
```

So our best three-predictor model includes `gestation`, `predation`, and `danger`. Let's take a look at it.

```
summary(lm(sleep ~ gestation + predation + danger, data = ms))
```

```
#
# Call:
# lm(formula = sleep ~ gestation + predation + danger, data = ms)
#
# Residuals:
#   Min     1Q Median     3Q    Max
# -5.3085 -2.1425 -0.2078  1.5183  6.4208
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept) 15.651683  0.907103 17.255 < 2e-16 ***
# gestation   -0.011775  0.003322 -3.545 0.000863 ***
# predation    2.193877  0.800392  2.741 0.008473 **
# danger      -3.777656  0.879825 -4.294 8.07e-05 ***
# ---
#
```

```
lm(sleep .).
```

```

# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 2.85 on 50 degrees of freedom
#   (8 observations deleted due to missingness)
# Multiple R-squared:  0.6415, Adjusted R-squared:  0.6199
# F-statistic: 29.82 on 3 and 50 DF,  p-value: 3.385e-11

```

And here's the best four-predictor model.

```
summary(lm(sleep ~ gestation + predation + danger + brain, data = ms))
```

```

#
# Call:
# lm(formula = sleep ~ gestation + predation + danger + brain,
#      data = ms)
#
# Residuals:
#       Min     1Q     Median      3Q     Max
# -5.3499 -2.2027 -0.2249  1.5745  6.3706
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept) 15.8593577  0.9322241 17.012 < 2e-16 ***
# gestation   -0.0157019  0.0052223 -3.007 0.004158 **
# predation    2.0804827  0.8091950  2.571 0.013226 *
# danger      -3.6012647  0.8986690 -4.007 0.000209 ***
# brain        0.0006377  0.0006542   0.975 0.334445
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 2.851 on 49 degrees of freedom
#   (8 observations deleted due to missingness)
# Multiple R-squared:  0.6483, Adjusted R-squared:  0.6196
# F-statistic: 22.58 on 4 and 49 DF,  p-value: 1.286e-10

```

We can also use `stepAIC` here.

```

m <- na.omit(ms[, -3:-4])
mod <- lm(sleep ~ (.)^2, data = m)
aicsel <- stepAIC(mod, trace = 0)
summary(aicsel)

#
# Call:
# lm(formula = sleep ~ body + brain + lifespan + gestation + predation +
#      exposure + danger + body:exposure + body:danger + brain:lifespan +
#      brain:gestation + brain:predation + brain:danger + lifespan:gestation +
#      lifespan:exposure + gestation:exposure + predation:danger +
#      exposure:danger, data = m)
#
# Residuals:

```

```

#      Min     1Q Median     3Q    Max
# -4.869 -1.465 -0.030  1.248  4.916
#
# Coefficients:
#                               Estimate Std. Error t value Pr(>|t|)
# (Intercept)           1.209e+01  2.958e+00  4.086 0.000275 ***
# body                  6.658e-03  4.243e-02  0.157 0.876293
# brain                 -5.243e-02 2.261e-02 -2.319 0.026933 *
# lifespan                1.976e-01 1.048e-01  1.886 0.068388 .
# gestation               -3.017e-02 1.479e-02 -2.040 0.049698 *
# predation                5.405e+00 1.636e+00  3.304 0.002352 **
# exposure                1.145e+00 1.796e+00  0.637 0.528391
# danger                 -3.918e+00 1.580e+00 -2.479 0.018627 *
# body:exposure            3.865e-02 1.534e-02  2.519 0.016976 *
# body:danger               -4.048e-02 1.381e-02 -2.931 0.006196 **
# brain:lifespan            2.486e-04 1.515e-04  1.641 0.110494
# brain:gestation            -1.687e-04 6.262e-05 -2.694 0.011161 *
# brain:predation             -3.363e-02 1.356e-02 -2.480 0.018575 *
# brain:danger                5.263e-02 1.787e-02  2.944 0.005983 **
# lifespan:gestation          2.609e-03 8.945e-04  2.916 0.006421 **
# lifespan:exposure            -1.553e-01 6.087e-02 -2.551 0.015717 *
# gestation:exposure           -7.685e-03 5.839e-03 -1.316 0.197461
# predation:danger              -1.027e+00 4.621e-01 -2.222 0.033493 *
# exposure:danger                6.829e-01 4.215e-01  1.620 0.115018
#
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 2.621 on 32 degrees of freedom
# Multiple R-squared:  0.7992, Adjusted R-squared:  0.6862
# F-statistic: 7.074 on 18 and 32 DF,  p-value: 9.703e-07

```

We would not have selected this model based on classical techniques.

## 20 Analysis of variance and analysis of covariance

### 20.1 Analysis of variance (ANOVA)

The same `lm()` function that we used for linear regression can also be used for analysis of variance (ANOVA). However, it is probably more straightforward to use the `aov()` function, which is a “wrapper” for `lm()`. The main difference between `aov()` and `lm()` is in the format of the output, although a traditional ANOVA table can be produced by applying the `anova()` function to an `lm` model. The `aov` function also provides some handy options for ANOVA, including the ability to specify an error term for hypothesis testing.

To demonstrate ANOVA in R, let’s start with a simple data set distributed with the base packages called `InsectSprays`. This data set shows the effectiveness of six different insecticides.

```
insects <- InsectSprays
summary(insects)

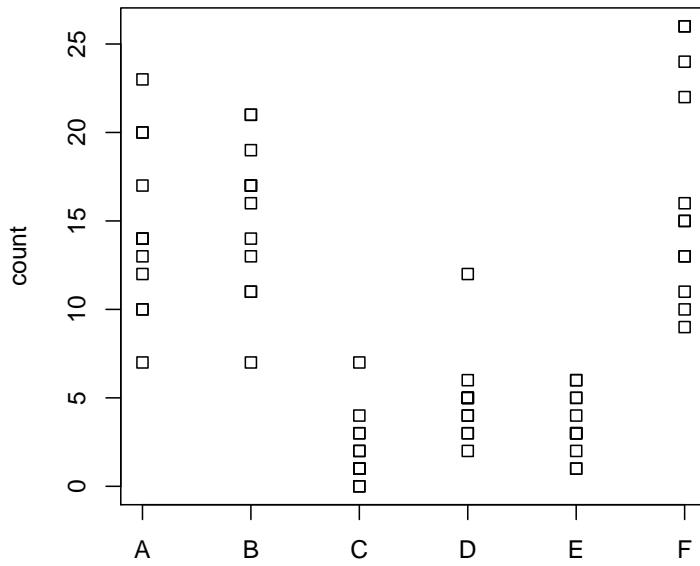
#      count     spray
# Min.   : 0.00   A:12
# 1st Qu.: 3.00   B:12
# Median : 7.00   C:12
# Mean   : 9.50   D:12
# 3rd Qu.:14.25   E:12
# Max.   :26.00   F:12

dfsumm(insects)

#
# 72 rows and 2 columns
# 43 unique rows
#           count     spray
# Class       numeric factor
# Minimum          0       A
# Maximum          26      F
# Mean            9.5      D
# Unique (excl. NA) 24      6
# Missing values      0      0
# Sorted           FALSE    TRUE
```

Let’s take a look at the data.

```
stripchart(count ~ spray, data = insects, vertical = TRUE)
```

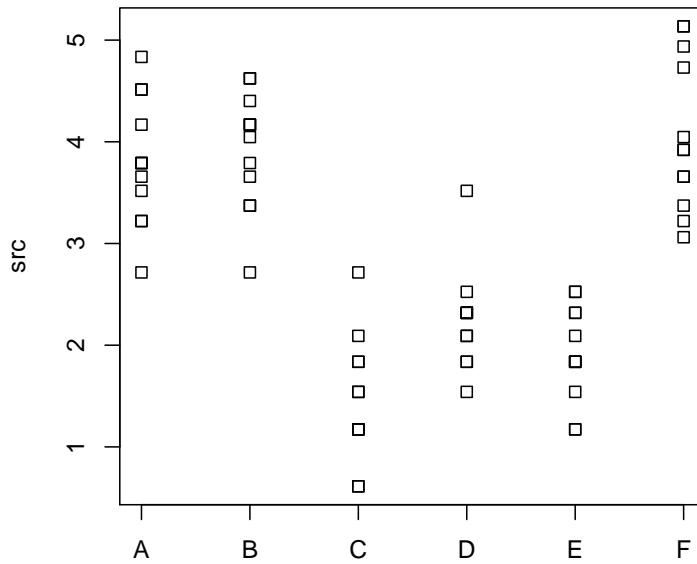


There seem to be some clear differences among sprays. Since the measured variable is a count (number of insects), it is not normally distributed. To make these data approximate a normal distribution, we can use a square root transformation<sup>129</sup> [29].

```
insects$src <- sqrt(insects$count + 3/8)
stripchart(src ~ spray, data = insects, vertical = TRUE)
```

---

<sup>129</sup> Another option is to use Poisson regression through the `glm()` function instead of a linear model. See Section 21 for more information.



Let's fit the model.

```
mod1 <- aov(src ~ spray, data = insects)
```

To get more detailed output, we can use the `summary()` function.

```
summary(mod1)

#           Df Sum Sq Mean Sq F value Pr(>F)
# spray      5 80.53 16.106  46.62 <2e-16 ***
# Residuals  66 22.80   0.345
# ---
# Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

So we see that `spray` had a highly significant effect on `count`.

We can extract confidence intervals for groups (or really, for parameter estimates) using the `confint()` function.

```
confint(mod1)

#           2.5 %    97.5 %
# (Intercept) 3.4727146 4.1502674
# sprayB     -0.3647920 0.5934123
# sprayC     -2.8340142 -1.8758099
# sprayD     -2.0378140 -1.0796097
```

```
# sprayE      -2.3728438 -1.4146395
# sprayF      -0.2241445  0.7340598
```

But note that these are intervals for model coefficients, which are all relative to the reference level in this model.

```
coef(mod1)

# (Intercept)      sprayB      sprayC      sprayD      sprayE
# 3.8114910   0.1143102  -2.3549121  -1.5587119  -1.8937416
# sprayF
# 0.2549576
```

To get confidence intervals that correspond to the means returned above, remove the intercept from the model.

```
mod1b <- aov(src ~ spray - 1, data = insects)
summary(mod1b)

#           Df Sum Sq Mean Sq F value Pr(>F)
# spray       6  688.2  114.70     332 <2e-16 ***
# Residuals 66    22.8     0.35
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

confint(mod1b)

#           2.5 %  97.5 %
# sprayA 3.472715 4.150267
# sprayB 3.587025 4.264578
# sprayC 1.117803 1.795355
# sprayD 1.914003 2.591555
# sprayE 1.578973 2.256526
# sprayF 3.727672 4.405225
```

We can specify this same model using the `lm()` function.

```
mod2 <- lm(src ~ spray, data = insects)
summary(mod2)

#
# Call:
# lm(formula = src ~ spray, data = insects)
#
# Residuals:
#       Min     1Q Median     3Q    Max
# -1.21011 -0.38480 -0.02005  0.38054  1.26503
#
```

```

# Coefficients:
#              Estimate Std. Error t value Pr(>|t|)
# (Intercept) 3.8115    0.1697  22.463 < 2e-16 ***
# sprayB      0.1143    0.2400   0.476   0.635
# sprayC     -2.3549    0.2400  -9.814 1.60e-14 ***
# sprayD     -1.5587    0.2400  -6.496 1.26e-08 ***
# sprayE     -1.8937    0.2400  -7.892 4.14e-11 ***
# sprayF      0.2550    0.2400   1.062   0.292
# ---
# Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 0.5878 on 66 degrees of freedom
# Multiple R-squared:  0.7793, Adjusted R-squared:  0.7626
# F-statistic: 46.62 on 5 and 66 DF, p-value: < 2.2e-16

```

To get the same output that `aov()` returns, you can use `anova()` on the `lm` object.

```

anova(mod2)

# Analysis of Variance Table
#
# Response: src
#           Df Sum Sq Mean Sq F value    Pr(>F)
# spray      5 80.528 16.1057 46.616 < 2.2e-16 ***
# Residuals 66 22.803  0.3455
# ---
# Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

The output from `summary()` might give you some insight into how R carries out ANOVA—it uses `lm` with a binary incidence matrix (also known as a dummy variable matrix). You can get more information on the variable coding with the `model.matrix()` function, which returns the `X` matrix for the regression.

```

head(model.matrix(mod1))

#   (Intercept) sprayB sprayC sprayD sprayE sprayF
# 1          1     0     0     0     0     0
# 2          1     0     0     0     0     0
# 3          1     0     0     0     0     0
# 4          1     0     0     0     0     0
# 5          1     0     0     0     0     0
# 6          1     0     0     0     0     0

```

From the output, it is clear that `spray = A` is taken as the reference level, and all `spray` coefficients will be relative to the response for `A`. Exactly what R does is flexible, however.

With `lm()` and associated wrapper functions, we can fit a model that does not have an intercept by including `-1` on the right side of the model formula.

```

mod3 <- lm(src ~ spray - 1, data = insects)
coef(mod3)

#   sprayA   sprayB   sprayC   sprayD   sprayE   sprayF
# 3.811491 3.925801 1.456579 2.252779 1.917749 4.066449

```

Now we have an explicit coefficient for `spray = A` and mean results for each group. But our hypothesis tests are not very meaningful.

```

summary(mod3)

#
# Call:
# lm(formula = src ~ spray - 1, data = insects)
#
# Residuals:
#       Min        1Q    Median        3Q       Max
# -1.21011 -0.38480 -0.02005  0.38054  1.26503
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# sprayA     3.8115    0.1697 22.463 < 2e-16 ***
# sprayB     3.9258    0.1697 23.137 < 2e-16 ***
# sprayC     1.4566    0.1697  8.584 2.39e-12 ***
# sprayD     2.2528    0.1697 13.277 < 2e-16 ***
# sprayE     1.9177    0.1697 11.302 < 2e-16 ***
# sprayF     4.0664    0.1697 23.965 < 2e-16 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 0.5878 on 66 degrees of freedom
# Multiple R-squared:  0.9679, Adjusted R-squared:  0.965
# F-statistic:  332 on 6 and 66 DF,  p-value: < 2.2e-16

```

```

anova(mod3)

#
# Analysis of Variance Table
#
# Response: src
#           Df Sum Sq Mean Sq F value    Pr(>F)
# spray      6 688.2 114.700 331.99 < 2.2e-16 ***
# Residuals 66  22.8   0.345
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

We can also modify the way that our **X** matrix is created, specifically, we can change how R creates the contrast matrices. There are several different types of contrasts that can be used in R—you can see a list in the help file for `contrast`. The default behavior of R is to use `contr.treatment()` contrasts for unordered factors and `contr.poly` for ordered factors. You can check and change these settings with the `options()` function.

```
options()$contrasts

#      unordered      ordered
# "contr.treatment" "contr.poly"
```

So in our example, we used treatment contrasts, with the `contr.treatment()` function. Here, each level of `spray` is compared to the first level, which is A here<sup>130</sup>

The default “reference” level is the first one listed.

```
levels(insects$spray)

# [1] "A" "B" "C" "D" "E" "F"
```

We can change the order with `factor()` or `relevel()`.

```
insects$spray2 <- relevel(insects$spray, ref = "F")

levels(insects$spray2)

# [1] "F" "A" "B" "C" "D" "E"
```

We can look at some alternatives for contrasts.

```
mod4 <- lm(src ~ spray, data = insects, contrasts = list(spray = "contr.helmert"))
```

In the call above, we set the contrasts locally—just for that model. We can also do it globally.

```
oldcontr <- options(contrasts = c("contr.helmert", "contr.poly"))
options()$contrasts

# [1] "contr.helmert" "contr.poly"
```

And now we could fit an identical model without explicitly specifying the contrasts.

```
mod4 <- lm(src ~ spray, data = insects)
```

Before continuing, let’s set the contrasts back to their defaults.

```
options(oldcontr)
options()$contrasts

#      unordered      ordered
# "contr.treatment" "contr.poly"
```

<sup>130</sup> With no correction for multiple tests, such as is done with multiple range tests.

Regardless of how it is set, this option gives Helmert contrasts, which, unlike treatment contrasts, are orthogonal. Using them usually makes interpretation more difficult. For the hypothesis tests returned by `summary()`, we are comparing the first spray to the intercept term, then the second spray to the first one, the third to the mean of the first and second sprays, the fourth to the mean of the first, second, and third sprays, and so on.

When it comes to overall tests for a term, e.g., `spray` in this model, contrast type has no effect. See?

```
anova(mod1)
```

```
# Analysis of Variance Table
#
# Response: src
#           Df Sum Sq Mean Sq F value    Pr(>F)
# spray      5 80.528 16.1057 46.616 < 2.2e-16 ***
# Residuals 66 22.803  0.3455
# ---
# Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
anova(mod4)
```

```
# Analysis of Variance Table
#
# Response: src
#           Df Sum Sq Mean Sq F value    Pr(>F)
# spray      5 80.528 16.1057 46.616 < 2.2e-16 ***
# Residuals 66 22.803  0.3455
# ---
# Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

For ordered factors, R uses orthogonal polynomial contrasts. This approach also produces easy-to-interpret results, where the individual coefficients are independent of each other. The default contrasts probably make the most sense in many cases<sup>131</sup>. Maybe the most common problem you will encounter when dealing with contrasts for unordered factors is in the selection of the “base” group. By default, it will be taken as the first level in the factor, so, here:

```
levels(insects$spray)[1]
```

```
# [1] "A"
```

But maybe `spray = A` isn’t the control treatment, and so it doesn’t make a lot of sense to compare all the other sprays to it. To change the base level, you can either change the order of levels in the factor with the `relevel()` or `factor()` functions, or else specify the base in a call to `contr.treatment()`. The first option is described above in Section 8<sup>132</sup>, and the second option is shown below.

<sup>131</sup> But clearly reasonable individuals (smarter than me) would not agree with that statement, since other contrasts are the default in other software.

<sup>132</sup> You could make level C show up first with this command: `insects$spray <- factor(insects$spray, levels = c('C', 'A', 'B', 'D', 'E', 'F'))`

```
mod5 <- lm(src ~ spray, data = insects,
           contrasts = list(spray = contr.treatment(levels(insects$spray), base = 3)))
```

Now `spray = C` is the base level.

```
summary(mod5)
```

```
#  
# Call:  
# lm(formula = src ~ spray, data = insects, contrasts = list(spray = contr.treatment(levels(insects$spray),  
#           base = 3)))  
#  
# Residuals:  
#       Min     1Q   Median     3Q    Max  
# -1.21011 -0.38480 -0.02005  0.38054  1.26503  
#  
# Coefficients:  
#             Estimate Std. Error t value Pr(>|t|)  
# (Intercept)  1.4566    0.1697  8.584 2.39e-12 ***  
# sprayA      2.3549    0.2400  9.814 1.60e-14 ***  
# sprayB      2.4692    0.2400 10.290 2.37e-15 ***  
# sprayD      0.7962    0.2400  3.318  0.00148 **  
# sprayE      0.4612    0.2400  1.922  0.05895 .  
# sprayF      2.6099    0.2400 10.876 2.33e-16 ***  
# ---  
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1  
#  
# Residual standard error: 0.5878 on 66 degrees of freedom  
# Multiple R-squared:  0.7793, Adjusted R-squared:  0.7626  
# F-statistic: 46.62 on 5 and 66 DF,  p-value: < 2.2e-16
```

Even with a “control” factor level, using a fixed alpha value for multiple tests means that the probability of falsely rejecting the null hypothesis in at least one case is higher than alpha (possibly much higher, depending on the number of comparisons). The easiest way around this problem is the Bonferroni adjustment: divide alpha by the number of comparisons. Here, for example, if we previously specified an alpha of 0.05, we would use  $0.05/6 = 0.00833$  to evaluate each comparison to the control. This approach only works well for a small number of comparisons. If you have no “control” treatment (probably true in this case), treatment contrasts don’t make a lot of sense. Instead, Tukey’s method is more appropriate. The Tukey test is applied using the `TukeyHSD()` function <sup>133</sup>. Note that this function requires `aov()` output—output from `lm()` will not work.

```
TukeyHSD(mod1)
```

```
# Tukey multiple comparisons of means  
# 95% family-wise confidence level  
#  
# Fit: aov(formula = src ~ spray, data = insects)  
#  
# $spray
```

<sup>133</sup> Many other tests are available in the `multcomp` package.

```

#          diff      lwr      upr      p adj
# B-A  0.1143102 -0.59000479  0.8186251 0.9968245
# C-A -2.3549121 -3.05922701 -1.6505971 0.0000000
# D-A -1.5587119 -2.26302685 -0.8543969 0.0000002
# E-A -1.8937416 -2.59805660 -1.1894267 0.0000000
# F-A  0.2549576 -0.44935734  0.9592726 0.8943236
# C-B -2.4692222 -3.17353717 -1.7649073 0.0000000
# D-B -1.6730221 -2.37733701 -0.9687071 0.0000000
# E-B -2.0080518 -2.71236676 -1.3037369 0.0000000
# F-B  0.1406474 -0.56366751  0.8449624 0.9916328
# D-C  0.7962002  0.09188521  1.5005151 0.0177353
# E-C  0.4611704 -0.24314454  1.1654854 0.3983576
# F-C  2.6098697  1.90555471  3.3141846 0.0000000
# E-D -0.3350298 -1.03934471  0.3692852 0.7291427
# F-D  1.8136695  1.10935455  2.5179845 0.0000000
# F-E  2.1486993  1.44438430  2.8530142 0.0000000

```

A two-factor or multi-factor ANOVA is carried out in a similar way. Let's use some data from Zar [29] on the respiration rate of three species of crabs in response to temperature to demonstrate.

```

respir <- read.csv("../data/respiration.csv")
dfsumm(respir)

#
# 72 rows and 4 columns
# 69 unique rows
#
#          sp      temp     sex     resp
# Class       integer factor factor numeric
# Minimum        1    high      F      1
# Maximum        3    med      M     3.6
# Mean           2    low      M     2.33
# Unique (excl. NA) 3      3      2     24
# Missing values 0      0      0      0
# Sorted         TRUE   FALSE  FALSE FALSE

respir$sp <- factor(respir$sp)
summary(respir)

#
#   sp      temp     sex     resp
# 1:24  high:24  F:36  Min.  :1.000
# 2:24  low :24  M:36  1st Qu.:1.900
# 3:24  med :24          Median :2.300
#                  Mean   :2.325
#                  3rd Qu.:2.900
#                  Max.  :3.600

```

The `aov()` function is designed for balanced designs. Let's check for balance using the `table()` function<sup>134</sup>, which is used to summarize counts.

---

<sup>134</sup> `table()` creates a contingency table, and is used for summarizing counts. The `replications()` function is an alternative, but I think it is less clear.

```



```

So it is a balanced and fully crossed design, with 4 replicates for each combination. Let's change the sorting order for `temp`.

```
respir$temp <- factor(respir$temp, levels = c("low", "med", "high"))
```

Now for the ANOVA.

```

mod1 <- aov(resp ~ sp + temp + sex, data = respir)
summary(mod1)

#           Df Sum Sq Mean Sq F value    Pr(>F)
# sp          2  1.818   0.909  15.487 3.06e-06 ***
# temp        2 24.656  12.328 210.093  < 2e-16 ***
# sex          1  0.009   0.009   0.151    0.698
# Residuals   66  3.873   0.059
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

Results from this model suggest that respiration rate is not dependent on `sex`. Let's add some interaction terms to explore this result more fully.

```
mod2 <- aov(resp ~ (sp + temp + sex)^3, data = respir)
summary(mod2)
```

```

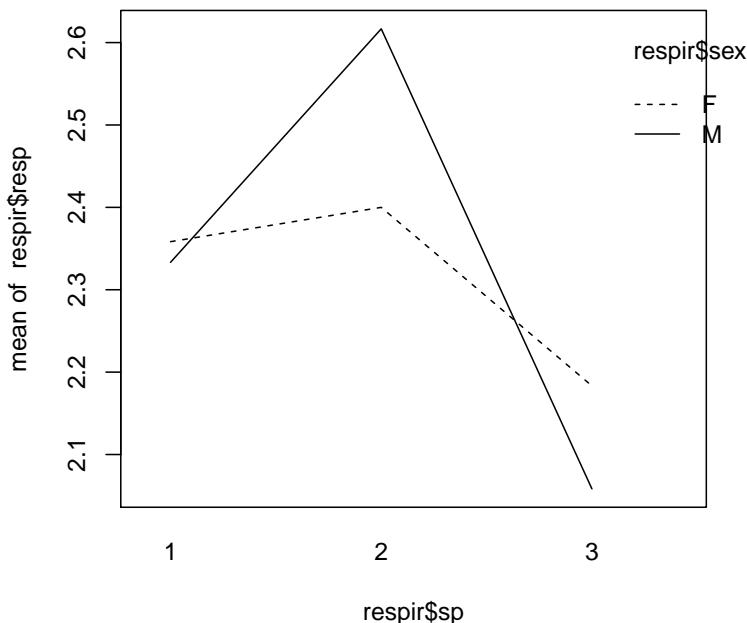
#           Df Sum Sq Mean Sq F value    Pr(>F)
# sp          2  1.818   0.909  24.475 2.71e-08 ***
# temp        2 24.656  12.328 332.024  < 2e-16 ***
# sex          1  0.009   0.009   0.239    0.6266
# sp:temp     4  1.102   0.275    7.418 7.75e-05 ***
# sp:sex      2  0.370   0.185    4.986   0.0103 *
# temp:sex    2  0.175   0.088    2.360   0.1041

```

```
# sp:temp:sex  4  0.221   0.055   1.485   0.2196
# Residuals   54  2.005   0.037
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

So, our analysis shows that respiration differs with species and temperature, and that the effect of temperature is dependent on the species. Additionally, the difference in respiration rates between sexes is dependent on species (the main effect is actually not significant). What do the results actually look like? We can use the function `interaction.plot()` to quickly generate some illuminating plots<sup>135</sup>.

```
interaction.plot(respir$sp, respir$sex, respir$resp)
```



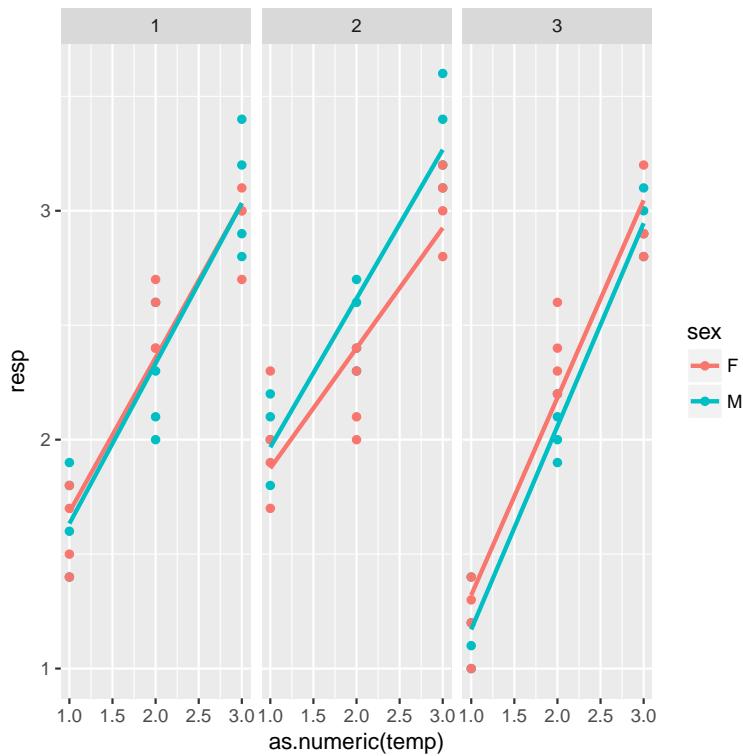
This plot shows why we don't see a significant main effect for `sex` (although this test is more-or-less irrelevant, considering that the interaction is significant).

Better-looking plots can be produced with the base graphics functions, but this factorial experiment is a good fit for functions in the `ggplot2` package.

```
library(ggplot2)

ggplot(respir, aes(as.numeric(temp), resp, colour = sex)) +
  facet_wrap(~ sp) + geom_point() +
  geom_smooth(method = "lm", se = FALSE)
```

<sup>135</sup> A shorter option is `with(respir, interaction.plot(sp, temp, resp))`.



If we want to see the actual mean respiration rates for each combination of levels, we can use the `model.tables()` function. The output from this function is a list with a class of `tables.aov`, and you can extract or manipulate data in it as with any other list.

```
model.tables(mod2, type = "means")

# Tables of means
# Grand mean
#
# 2.325
#
#   sp
#   sp
#       1      2      3
# 2.3458 2.5083 2.1208
#
#   temp
#   temp
#       low     med     high
# 1.6125 2.3167 3.0458
#
#   sex
#   sex
#       F      M
# 2.3139 2.3361
#
#   sp:temp
#   temp
# sp  low    med    high
```

```

#   1 1.638 2.388 3.013
#   2 2.000 2.350 3.175
#   3 1.200 2.213 2.950
#
#   sp:sex
#       sex
# sp   F      M
#   1 2.3583 2.3333
#   2 2.4000 2.6167
#   3 2.1833 2.0583
#
#   temp:sex
#       sex
# temp   F      M
#   low  1.6000 1.6250
#   med   2.3667 2.2667
#   high 2.9750 3.1167
#
#   sp:temp:sex
# , , sex = F
#
#       temp
# sp  low   med   high
#   1 1.600 2.525 2.950
#   2 1.975 2.200 3.025
#   3 1.225 2.375 2.950
#
# , , sex = M
#
#       temp
# sp  low   med   high
#   1 1.675 2.250 3.075
#   2 2.025 2.500 3.325
#   3 1.175 2.050 2.950

```

Note that these means are the actual, arithmetic means, not the marginal means (known to SAS users as “least-squares means”)—the two will only differ with unbalanced data. To get marginal means, the best bet may be to just use `predict()` with a new data frame. There is also an `lsmeans` package.

As with regression models, we can also apply the `plot()` function to the model output to get diagnostic plots.

Let’s back up a bit and try to get a better understanding of analysis of variance in R. Remember that `aov()` is just a wrapper for `lm()`. So, let’s try `lm` itself.

```

mod5 <- lm(resp~(sp+temp+sex)^2, data = respir)
summary(mod5)

#
# Call:
# lm(formula = resp ~ (sp + temp + sex)^2, data = respir)
#

```

```

# Residuals:
#      Min       1Q   Median      3Q      Max
# -0.31389 -0.14410 -0.00486  0.13576  0.40972
#
# Coefficients:
#                  Estimate Std. Error t value Pr(>|t|)
# (Intercept)    1.64861   0.08638 19.086 < 2e-16 ***
# sp2            0.24167   0.11310  2.137  0.03684 *
# sp3           -0.38750   0.11310 -3.426  0.00113 **
# tempmed        0.81250   0.11310  7.184 1.44e-09 ***
# temphigh       1.31667   0.11310 11.642 < 2e-16 ***
# sexM          -0.02222   0.10324 -0.215  0.83033
# sp2:tempmed   -0.40000   0.13851 -2.888  0.00544 **
# sp3:tempmed   0.26250   0.13851  1.895  0.06306 .
# sp2:temphigh  -0.20000   0.13851 -1.444  0.15414
# sp3:temphigh  0.37500   0.13851  2.707  0.00890 **
# sp2:sexM       0.24167   0.11310  2.137  0.03684 *
# sp3:sexM       -0.10000   0.11310 -0.884  0.38023
# tempmed:sexM  -0.12500   0.11310 -1.105  0.27361
# temphigh:sexM 0.11667   0.11310  1.032  0.30655
#
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 0.1959 on 58 degrees of freedom
# Multiple R-squared:  0.9267, Adjusted R-squared:  0.9102
# F-statistic: 56.39 on 13 and 58 DF, p-value: < 2.2e-16

```

To generate an ANOVA table, we can use the `anova()` function.

```

anova(mod5)

# Analysis of Variance Table
#
# Response: resp
#              Df  Sum Sq Mean Sq  F value    Pr(>F)
# sp            2  1.8175  0.9088 23.6829 3.028e-08 ***
# temp          2 24.6558 12.3279 321.2767 < 2.2e-16 ***
# sex           1  0.0089  0.0089  0.2317  0.63211
# sp:temp       4  1.1017  0.2754  7.1776 9.136e-05 ***
# sp:sex         2  0.3703  0.1851  4.8249  0.01153 *
# temp:sex      2  0.1753  0.0876  2.2839  0.11097
# Residuals    58  2.2256  0.0384
#
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

This should be identical to what `summary(aov(mod4))` returned above.

Don't forget that the ANOVA table that you get with `lm()` objects (e.g. from `aov()`) in R returns tests based on Type I SS. We can demonstrate that this is the case. First, let's try dropping the last interaction term:

```
mod6 <- update(mod5, ~. - temp:sex)
```

Now, we can compare the two models with `anova()`.

```
anova(mod5, mod6)

# Analysis of Variance Table
#
# Model 1: resp ~ (sp + temp + sex)^2
# Model 2: resp ~ sp + temp + sex + sp:temp + sp:sex
#   Res.Df   RSS Df Sum of Sq    F Pr(>F)
# 1      58 2.2256
# 2      60 2.4008 -2  -0.17528 2.2839  0.111
```

This result is identical to what we got above (well, with a different number of digits).

If you have an unbalanced design, Type I SS are not what you want. What can you do? You can use `update()` and `anova()` to compare any two nested models, which allows you to look at the effect of any one variable, given any number of other variables in the model. For Type III SS, you can use the `drop1()` function, which returns results for the effect of single variables given all other variables in the model. However, this function will not return results for “main” effects if interactions are included in your model—this is a good thing<sup>136</sup>!

```
drop1(mod5, test = "F")

# Single term deletions
#
# Model:
# resp ~ (sp + temp + sex)^2
#           Df Sum of Sq   RSS     AIC F value    Pr(>F)
# <none>            2.2256 -222.32
# sp:temp    4   1.10167 3.3272 -201.37  7.1776 9.136e-05 ***
# sp:sex     2   0.37028 2.5958 -215.24  4.8249  0.01153 *
# temp:sex   2   0.17528 2.4008 -220.86  2.2839  0.11097
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Lastly, there are other functions available for carrying out ANOVA with Type II SS, e.g., `Anova()` in the `car` package.

## 20.2 Analysis of covariance (ANCOVA)

Analysis of covariance (ANCOVA) is useful when you have both categorical and continuous predictor variables. In R, ANCOVA can be carried out using the `lm` function.

To demonstrate ANCOVA, we will use a data set on copper toxicity to *Daphnia magna* from Ryan (2005) [19]. The experimental design is a 7 by 3 by 3 factorial design, with 7 dissolved organic

<sup>136</sup> And, some R users might say, a shortcoming of SAS. There is a lot of discussion in the R archives on ANOVA SS. If you are confused, check out: <http://cran.r-project.org/doc/FAQ/R-FAQ.html>, <https://stat.ethz.ch/pipermail/r-help/2008-February/153740.html>, <http://markmail.org/message/pjicdzsxdjzvs6en>.

matter (DOM) sources, 3 DOM concentrations (measured as DOC—dissolved organic carbon), and 3 pH levels. The experiment was designed to be balanced, but it is not possible to achieve the exact pH values and DOC concentrations that are desired. The column names that start with `n` are the “nominal” (i.e., target) values. Copper toxicity (expressed as LC50) was measured for each combination of levels.

```
tox <- read.csv("../data/ogeechee_tox.csv")
dfsumm(tox)

#
# 126 rows and 8 columns
# 126 unique rows
#          test   n.doc   n.ph    rep dom.source     ph
# Class      integer integer integer integer integer numeric
# Minimum        2       2       6       1       1      5.86
# Maximum       144      15       8       2       7      8.4
# Mean          72       7       7       1       4      7.15
# Unique (excl. NA) 126      3       3       2       7      67
# Missing values 0       0       0       0       0      0
# Sorted        FALSE    FALSE   TRUE  FALSE  FALSE FALSE
#          doc    lc50
# Class      numeric numeric
# Minimum        2      5.42
# Maximum       16.1     575
# Mean          7.97     126
# Unique (excl. NA) 98     126
# Missing values 0      0
# Sorted        FALSE    FALSE
```

We need to make `dom.source` a factor.

```
tox$dom.source <- factor(tox$dom.source)
```

And, since solute concentrations and LC50s are generally log-normally distributed, and since we might expect that log LC50 is proportional to log DOC concentration, we should log transform `doc` and `lc50`.

```
tox$ldoc <- log10(tox$doc)
tox$llc50 <- log10(tox$lc50)
```

Checking balance.

```
table(tox$dom.source, tox$n.doc, tox$n.ph)
```

```
# , , = 6
#
#
#      2 7 15
#      1 2 2 2
#      2 2 2 2
#      3 2 2 2
```

```

#   4 2 2 2
#   5 2 2 2
#   6 2 2 2
#   7 2 2 2
#
# , , = 7
#
#
#   2 7 15
#   1 2 2 2
#   2 2 2 2
#   3 2 2 2
#   4 2 2 2
#   5 2 2 2
#   6 2 2 2
#   7 2 2 2
#
# , , = 8
#
#
#   2 7 15
#   1 2 2 2
#   2 2 2 2
#   3 2 2 2
#   4 2 2 2
#   5 2 2 2
#   6 2 2 2
#   7 2 2 2

```

Let's start with

```

mod <- lm(llc50 ~ (dom.source + ldoc + ph)^2, data = tox)
summary(mod)

#
# Call:
# lm(formula = llc50 ~ (dom.source + ldoc + ph)^2, data = tox)
#
# Residuals:
#       Min         1Q     Median        3Q       Max
# -0.259978 -0.041663  0.009425  0.053572  0.150076
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)    
# (Intercept) -3.81833   0.27024 -14.129 < 2e-16 ***
# dom.source2  0.51286   0.28910   1.774 0.078990 .  
# dom.source3  1.01760   0.29146   3.491 0.000706 *** 
# dom.source4  0.68690   0.29100   2.360 0.020117 *  
# dom.source5  0.64660   0.28754   2.249 0.026638 *  
# dom.source6  0.09499   0.28672   0.331 0.741085    
# dom.source7  0.31169   0.28617   1.089 0.278596    
# ldoc         1.88073   0.22770   8.260 4.97e-13 ***
# ph          0.69521   0.03737  18.603 < 2e-16 ***

```

```

# dom.source2:ldoc  0.15330   0.08720   1.758 0.081675 .
# dom.source3:ldoc  0.10787   0.08751   1.233 0.220445
# dom.source4:ldoc  0.09537   0.08749   1.090 0.278206
# dom.source5:ldoc  0.10160   0.08756   1.160 0.248566
# dom.source6:ldoc  0.15615   0.08853   1.764 0.080687 .
# dom.source7:ldoc  0.12921   0.08846   1.461 0.147159
# dom.source2:ph    -0.10491   0.03950   -2.656 0.009149 **
# dom.source3:ph    -0.16600   0.03983   -4.168 6.37e-05 ***
# dom.source4:ph    -0.11637   0.03968   -2.933 0.004133 **
# dom.source5:ph    -0.11465   0.03916   -2.927 0.004200 **
# dom.source6:ph    -0.03673   0.03906   -0.940 0.349202
# dom.source7:ph    -0.06406   0.03896   -1.644 0.103172
# ldoc:ph           -0.13452   0.03060   -4.396 2.66e-05 ***
#
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 0.0898 on 104 degrees of freedom
# Multiple R-squared:  0.9776, Adjusted R-squared:  0.9731
# F-statistic: 216.5 on 21 and 104 DF, p-value: < 2.2e-16

```

`anova(mod)`

```

# Analysis of Variance Table
#
# Response: llc50
#              Df  Sum Sq Mean Sq  F value    Pr(>F)
# dom.source      6  0.1962  0.0327   4.0550 0.0010804 ***
# ldoc            1 16.6982 16.6982 2070.9118 < 2.2e-16 ***
# ph              1 19.3704 19.3704 2402.3121 < 2.2e-16 ***
# dom.source:ldoc 6  0.0306  0.0051   0.6330 0.7035138
# dom.source:ph   6  0.2069  0.0345   4.2759 0.0006842 ***
# ldoc:ph         1  0.1558  0.1558   19.3281 2.664e-05 ***
# Residuals       104 0.8386  0.0081
#
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

Dropping insignificant predictors,

```

mod2 <- lm(llc50 ~ (dom.source + ldoc + ph)^2 - dom.source:ldoc, data = tox)
anova(mod2)

```

```

# Analysis of Variance Table
#
# Response: llc50
#              Df  Sum Sq Mean Sq  F value    Pr(>F)
# dom.source      6  0.1962  0.0327   4.1184 0.0009083 ***
# ldoc            1 16.6982 16.6982 2103.2746 < 2.2e-16 ***
# ph              1 19.3704 19.3704 2439.8537 < 2.2e-16 ***
# dom.source:ph   6  0.2006  0.0334   4.2110 0.0007485 ***
# ldoc:ph         1  0.1580  0.1580   19.9033 1.976e-05 ***
# Residuals       110 0.8733  0.0079

```

```
# ---
# Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

We can conclude that DOM source is a significant categorical predictor, and that DOC concentration and pH both have a significant linear effect on LC50, although the slope of the response to pH differs among DOM sources and the slope of the response to DOC concentration differs in response pH values (or vice versa). Let's look at regression parameters.

```
summary(mod2)
```

```
# 
# Call:
# lm(formula = llc50 ~ (dom.source + ldoc + ph)^2 - dom.source:ldoc,
#      data = tox)
#
# Residuals:
#       Min     1Q Median     3Q    Max
# -0.265161 -0.050208  0.008665  0.056682  0.152552
#
# 
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)    
# (Intercept) -3.87739   0.26625 -14.563 < 2e-16 ***
# dom.source2  0.59756   0.28238   2.116 0.036589 *  
# dom.source3  1.07257   0.28462   3.768 0.000266 *** 
# dom.source4  0.73278   0.28372   2.583 0.011113 *  
# dom.source5  0.69673   0.27989   2.489 0.014298 *  
# dom.source6  0.18511   0.27915   0.663 0.508637    
# dom.source7  0.38226   0.27854   1.372 0.172742    
# ldoc         1.99341   0.21759   9.161 3.26e-15 ***
# ph           0.69205   0.03704  18.682 < 2e-16 ***
# dom.source2:ph -0.10018  0.03912  -2.561 0.011786 *  
# dom.source3:ph -0.16208  0.03945  -4.109 7.68e-05 *** 
# dom.source4:ph -0.11257  0.03931  -2.864 0.005014 **  
# dom.source5:ph -0.11075  0.03880  -2.854 0.005155 ** 
# dom.source6:ph -0.03245  0.03870  -0.838 0.403597    
# dom.source7:ph -0.06000  0.03860  -1.554 0.122970    
# ldoc:ph        -0.13543  0.03036  -4.461 1.98e-05 *** 
# --- 
# Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 0.0891 on 110 degrees of freedom
# Multiple R-squared:  0.9767, Adjusted R-squared:  0.9735 
# F-statistic: 307.5 on 15 and 110 DF,  p-value: < 2.2e-16
```

For the coefficients only:

```
coef(mod2)
```

```
# (Intercept)  dom.source2  dom.source3  dom.source4
# -3.87739361  0.59755833  1.07256669  0.73278355
# dom.source5  dom.source6  dom.source7  ldoc
```

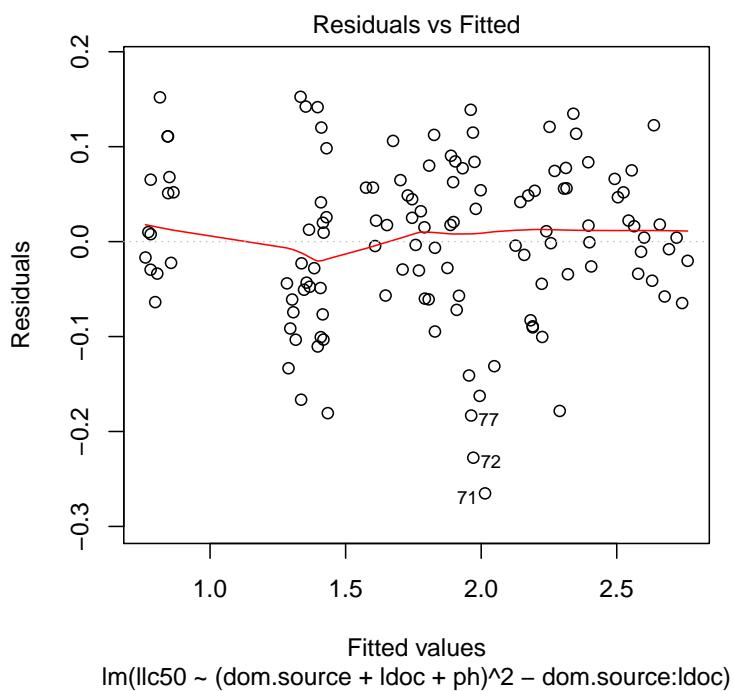
```

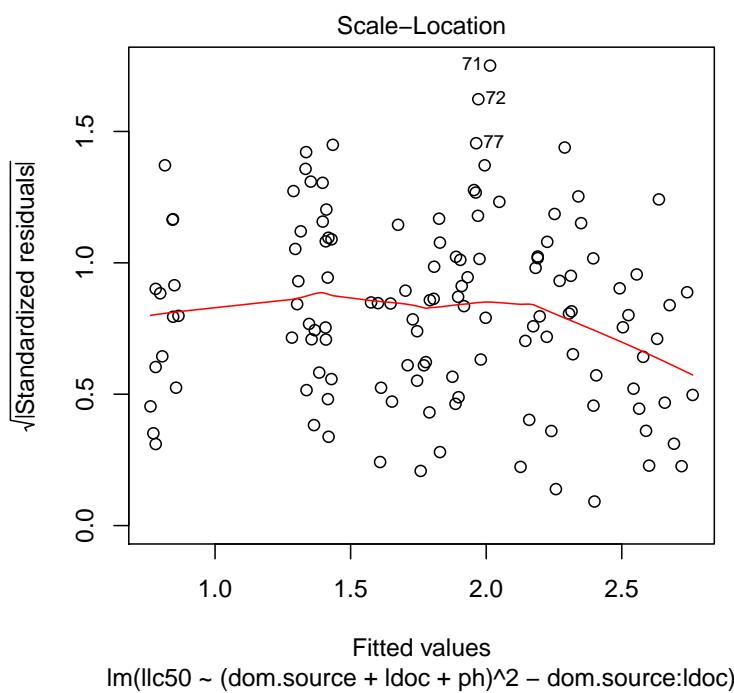
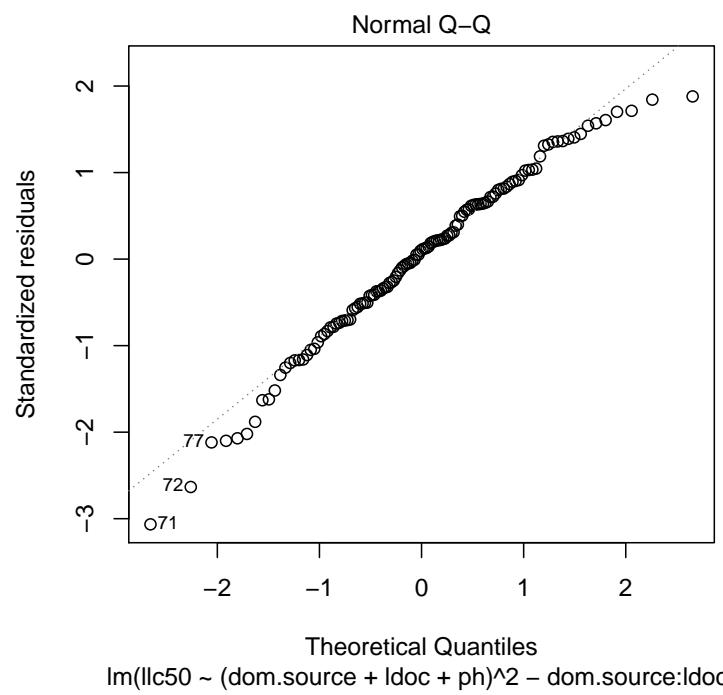
#      0.69672694    0.18510862    0.38225760    1.99340957
#          ph dom.source2:ph dom.source3:ph dom.source4:ph
#      0.69204731   -0.10018229   -0.16208067   -0.11256778
# dom.source5:ph dom.source6:ph dom.source7:ph      ldoc:ph
#   -0.11074601   -0.03244520   -0.05999614   -0.13543281

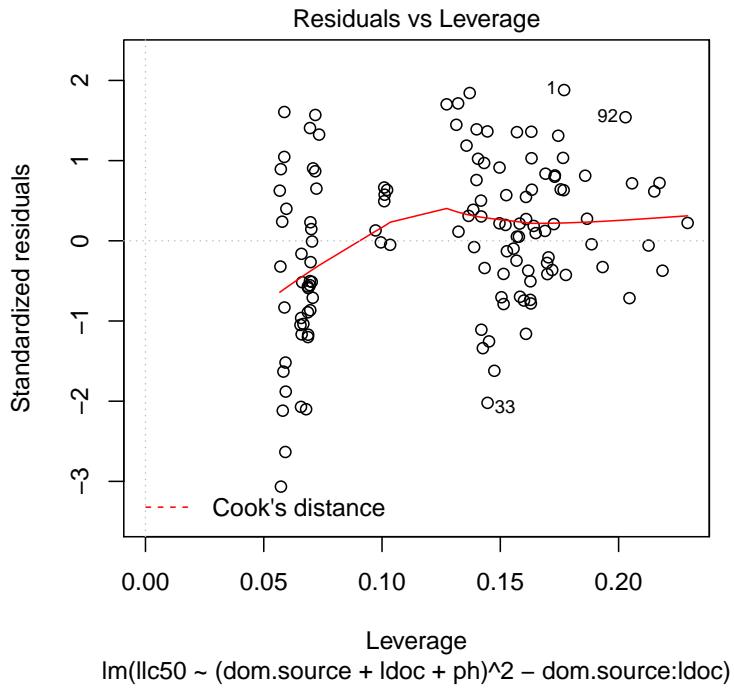
```

Using R's `plot()` function along with some others, it is possible to very clearly display factorial data and model fit. Simply applying the `plot()` function to ANOVA or ANCOVA output will give you some useful diagnostic plots as we have seen with other `lm()` objects.

```
plot(mod2)
```







Here is some more advanced code for plotting these results using base graphics functions. One of the resulting seven plots is shown below the code<sup>137</sup>.

First we need to get the predictions.

```

tox$llc50.pred <- predict(mod2)
tox$lc50.pred <- 10^tox$llc50.pred
tox$n.ph <- factor(tox$n.ph)

source("../functions/logaxis.R")
n.colors <- c("red", "blue", "green")

for(i in levels(tox$dom.source)[1]) {
  sub.1 <- subset(tox, dom.source == i)

  plot(1, 1, type = "n", log = "xy", xlim = c(1, 30), ylim = c(5, 1000),
    xlab = "DOC (mg/L)", ylab = expression("Dissolved Cu LC50"^{~(mu*g/L)}),
    main = paste("DOC", i), las = 1, axes = FALSE)

  logaxis(1)
  logaxis(2)
  box()

  k <- 0
}

```

<sup>137</sup>Remove the [1] in the `for` line to get all of them.

```

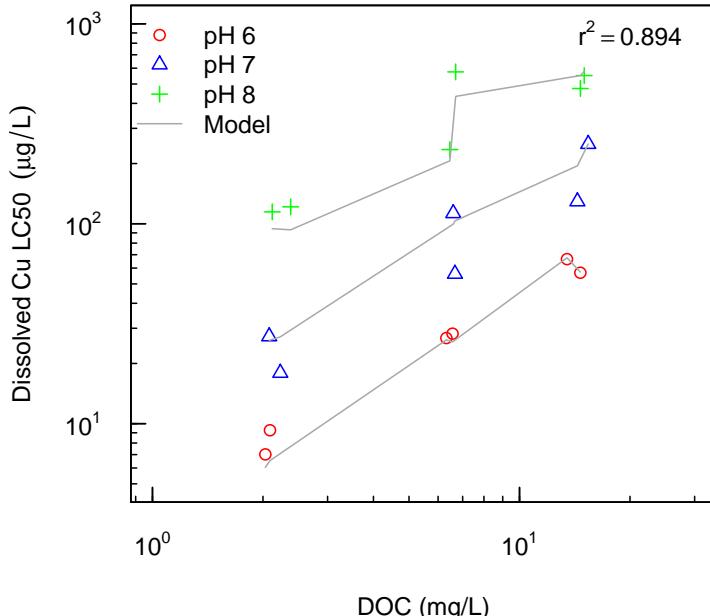
for (j in levels(sub.1$n.ph)) {
  k <- k+1
  sub.2 <- subset(sub.1, n.ph == j)
  sub.2 <- sub.2[order(sub.2$doc), ]
  points(sub.2$doc, sub.2$lc50, pch = k, col = n.colors[k])
  points(sub.2$doc, sub.2$lc50.pred, type = "l", col = "darkgray")
}

if (i == 1) legend("topleft", c("pH 6", "pH 7", "pH 8", "Model"), pch = 1:4,
  col = c(n.colors[1:3], "darkgray"), lty = c(0, 0, 0, 1),
  pt.cex = c(1, 1, 1, 0), bty = "n")

text(20, 900, substitute(r^2 == x, list(x = signif(cor(sub.2$lc50.pred, sub.2$lc50)^2, 3))))
}

```

## DOC 1



And here is some alternate code that relies on functions from the `plyr` and `ggplot2` packages.

```

library(plyr)
library(ggplot2)

tox$n.ph <- factor(tox$n.ph)

summ <- ddply(tox, .(dom.source), summarise, r2 = signif(cor(lc50.pred, lc50)^2, 2))

theme_set(theme_bw())

p <- ggplot(tox, aes(doc, lc50)) + geom_point(aes(colour = n.ph, shape = n.ph)) +

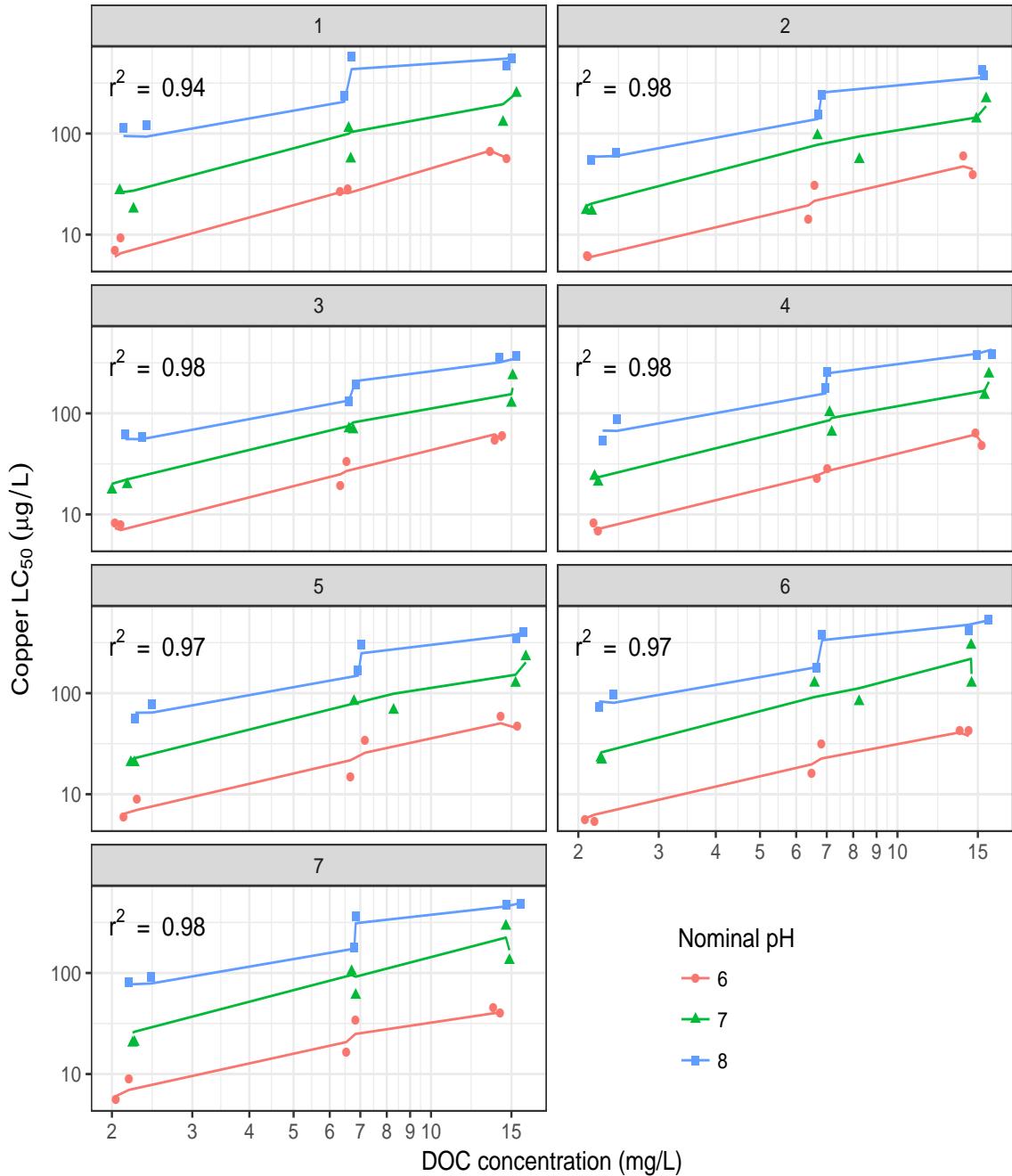
```

```

scale_x_log10(breaks = c(1:10, 15)) + scale_y_log10() +
facet_wrap(~dom.source, ncol = 2) +
geom_line(aes(doc, lc50.pred, colour = n.ph)) +
labs(x = "DOC concentration (mg/L)",
y = expression('Copper LC'[50]~(mu*g/L)),
colour = "Nominal pH", shape = 'Nominal pH', parse = TRUE)

p + geom_text(aes(x = 10^0.4, y = 10^2.5, label = paste("r^2~" = '~~', r2)),
data = summ, parse = TRUE, size = 4) + theme(legend.position = c(0.7, 0.1))

```



## 21 Generalized linear models

### 21.1 Introduction to `glm`

Generalized linear models (GLMs) are a very flexible class of statistical models. The generic form of a GLM is given by

$$g(E(y)) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_m x_m$$

where  $g$  is the link function,  $E(y)$  represents the expected values of the response variable (i.e., the observed values minus the error),  $x_1$  through  $x_m$  are predictors, and  $\beta_1$  through  $\beta_m$  are coefficients. In R, GLM models are available through the `glm` function in the `stats` packages, as well as other functions in add-on packages. There are eight different error distributions available in `glm`, including `gaussian` (normal), `binomial`, and `poisson`, each with a default link function. Arguments for `glm` are:

```
args(glm)

# function (formula, family = gaussian, data, weights, subset,
#         na.action, start = NULL, etastart, mustart, offset, control = list(...),
#         model = TRUE, method = "glm.fit", x = FALSE, y = TRUE, contrasts = NULL,
#         ...)
# NULL
```

The `glm` function can be used for data with non-normal responses—a common example is a binomial response, such as present/absent or dead/alive. In this case, errors are not normally distributed and the relationship between a predictor and response variable is not linear. In this section, I'll describe the use of `glm()` for fitting binomial, Poisson, and, along the way, Gaussian GLMs. There are additional types of GLMs that can be fit with the `glm()` function and there are other functions available in add-on packages that offer even more flexibility—see Faraway [7] and Venables and Ripley [24] for more information.

### 21.2 Binary responses

Let's start with a simple binomial example on copper toxicity to water fleas (*Daphnia magna*).

```
tox <- read.csv("../data/cu_tox_test.csv")
tox

#      cu alive tot
# 1  1.20    20  20
# 2  8.19    20  20
# 3 14.29    18  20
# 4 22.21    11  20
# 5 30.68     4  20
# 6 47.45     2  20
# 7 59.21     0  20
```

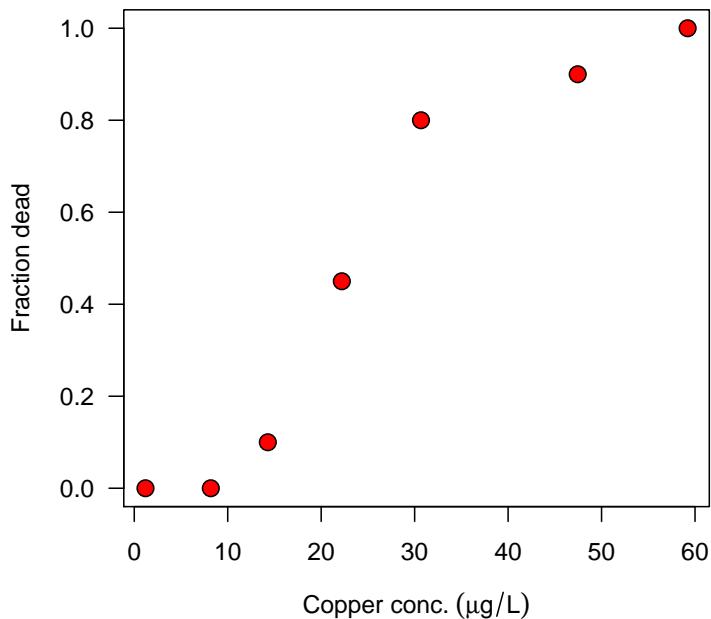
These results are from a single trial with seven copper concentrations. For each concentration, 20 water fleas were placed in water with the given concentration ( $\mu\text{g/L}$ ), and their status (dead or alive) was recorded after some set amount of time. With the data in `tox`, we can use three different forms. Our data are currently grouped—each row has results for several individuals. With the data frame we have, we can either specify the response as a proportion and specify a `weights` argument, which is the total number of organisms, or we can use a *matrix* with the number dead and the number alive as the response variable. Alternatively, ungrouped data, with one outcome (dead or alive) per row, could be used as well. Let's start out with the proportion of individuals dead as the response.

```
tox$dead <- tox$tot-tox$alive
tox$pdead <- tox$dead/tox$tot
tox
```

	cu	alive	tot	dead	pdead
# 1	1.20	20	20	0	0.00
# 2	8.19	20	20	0	0.00
# 3	14.29	18	20	2	0.10
# 4	22.21	11	20	9	0.45
# 5	30.68	4	20	16	0.80
# 6	47.45	2	20	18	0.90
# 7	59.21	0	20	20	1.00

The new `pdead` column is also useful for plotting these data.

```
plot(pdead ~ cu, data = tox, xlab = expression("Copper conc."^(mu*g/L)), ylab = "Fraction dead", pch = 16)
```



There is clearly a strong response to copper in solution. Now for the model.

```
modg <- glm(pdead ~ cu, family = binomial, weights = tot, data = tox)
```

Notice that we didn't specify a link function in the above call. This is because each `family` has a default link function (which can be found in the help file for `family()`), and the "logit" is the default for `binomial`, which is perfectly suitable for these data. So we are carrying out logistic regression in this example.

For the other (matrix) method, we can create a matrix using `cbind()`.

```
resp <- cbind(tox$dead, tox$alive)
modm <- glm(resp ~ cu, family = binomial, data = tox)
summary(modm)

#
# Call:
# glm(formula = resp ~ cu, family = binomial, data = tox)
#
# Deviance Residuals:
#      1       2       3       4       5       6       7 
# -0.7689 -1.4015 -0.3770  0.7625  0.8531 -1.8014  0.3306 
#
# Coefficients:
#             Estimate Std. Error z value Pr(>|z|)    
# (Intercept) -4.41625   0.76028 -5.809 6.29e-09 ***
# cu          0.17425   0.03067  5.681 1.34e-08 ***
# ---        
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1 
#
# (Dispersion parameter for binomial family taken to be 1)
#
# Null deviance: 119.8180  on 6  degrees of freedom
# Residual deviance:  7.3608  on 5  degrees of freedom
# AIC: 22.887
#
# Number of Fisher Scoring iterations: 5
```

So far we've been working with grouped data, where each observation in our data set contains results for multiple individuals (we could also call it aggregated or table data). In general this is the approach most researchers use for organizing binary data when it is an option. Each row in our `tox` data frame can be called a "covariate class". It is often not possible to use this approach when working with continuous predictors. Hypothesis testing is more difficult when this is the case. So, if there is a choice (there may not be with continuous predictors), use grouped data.

To demonstrate the third approach, we need to "ungroup" our data. This probably isn't a very common operation, since you have to have the original (ungrouped) data in order to produce the grouped version, but we only have the latter, so let's use a function called `expand.bin` that I wrote for this purpose.

```
expand.bin <- function(dat, success, total, dropcol, name.success) {
  nr <- nrow(dat)
  x <- dat[, -which(names(dat) %in% c(success, total, dropcol)), drop = FALSE]
```

```

success <- dat[, success]
total <- dat[, total]
fail <- total - success

out <- NULL
for(i in 1:nr) {
  for(j in 1:success[i]) {
    out <- rbind(out, cbind(x[i, ,drop = FALSE], success = 1))
  }
  for(j in 1:fail[i]) {
    out <- rbind(out, cbind(x[i, ,drop = FALSE], success = 0))
  }
}
names(out)[names(out) == 'success'] <- name.success
rownames(out) <- 1:nrow(out)
as.data.frame(out)
}

args(expand.bin)

# function (dat, success, total, dropcol, name.success)
# NULL

toxu <- expand.bin(tox, "dead", "tot", c("alive", "pdead"), name.success = "dead")
dfsumm(toxu)

#
# 146 rows and 2 columns
# 14 unique rows
#          cu     dead
# Class      numeric numeric
# Minimum      1.2      0
# Maximum      59.2      1
# Mean         26      0.473
# Unique (excl. NA)    7      2
# Missing values 0      0
# Sorted       TRUE     FALSE

head(toxu)

#   cu dead
# 1 1.2  1
# 2 1.2  1
# 3 1.2  0
# 4 1.2  0
# 5 1.2  0
# 6 1.2  0

tail(toxu)

```

```

#      cu dead
# 141 59.21    1
# 142 59.21    1
# 143 59.21    1
# 144 59.21    1
# 145 59.21    0
# 146 59.21    0

```

It looks like that worked. A more likely operation than the one we just carried out with `expand.bin()` is the reverse one. The tools we covered in Section 15 can be used for this aggregation step<sup>138</sup>.

And now for the model.

```

modu <- glm(dead ~ cu, family = binomial, data = toxu)
summary(modu)

#
# Call:
# glm(formula = dead ~ cu, family = binomial, data = toxu)
#
# Deviance Residuals:
#      Min       1Q   Median       3Q      Max
# -2.6585  -0.6723  -0.3780   0.4443   2.3128
#
# Coefficients:
#             Estimate Std. Error z value Pr(>|z|)
# (Intercept) -2.72950   0.44328  -6.157 7.39e-10 ***
# cu          0.10528   0.01664   6.328 2.49e-10 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# (Dispersion parameter for binomial family taken to be 1)
#
# Null deviance: 201.96 on 145 degrees of freedom
# Residual deviance: 121.54 on 144 degrees of freedom
# AIC: 125.54
#
# Number of Fisher Scoring iterations: 5

```

Note that while the parameter estimates are identical between the two models,

```

coefficients(modg)

# (Intercept)           cu
# -4.4162494   0.1742492

coefficients(modu)

```

---

<sup>138</sup> Here is one option: `toxg <- ddply(toxu, "cu", summarise, alive = sum(dead == 0), tot = length(dead), dead = sum(dead == 1), pdead = sum(dead == 1)/length(dead))` The `table()` function is an alternative, but it is surprisingly difficult to get the result in the exact format we want. The format returned from `as.data.frame(table(toxu$cu, toxu$dead))` could be useful as well.

```

# (Intercept)          cu
# -2.7294984   0.1052786

all.equal(coefficients(modg), coefficients(modm), coefficients(modu))

# [1] TRUE

```

the deviance estimates are not:

```

modg[c("deviance", "null.deviance")]

# $deviance
# [1] 7.360814
#
# $null.deviance
# [1] 119.818

modm[c("deviance", "null.deviance")]

# $deviance
# [1] 7.360814
#
# $null.deviance
# [1] 119.818

modu[c("deviance", "null.deviance")]

# $deviance
# [1] 121.5448
#
# $null.deviance
# [1] 201.9604

```

Did you catch the warning message R returned when we tried to fit the saturated model to the ungrouped data? The `glm()` function uses a iterative weighted least squares algorithm to fit models. And you'll will rarely see this warning—it is a very effective approach.

Let's discuss hypothesis testing with `glm()`, and the binomial distribution in particular. With generalized linear models, we can think of two types of hypothesis tests: goodness-of-fit tests, and comparisons of models. The first type is used to determine if the model fits the data, and generally cannot be done with linear models (since the dispersion is determined from the data). With a binomial distribution, the model deviance can be used for goodness-of-fit. The null distribution for the deviance is asymptotically  $\chi^2$ .

```

(dev <- deviance(modg))

# [1] 7.360814

```

```
(df <- df.residual(modg))

# [1] 5

1-pchisq(dev, df)

# [1] 0.195159
```

For a model that fits the data well, the  $P$ -value should be high—not the 0.195 we see here. This result suggests that we could do better. We'll work on improving the model fit below.

This test is not accurate for data sets with small group sizes. Here we have 20 individuals in each group, which is not small. In the case of our ungrouped, binary, data set `toxu`, and the resulting model `modu`, the test is not appropriate.

```
# Not accurate!
(dev <- deviance(modu))

# [1] 121.5448

(df <- df.residual(modu))

# [1] 144

1-pchisq(dev, df)

# [1] 0.9129518

# Not accurate!
```

What can we do? In this case, just group the data! In other cases, e.g., working with a continuous predictor or multiple continuous predictors, there are approaches that can be applied for creating groups and testing for goodness-of-fit [1].

Let's move on to hypothesis tests for comparing nested models. As with other R functions for fitting statistical models, we can get a summary of the model that includes Wald-type test results with the `summary` function.

```
summary(modg)

#
# Call:
# glm(formula = pdead ~ cu, family = binomial, data = tox, weights = tot)
#
# Deviance Residuals:
#      1       2       3       4       5       6       7
```

```

# -0.7689 -1.4015 -0.3770  0.7625  0.8531 -1.8014  0.3306
#
# Coefficients:
#             Estimate Std. Error z value Pr(>|z|)
# (Intercept) -4.41625    0.76028 -5.809 6.29e-09 ***
# cu          0.17425    0.03067  5.681 1.34e-08 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# (Dispersion parameter for binomial family taken to be 1)
#
# Null deviance: 119.8180 on 6 degrees of freedom
# Residual deviance: 7.3608 on 5 degrees of freedom
# AIC: 22.887
#
# Number of Fisher Scoring iterations: 5

```

Not surprisingly, `cu` appears to have a highly significant effect on water flea survival. In effect, this test is comparing a model with `cu` to one without it (the null model in this case). The test is based on the assumption that the ratio of the estimate to its standard error follows a  $z$  distribution in the null case. This assumption is not always reasonable<sup>139</sup>, and so the  $P$ -value column should be taken as approximate. This limitation is a universal problem with generalized linear models, and not specific to R [10].

An alternative, which is still imperfect but more reliable than the Wald test, is the difference in deviance test (likelihood ratio test). We can use the `anova()` function to compare the residual deviance of our model to a naive “null” model.

```

anova(modg, test = "Chi")

# Analysis of Deviance Table
#
# Model: binomial, link: logit
#
# Response: pdead
#
# Terms added sequentially (first to last)
#
#
#      Df Deviance Resid. Df Resid. Dev  Pr(>Chi)
# NULL           6     119.818
# cu      1     112.46   5     7.361 < 2.2e-16 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

We get a lower  $P$ -value than with the Wald test, although both indicate a highly significant relationship. This `anova` function can be used to compare any two nested models. For example, suppose we wanted to add an additional term.

---

<sup>139</sup> If inaccurate, it is generally overly conservative, i.e., the  $P$ -value is larger than the true value.

```

modq <- glm(pdead ~ cu + I(cu^2), family = binomial, weights = tot, data = tox)
anova(modg, modq, test = "Chi")

# Analysis of Deviance Table
#
# Model 1: pdead ~ cu
# Model 2: pdead ~ cu + I(cu^2)
#   Resid. Df Resid. Dev Df Deviance Pr(>Chi)
# 1       5    7.3608
# 2       4    3.5063  1   3.8545  0.04961 *
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

Interestingly, it looks like a quadratic terms significantly improves the model fit. Let's take a look at goodness-of-fit again.

```

1-pchisq(deviance(modq), df.residual(modq))

# [1] 0.4769232

```

That is much better! We'll try some different link functions below as well.

We can use all of the extractor functions described above (p. 204) for `lm` objects here with `glm` objects as well.

So, for example, if we wanted to add predicted values to our data frame, we could use `fitted` or `predict`. Note that there are three types of predictions that `predict` will return from a `glm` object.

```

args(predict.glm)

# function (object, newdata = NULL, type = c("link", "response",
#     "terms"), se.fit = FALSE, dispersion = NULL, terms = NULL,
#     na.action = na.pass, ...)
# NULL

```

If we are interested in directly comparing predictions to measurements, we want `type = "response"`.

```

preds <- data.frame(cu = seq(0, 60, length.out = 30))
preds$pdead <- predict(modq, newdata = preds, type = "response")

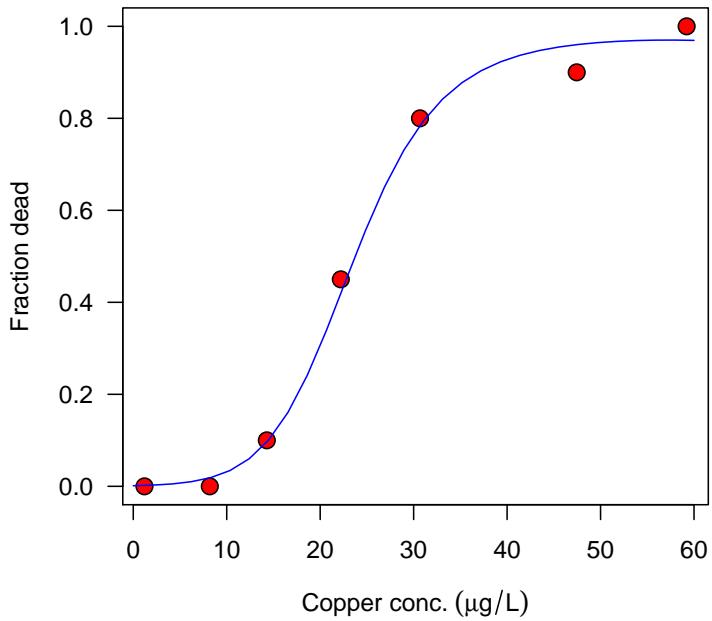
```

We could plot these results, of course.

```

plot(pdead ~ cu, data = tox, xlab = expression("Copper conc."^(mu*g/L)),
      ylab = "Fraction dead", pch = 21, cex = 1.5, bg = "red", las = 1)
lines(pdead ~ cu, data = preds, col = "blue")

```



The `residuals` function also returns a few different types of residuals when applied to a `glm` object.

```
args(residuals.glm)

# function (object, type = c("deviance", "pearson", "working",
#   "response", "partial"), ...)
# NULL
```

For model diagnostics, you probably want the deviance residuals, which are an attempt to partition the total model deviance (3.51 in this case) among all the points<sup>140</sup>.

```
(d <- residuals(modq))

#           1          2          3          4          5
# -0.27671508 -0.86043055  0.05771088  0.27773253  0.17104818
#           6          7
# -1.16236130  1.10840725

sum(d^2)

# [1] 3.506286

deviance(modq)

# [1] 3.506286
```

---

<sup>140</sup> This relationship is analogous to that between residuals and the residual sum of squares from a linear model.

## 21.3 Count data

By simplifying our `satell` count response down to a binary response, we are ignoring some of the information in this data set. Can we just use the count itself as our response?

```
hs <- read.csv("../data/horseshoe.csv")
dfsumm(hs)

#
# 173 rows and 5 columns
# 173 unique rows
#          color   spine   width satell weight
# Class      integer integer numeric integer integer
# Minimum        2       1      21       0     1200
# Maximum        5       3     33.5      15     5200
# Mean           3       3     26.3      2     2300
# Unique (excl. NA)  4       3      66      15      56
# Missing values  0       0      0       0      0
# Sorted        FALSE    FALSE   FALSE   FALSE   FALSE

mod1 <- glm(satell ~ width + color + spine, family = poisson, data = hs)
summary(mod1)

#
# Call:
# glm(formula = satell ~ width + color + spine, family = poisson,
#      data = hs)
#
# Deviance Residuals:
#      Min      1Q      Median      3Q      Max
# -2.9787 -1.9516 -0.5332  0.9819  4.7935
#
# Coefficients:
#             Estimate Std. Error z value Pr(>|z|)
# (Intercept) -2.37370  0.65243 -3.638 0.000275 ***
# width        0.15003  0.02084  7.200 6.01e-13 ***
# color        -0.17415  0.06649 -2.619 0.008817 **
# spine         0.01089  0.05562  0.196 0.844840
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# (Dispersion parameter for poisson family taken to be 1)
#
# Null deviance: 632.79  on 172  degrees of freedom
# Residual deviance: 560.16  on 169  degrees of freedom
# AIC: 923.46
#
# Number of Fisher Scoring iterations: 6
```

Considering goodness-of-fit, we actually don't even need to consult the  $\chi^2$  distribution for this one: a deviance of 560.1629514 with 169 degrees of freedom is terrible.

```

1-pchisq(deviance(mod1), df.residual(mod1))

# [1] 0

```

See?

But let's assess the significance of individual predictors for now.

```

anova(mod1, test = "Chisq")

# Analysis of Deviance Table
#
# Model: poisson, link: log
#
# Response: satell
#
# Terms added sequentially (first to last)
#
#
#      Df Deviance Resid. Df Resid. Dev  Pr(>Chi)
# NULL          172     632.79
# width         1    64.913     171     567.88 7.828e-16 ***
# color         1     7.677     170     560.20  0.005592 **
# spine         1     0.038     169     560.16  0.844698
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

Again, only `width` and `color` seem to be the only significant predictors.

```

mod2 <- update(mod1, ~. - spine)

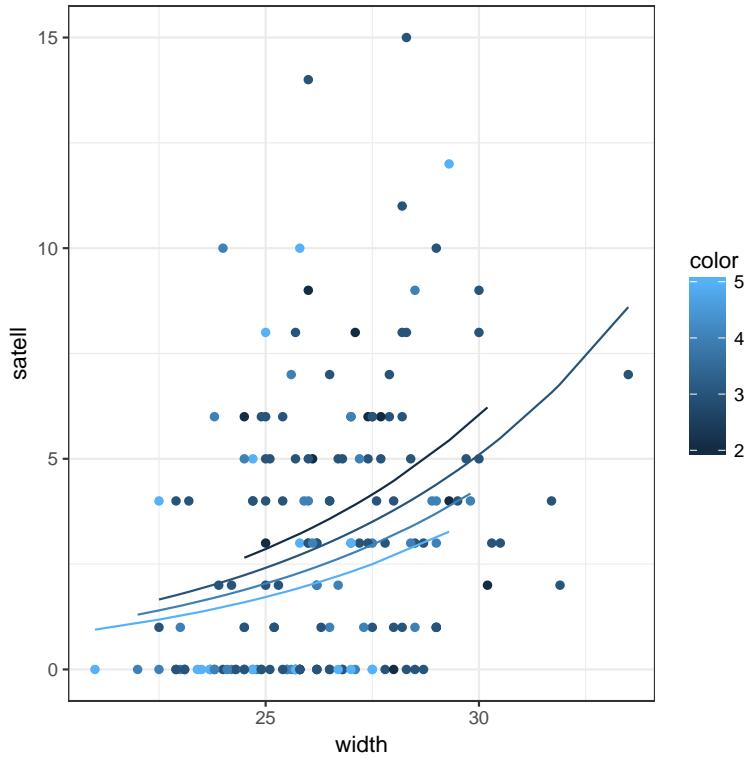
```

Looking at the help file for `poisson`, we see that the default link function is "`log`". This would give us an exponential response of `sat` to `width`. Let's try to visualize it.

```

hs$pred2 <- predict(mod2, type = "response")
library(ggplot2)
ggplot(data = hs) + geom_point(aes(width, satell, colour = color)) +
  geom_line(aes(width, pred2, group = factor(color), colour = color))

```



Let's try an identity link.

```
modi <- glm(satell ~ width + color, family = poisson(link = "identity"), data = hs)

# Warning in log(y/mu):  NaNs produced

# Error:  no valid set of coefficients has been found:  please supply starting values
```

We could provide starting estimates through the `start` argument, but it turns out that doesn't help here. A little trickery to figure out if we can work around this problem. So which observations are causing the problem?

```
mod100 <- glm(satell + 100 ~ width + color, family = poisson(link = "identity"),
                data = hs, start = coef(mod2)+c(100, 0, 0))
which(fitted(mod100)<100)

# 14
# 14
```

Let's drop this point, and refit without the offset.

```
modi <- glm(satell ~ width + color, family = poisson(link = "identity"),
             data = hs[-14, ], start = coef(mod100)+c(-100, 0, 0))
summary(modi)

#
```

```

# Call:
# glm(formula = satell ~ width + color, family = poisson(link = "identity"),
#      data = hs[-14, ], start = coef(mod100) + c(-100, 0, 0))
#
# Deviance Residuals:
#       Min      1Q  Median      3Q     Max
# -2.9052 -1.9362 -0.4275  0.9182  4.6584
#
# Coefficients:
#             Estimate Std. Error z value Pr(>|z|)
# (Intercept) -8.51534   1.80310 -4.723 2.33e-06 ***
# width        0.47901   0.06149  7.790 6.70e-15 ***
# color        -0.33841   0.14965 -2.261  0.0237 *
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# (Dispersion parameter for poisson family taken to be 1)
#
# Null deviance: 626.94 on 171 degrees of freedom
# Residual deviance: 551.93 on 169 degrees of freedom
# AIC: 913.22
#
# Number of Fisher Scoring iterations: 8

anova(mod1, test = "Chisq")

# Warning in log(y/mu):  NaNs produced

# Error: no valid set of coefficients has been found: please supply starting values

anova(mod100, test = "Chisq")

# Analysis of Deviance Table
#
# Model: poisson, link: identity
#
# Response: satell + 100
#
# Terms added sequentially (first to last)
#
#
#          Df Deviance Resid. Df Resid. Dev Pr(>Chi)
# NULL              172      16.384
# width    1  1.91551     171      14.469  0.1664
# color    1  0.18232     170      14.286  0.6694

```

## 22 Random and mixed-effects models

### 22.1 Introduction to the `lme4` package

All the statistical models that we have considered so far include only fixed effects as predictors. Fixed effects are population parameters that estimated using, e.g., `lm`, such as the mean respiration rate of female crabs of species 1. Conversely, a random effect is specific to an individual experimental unit, and it makes sense to estimate parameters that describes its variation among experimental units. Whether or not a predictor should be entered as a fixed or a random effect is sometimes subjective, and in some cases, irrelevant if you are interested in inference about the clearly fixed effect(s). The term mixed-effects models is used for models with both fixed and random effects.

We'll use the `lme4` package to fit mixed-effects models. There is some overlap between the functionality of this package and the `nlme` package. Before we dive into the details, however, it is probably useful to discuss mixed-effects models in R from a broader perspective. The field of mixed-effects models is an active one, new knowledge is currently being generated, and the `lme4` package is at the front edge of research. R contributors (in particular Douglas Bates, Professor of Statistics at the University of Wisconsin and the author of `lme4`) are involved in this research. If you have experience with mixed models using SAS, you will notice that output from `lme4` functions differs from SAS methods, in particular with regard to statistical inference.

The difficulty in statistical inference for mixed effects models is due to two problems: the calculated  $F$  statistic does not necessarily follow the  $F$  distribution, and there is no good way to determine degrees of freedom<sup>141</sup>. Julian Faraway (the author of two good books on statistical models in R) sums up the issue (Faraway 2010): “One approach is to simply provide the approximate solution even though it is known to be poor in some cases. Or one can take the approach that no answer (at least for now) is better than a possibly poor answer, which is the approach currently taken in `lme4`.” Faraway continues, “`lme4` is perhaps the best software generally available for fitting [mixed effects] models, but given the state of the field, there will be scope for significant improvements for some time.”

An older package, which shares an author with the `lme4` package is called `nlme`. It is less efficient at fitting models, and takes a less cautious approach to hypothesis tests. We will focus on the `lmer()` function (in the `lme4` package), which is for linear mixed effect models<sup>142</sup>. Its arguments are given below.

```
library(lme4)
args(lmer)

# function (formula, data = NULL, REML = TRUE, control = lmerControl(),
#        start = NULL, verbose = 0L, subset, weights, na.action, offset,
#        contrasts = NULL, devFunOnly = FALSE, ...)
# NULL
```

The syntax for `lmer` is similar to `lm`, `glm`, and other similar functions. An important addition is the syntax for specifying random effects within the `formula` argument, which I'll discuss below.

<sup>141</sup> You can find more discussion (and demonstration) of these problems in [13] and online. A good place to start online is <https://stat.ethz.ch/pipermail/r-help/2006-May/094765.html>.

<sup>142</sup> The `lme4` package also includes functions for generalized linear mixed models and nonlinear mixed models. See the help file for `lmer()` for a complete list.

## 22.2 Random effects models

Let's use it for a simple example with only a single random effect, using the data in eagles.csv.

```
eagles <- read.csv("../data/eagles.csv")
dfsumm(eagles)

#
# 147 rows and 2 columns
# 126 unique rows
#           site achlor
# Class      factor numeric
# Minimum     Ballville 0.375
# Maximum     Wills Ck  5.45
# Mean        Mercer   1.54
# Unique (excl. NA) 40    92
# Missing values 0    0
# Sorted      FALSE   FALSE
```

The data are on the concentration of a synthetic chemical (alpha-Chlordane) in the blood of eagles collected from 40 or so sites around the Great Lakes in the northeastern US. Let's make a more illuminating summary with ddply.

```
(summ <- ddply(eagles, "site", summarise, mean = mean(achlor), n = length(achlor)))

#
#           site      mean n
# 1       Ballville 2.7250000 4
# 2      Camp Perry 1.3250000 6
# 3 Carroll Twp-Camp Perry 2.3900000 2
# 4 Carroll Twp-Toussaint 2.3400000 1
# 5      Cedar Pt NWR 3.0580000 5
# 6          Cr 306 2.9900000 1
# 7          CR 306 2.2850000 8
# 8 Davis Besse-Toussaint 1.8200000 1
# 9      Ft Seneca 1.3414286 7
# 10         Garlo 0.3750000 3
# 11      Gibsonburg 1.8950000 5
# 12         Gonya 1.8550000 2
# 13 Indian Mill-Sycamore 1.3600000 1
# 14      Killdeer 0.4650000 8
# 15         Kinsman 0.3750000 3
# 16         Knobby 0.7825000 2
# 17     magee Marsh 1.2500000 1
# 18     Magee Marsh 1.8842857 7
# 19         Meander 1.2922222 9
# 20         Mercer 1.0300000 2
# 21 Metzger-Peach Is 1.4900000 1
# 22         Mosquito 0.3750000 3
# 23         Mud Ck 2.0757143 7
# 24 Old Womans Ck 1.1016667 3
# 25      Ottawa NWR 2.1412500 4
# 26      Ottawa SC 1.9200000 1
```

```

# 27          Peach Is 0.9725000 4
# 28      Pickeral Ck 1.2150000 7
# 29          Pymatuning 0.3750000 2
# 30          Rockwell 1.2556250 8
# 31          Rookery 0.9566667 6
# 32          Rossford 2.9900000 2
# 33      Sandusky Airport 1.6850000 2
# 34          Sass 2.7350000 2
# 35          Shandngo 0.3750000 1
# 36          Shendngo 0.4560000 5
# 37  Smithville-Sycamore 1.4450000 2
# 38          Snow Lk 0.9787500 4
# 39  Tri Valley-Wills Ck 3.8200000 1
# 40          Wills Ck 3.6225000 4

```

It looks like we have some problems with inconsistent capitalization, which we can fix. And there happens to be a detection limit issue problem in this dataset<sup>143</sup>, so we'll remove all observations below the detection limit of 0.75 ng/g. And, let's (arbitrarily) only look at those sites with 4 or more observations. It will take a few steps to fix all of these problems.

```

eagles$site <- tolower(eagles$site)
eag <- subset(eagles, achlor>0.75)
summ <- ddply(eag, "site", summarise, mean = mean(achlor), n = length(achlor))
eag <- subset(eag, achlor>0.75 & site %in% summ$site[summ$n>3])
eag$site <- factor(eag$site)

```

We are left with 13 locations, and an unbalanced design:

```

ddply(eag, .variables = "site", .fun = summarise, min = min(achlor), mean = mean(achlor), n = length(achlor))

#           site  min     mean   n
# 1    ballville 1.69 2.725000 4
# 2    camp perry 1.28 1.800000 4
# 3  cedar pt wr 2.22 3.058000 5
# 4          cr 306 1.76 2.363333 9
# 5    ft seneca 1.51 1.728000 5
# 6  gibsonburg 1.81 2.275000 4
# 7  magee marsh 1.21 1.805000 8
# 8      meander 0.87 1.554286 7
# 9      mud ck 1.61 2.075714 7
# 10  pickeral ck 1.10 1.355000 6
# 11  rockwell 0.78 1.381429 7
# 12      rookery 0.89 1.247500 4
# 13      wills ck 1.73 3.622500 4

```

Let's fit construct a random effects model with `lmer`.

```

library(lme4)
mod <- lmer(achlor ~ 1|site, data = eag)
summary(mod)

```

<sup>143</sup> Although you can't tell from just looking at the data.

```

# Linear mixed model fit by REML ['lmerMod']
# Formula: achlor ~ 1 | site
#   Data: eag
#
# REML criterion at convergence: 173.6
#
# Scaled residuals:
#     Min      1Q  Median      3Q     Max
# -2.3380 -0.6100 -0.2427  0.6092  3.2486
#
# Random effects:
# Groups   Name        Variance Std.Dev.
# site     (Intercept) 0.4032   0.6350
# Residual           0.4434   0.6659
# Number of obs: 74, groups: site, 13
#
# Fixed effects:
#             Estimate Std. Error t value
# (Intercept)  2.0657    0.1935 10.68

```

The syntax for random terms uses a vertical bar (pipe symbol) to separate the grouping variable (the right side of `|`) from the fixed term that varies among the groups (to the left of `|`). In this model, the `1|site` indicates that an intercept term varies among the levels of `site`. In fitting this model, we are not interested in whether concentrations differ among sites but rather how they differ. And, how does variability within sites compare to variability across sites? We can use `lm` to fit a similar model, by treating `site` as a fixed effect.

```

lmod <- lm(achlor ~ site, data = eag)
anova(lmod)

# Analysis of Variance Table
#
# Response: achlor
#              Df Sum Sq Mean Sq F value    Pr(>F)
# site          12 29.393 2.44940  5.5501 2.642e-06 ***
# Residuals    61 26.921 0.44133
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

Residual variance is given in the output, and is similar to the result from `lmer`, but it is difficult to calculate the variance of the site effect for these unbalanced data.

There are some extractor functions for `mer` objects like `mod`. Useful ones include `fixef()` and `raneff()`, for returning estimates of fixed and random effects, respectively.

```
fixef(mod)
```

```

# (Intercept)
#       2.065657

```

```
raneff(mod)
```

```

# $site
#           (Intercept)
# ballville      0.517174376
# camp perry    -0.208375368
# cedar pt nbr   0.813451892
# cr 306        0.265267320
# ft seneca     -0.276786853
# gibsonburg    0.164204230
# magee marsh   -0.229159367
# meander       -0.441948459
# mud ck         0.008692112
# pickeral ck   -0.600590441
# rockwell      -0.591338895
# rookery       -0.641744270
# wills ck       1.221153723

```

But, unfortunately, the package does not include the large suite of functions that make up `nlme`. Keeping in mind the limitations discussed above, the `nlme` package can also be used. Because these two packages do have functions that share names, we are going to be keep only one in our workspace at a time.

```

detach("package:lme4")
library(nlme)
mod2 <- lme(achlor ~ 1 ,data = eag, random= ~ 1|site)
summary(mod2)

# Linear mixed-effects model fit by REML
# Data: eag
#       AIC      BIC      logLik
#  179.5509 186.4223 -86.77546
#
# Random effects:
#   Formula: ~1 | site
#             (Intercept)  Residual
# StdDev:    0.6350133 0.6658809
#
# Fixed effects: achlor ~ 1
#                  Value Std.Error DF  t-value p-value
# (Intercept) 2.065657  0.193502 61 10.67512      0
#
# Standardized Within-Group Residuals:
#       Min        Q1        Med        Q3        Max
# -2.3379716 -0.6099625 -0.2426769  0.6092068  3.2486134
#
# Number of Observations: 74
# Number of Groups: 13

fixef(mod2)

# (Intercept)
# 2.065657

```

```

ranef(mod2)

#           (Intercept)
# ballville    0.517174378
# camp perry   -0.208375369
# cedar pt nwr  0.813451894
# cr 306      0.265267320
# ft seneca    -0.276786854
# gibsonburg   0.164204230
# magee marsh   -0.229159367
# meander      -0.441948460
# mud ck       0.008692112
# pickeral ck   -0.600590442
# rockwell     -0.591338896
# rookery      -0.641744272
# wills ck      1.221153727

```

The results seem identical, which is reassuring. We can look at uncertainty in our estimates with the `intervals()` function.

```

intervals(mod2)

# Approximate 95% confidence intervals
#
# Fixed effects:
#           lower     est.     upper
# (Intercept) 1.678726 2.065657 2.452588
# attr(),"label")
# [1] "Fixed effects:"
#
# Random Effects:
# Level: site
#           lower     est.     upper
# sd((Intercept)) 0.3864044 0.6350133 1.043575
#
# Within-group standard error:
#           lower     est.     upper
# 0.5571441 0.6658809 0.7958396

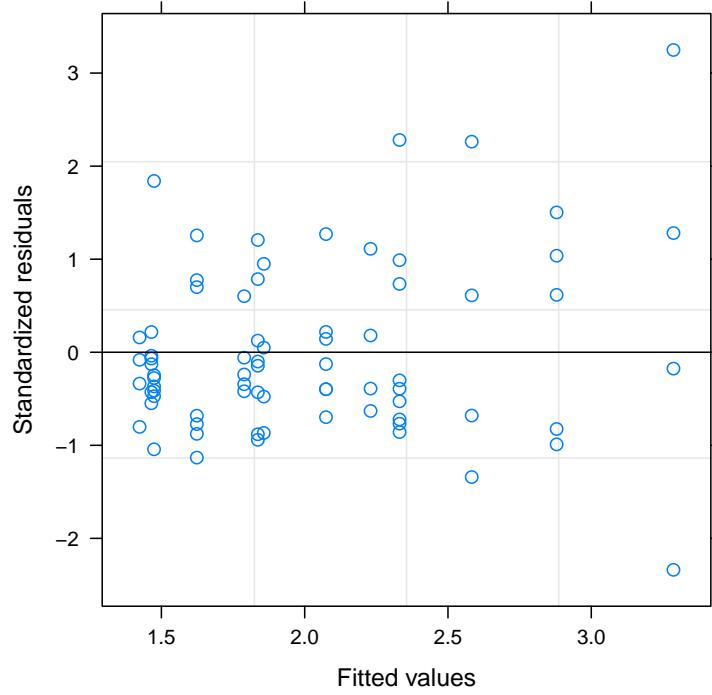
```

And, the `nlme` package includes some very useful methods for producing diagnostic plots. With this simple model, we'll get a simple, but still useful plot.

```

plot(mod2)

```



For this model, we should be concerned about constant variance within groups. The plot doesn't suggest there is a problem. We'll discuss more on model diagnostics below.

## 22.3 Mixed effects models

Mixed-effects models are used to analyze data that has some kind of grouping structure. Examples include repeated-measures, longitudinal (long-term repeated measures), blocked, and multilevel designs. We will look at a few of these designs in this section.

The data in `willow.csv` are on short-rotation willow shrub growth in response to fertilization.

```
willow <- read.csv("../data/willow.csv")
dfsumm(willow)

#
# 144 rows and 7 columns
# 144 unique rows
#
#           field    plot     trt      yr1      yr   yield
# Class      factor integer factor integer integer numeric
# Minimum     Old        1 Control   2004      0    1.68
# Maximum     Young      24 Urea     2007      3    40.9
# Mean        Young      12 Sludge   2005      1    21.5
# Unique (excl. NA) 2        24       6      4      4    141
# Missing values 0        0        0      0      0    0
# Sorted      FALSE     FALSE    FALSE    FALSE    FALSE FALSE
#
#           height
# Class      numeric
# Minimum     1.53
# Maximum     6.75
# Mean        4.11
# Unique (excl. NA) 129
# Missing values 0
# Sorted      FALSE
```

The two response variables are `yield` and `height`, and `year` is the year of measurement. Different fertilizer treatments were applied to plots within two fields. The experimental unit was a plot, and we have measurements on each plot over multiple years. So this is a repeated measures design. Let's focus on only one field value, `Young`, which contained young plants.

```
w <- subset(willow, field == "Young")
```

And let's make a couple of changes to the new data frame.

```
w$field <- factor(w$field)
w$plot <- factor(w$plot)
w$yr <- w$yr - min(w$yr) - 1
```

The data frame has six different treatments (given in the `trt` variable—`Control` is the control), and now 72 observations.

```
dfsumm(w)
```

```
#
```

```

# 72 rows and 7 columns
# 72 unique rows
#           field   plot     trt    yr1      yr    yield
# Class       factor  factor  factor integer numeric numeric
# Minimum      Young     1 Control  2005     -1    1.68
# Maximum      Young    24   Urea   2007      1    37.8
# Mean         Young    12 Sludge  2006      0    16.7
# Unique (excl. NA) 1    24      6      3      3    70
# Missing values 0    0      0      0      0      0
# Sorted        TRUE FALSE FALSE  TRUE  TRUE FALSE
#           height
# Class        numeric
# Minimum      1.53
# Maximum      5.87
# Mean         3.52
# Unique (excl. NA) 69
# Missing values 0
# Sorted        FALSE

levels(w$trt)

# [1] "Control"      "ManureH"       "ManureL"       "Sludge"
# [5] "SludgeManure" "Urea"

```

Let's see how many replicates are present.

```

table(w$trt, w$year)

# Error in table(w$trt, w$year): all arguments must have the same length

```

So there are four replicates for each `trt:year` combination in a balanced design. The experimental unit is the plot, and we should have 24 of them ( $4 \cdot 6 = 24$ ). This is in agreement with the summary above.

Let's take a look at the data, to get an idea of what we expect and what type of a model we should fit.

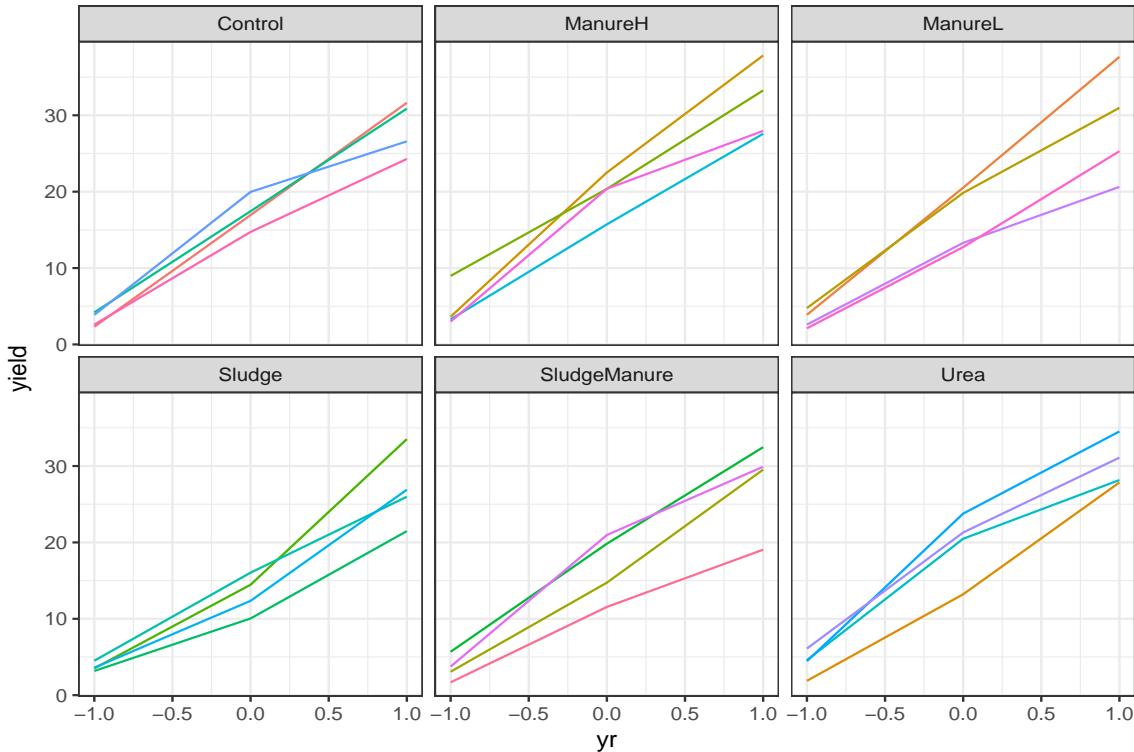
```

library(ggplot2)
head(w)

#   field plot      trt  yr1 yr yield height
# 1 Young   1 Control 2005 -1  2.32  1.75
# 2 Young   2 ManureL 2005 -1  3.88  2.00
# 3 Young   3   Urea  2005 -1  1.88  1.58
# 4 Young   4 ManureH 2005 -1  3.63  1.89
# 5 Young   5 ManureL 2005 -1  4.76  2.04
# 6 Young   6 SludgeManure 2005 -1  3.07  1.87

```

```
ggplot(data = w)+geom_line(aes(yr, yield, colour = plot)) + facet_wrap(facets= ~trt, ncol = 3, nrow =
```



We are interested in seeing if the fertilizer treatments had an effect on overall biomass production. These plots suggest that a linear trend over time may be sufficient, so let's start there.

```
detach("package:nlme")
library(lme4)
mod1 <- lmer(yield ~ trt + yr + (1|plot), data = w, REML = FALSE)
```

The random effect is specified with `(1|plot)` and we used the maximum likelihood method for fitting the model (the default, restricted maximum likelihood, won't work for the test we'll use below). Notice that the random effect is enclosed in parentheses in the formula—this is essential, depending on the way the formula is specified, and so it is a good idea to always do it. Let's take a look at the model results.

```
mod1

# Linear mixed model fit by maximum likelihood  ['lmerMod']
# Formula: yield ~ trt + yr + (1 | plot)
#   Data: w
#       AIC      BIC    logLik  deviance df.resid
#  386.3166 406.8066 -184.1583  368.3166      63
# Random effects:
# Groups   Name        Std.Dev.
# plot     (Intercept) 2.169
# Residual           2.584
# Number of obs: 72, groups: plot, 24
# Fixed Effects:
# (Intercept)      trtManureH      trtManureL      trtSludge
#             16.288          2.418         -0.095        -1.658
```

```
# trtSludgeManure      trtUrea      yr
#           -0.270       1.832     12.582
```

As with `lm()`, treatment contrasts are used by default<sup>144</sup>, and the `Control` group was taken as the “base” group, which is perfect. We can see that the random effect `plot` is assigned a substantial part of the overall variance.

Note that output from `summary` does not include  $P$ -values. But, you can get an idea of the approximate significance of each term by looking at the reported  $t$ -value. Can’t remember how big a big  $t$ -value is? Let’s make a  $t$  table.

```
ttab <- outer(c(3, 10, Inf), c(0.75, 0.90, 0.95, 0.975), function(df, p) round(qt(p = p, df = df), 2))
dimnames(ttab) <- list(df = c(3, 10, Inf), `alpha(1)` = c(0.50, 0.20, 0.10, 0.05))
ttab

#      alpha(1)
# df    0.5 0.2 0.1 0.05
#   3    0.76 1.64 2.35 3.18
#  10    0.70 1.37 1.81 2.23
# Inf   0.67 1.28 1.64 1.96
```

The hypothesis test for the intercept term is irrelevant, and clearly there is an effect of year (these are growing plants, after all). But apart from these terms, nothing much is going on—none of the treatments seemed to be very different from each other.

As with the ANOVA examples we ran using `lm` in Section 20, output from `summary` shows results for our dummy variables, and not an overall test of the `trt` effect. The output does suggest that none of the individual treatments seem to differ much from the control (`trt == C`).

But how do we actually carry out some hypothesis tests? There are a few available approaches. The easiest one is to not use `lmer()`, but use `aov()` with an `Error()` term. This test is appropriate only for balanced data, which we have here.

```
rma.mod <- aov(yield ~ trt + yr + Error(plot), data = w)
summary(rma.mod)

#
# Error: plot
#          Df Sum Sq Mean Sq F value Pr(>F)
# trt      5 134.5   26.89    0.97  0.462
# Residuals 18 499.1   27.73
#
# Error: Within
#          Df Sum Sq Mean Sq F value Pr(>F)
# yr       1  7598    7598   1114 <2e-16 ***
# Residuals 47   321        7
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

coef(rma.mod)
```

---

<sup>144</sup> We could check with `model.matrix()`.

```

# (Intercept) :
# (Intercept)
#   16.65875
#
# plot :
#      trtManureH      trtManureL      trtSludge trtSludgeManure
#      2.418333     -0.095000     -1.657500      -0.270000
#      trtUrea
#      1.831667
#
# Within :
#      yr
# 12.58167

```

We would conclude that the `trt` treatments have no effect on `biomass` yield. But the `lmer()` function can do things that `aov()` cannot, so we can't rely on `aov` for all cases. An alternative to test for an overall treatment effect is the likelihood ratio test. To do this, the model must be fit using the maximum likelihood method—the default method is *restricted* maximum likelihood, which is not compatible with the likelihood ratio test. First, let's fit a null model, without the fixed effect.

```
nmod <- lmer(yield ~ yr + (1|plot), data = w, REML = FALSE)
```

We can carry out a likelihood ratio test with the `anova()` function, but the result is sometimes inaccurate, since the  $\chi^2$  distribution is not always a good approximation. Here is the test:

```

anova(mod1, nmod)

# Data: w
# Models:
# nmod: yield ~ yr + (1 | plot)
# mod1: yield ~ trt + yr + (1 | plot)
#       Df    AIC    BIC  logLik deviance Chisq Chi Df Pr(>Chisq)
# nmod  4 382.04 391.15 -187.02    374.04
# mod1  9 386.32 406.81 -184.16    368.32 5.7252      5     0.3339

```

This  $P$ -value is smaller than the result from `aov`, suggesting that it is not accurate. As described in Faraway (2006: 160), a parametric bootstrap method is a better option. Let's calculate the likelihood ratio test value that we want to assess (or read it from the `Chisq` column in the output from `anova()`), and set up a simulation to carry out this approach.

```
lrstat <- as.numeric(2*(logLik(mod1) - logLik(nmod)))
```

To run a simulation, we can use the `simulate()` function, which will generate simulated data based on the distribution from a fitted model. In the loop below, we simulate data from the null model, and see how frequently we encounter a likelihood ratio test value as large as the one we have between our full model `mod1` and the null model `nmod`.

```

lr.sim <- numeric(1000)
system.time(
  for(i in 1:1000) {

```

```

y <- simulate(nmod) [, 1]
nmod.tmp <- lmer(y ~ yr + (1|plot), data = w, REML = FALSE)
amod.tmp <- lmer(y ~ trt + yr + (1|plot), data = w, REML = FALSE)
lr.sim[i] <- 2*as.numeric(logLik(amod.tmp) - logLik(nmod.tmp))
}
)

# Warning in optwrap(optimizer, devfun, getStart(start, rho$lower, rho$pp), : convergence
# code 3 from bobyqa: bobyqa -- a trust region step failed to reduce q

# Warning in optwrap(optimizer, devfun, getStart(start, rho$lower, rho$pp), : convergence
# code 3 from bobyqa: bobyqa -- a trust region step failed to reduce q

#   user  system elapsed
# 40.167  0.037 40.214

mean(lr.sim > lrstat)

# [1] 0.456

```

It looks like the `anova`  $P$ -value is indeed too low.

Again, the fertilizer treatments did not appear to have any effect on the yield of willow—we have no reason to reject the null hypothesis. Before we consider model diagnostics, let’s think about other plausible models. We’ve tested the effect of fertilizer treatment on overall yields (and did not see an effect), but what if the treatments had differing effects on the change in biomass over time? We can include this effect in the model as an interaction term.

```
mod2 <- lmer(yield ~ (yr + trt)^2 + (1|plot), data = w)
```

And let’s test the significant of the interaction term with a comparison to `mod1`.

```

anova(mod1, mod2)

# refitting model(s) with ML (instead of REML)

# Data: w
# Models:
# mod1: yield ~ trt + yr + (1 | plot)
# mod2: yield ~ (yr + trt)^2 + (1 | plot)
#       Df     AIC     BIC  logLik deviance Chisq Chi Df Pr(>Chisq)
# mod1  9 386.32 406.81 -184.16    368.32
# mod2 14 393.65 425.52 -182.82    365.65 2.6708      5     0.7506

```

We could follow up with a bootstrap analysis, but it hardly seems worth the effort—there is no evidence of an effect here. But we can quickly use `aov()`.

```

summary(aov(yield ~ (trt + yr)^2 + Error(plot/yr), data = w))

#
# Error: plot
#           Df Sum Sq Mean Sq F value Pr(>F)
# trt        5 134.5   26.89   0.97  0.462
# Residuals 18 499.1   27.73
#
# Error: plot:yr
#           Df Sum Sq Mean Sq F value Pr(>F)
# yr         1 7598    7598 689.176 8.41e-16 ***
# trt:yr     5    17      3   0.315   0.898
# Residuals 18    198     11
# ---
# Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Error: Within
#           Df Sum Sq Mean Sq F value Pr(>F)
# Residuals 24 104.8   4.367

```

So there is no evidence that the `trt` treatments changed overall biomass production, or the change in yield over time.

But maybe

```

mod3 <- lmer(yield ~ (yr + trt)^2 + (yr|plot), data = w)
mod3

# Linear mixed model fit by REML ['lmerMod']
# Formula: yield ~ (yr + trt)^2 + (yr | plot)
# Data: w
# REML criterion at convergence: 319.1606
# Random effects:
# Groups   Name        Std.Dev. Corr
# plot     (Intercept) 2.826
#          yr          1.946   1.00
# Residual            1.978
# Number of obs: 72, groups: plot, 24
# Fixed Effects:
# (Intercept)             yr          trtManureH
# 16.28750       12.55375      2.41833
# trtManureL      trtSludge    trtSludgeManure
# -0.09500       -1.65750      -0.27000
# trtUrea        yr:trtManureH  yr:trtManureL
# 1.83167        0.91625      0.09875
# yr:trtSludge   yr:trtSludgeManure  yr:trtUrea
# -0.91750       -0.45500      0.52500

```

This gives us a substantial drop in residual variance.

```

anova(mod1, mod3)

# refitting model(s) with ML (instead of REML)

# Data: w
# Models:
# mod1: yield ~ trt + yr + (1 | plot)
# mod3: yield ~ (yr + trt)^2 + (yr | plot)
#       Df   AIC   BIC logLik deviance Chisq Chi Df Pr(>Chisq)
# mod1  9 386.32 406.81 -184.16    368.32
# mod3 16 374.11 410.53 -171.05    342.11 26.21      7  0.0004619 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

And there is some evidence that the response to `yr` does indeed depend on `plot`.

Do you see how `lmer()` allows us to analyze data that would be difficult to analyze with `lm()`? It may be worth reviewing why `lm()` is more difficult for grouped data.

```

# This model violates the independence assumption!
mod2lm <- lm(yield ~ (yr + trt)^2, data = w)
# This model violates the independence assumption!

```

Let's take a look at the overall tests.

```

# This model violates the independence assumption!
anova(mod2lm)

# Analysis of Variance Table
#
# Response: yield
#             Df Sum Sq Mean Sq F value Pr(>F)
# yr          1 7598.3 7598.3 568.1785 < 2e-16 ***
# trt         5  134.5   26.9   2.0110 0.08992 .
# yr:trt      5   17.4    3.5   0.2595 0.93333
# Residuals  60  802.4   13.4
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

Why do we now see a  $P$ -value that is so much lower? We've ignored the grouping component of our data set, so we've overestimated the SS for `trt` and overestimated the model degrees of freedom as well. We can see both of these problems by comparing the above output to the output from `aov()`, above.

We should be able to get close to the `lmer()` results by a response feature analysis. First, let's fit a model to each individual plot. The `lmList` function in the `lme4` can be used for this task, but there doesn't seem to be an easy way to include a `trt` column in the output.

```

lms <- lmList(yield ~ yr|plot, data = w)
lms

# Call:
#   Model: yield ~ yr | plot
#   Data: w
#
# Coefficients:
#   (Intercept)      yr
# 1       16.96667 14.660
# 2       20.68333 16.885
# 3       14.31333 12.985
# 4       21.32333 17.100
# 5       18.52667 13.110
# 6       15.77667 13.230
# 7       20.86333 12.145
# 8       17.16333 15.005
# 9       19.32667 13.395
# 10      11.56333  9.155
# 11      17.50333 13.345
# 12      15.51000 10.735
# 13      17.74667 11.790
# 14      15.52667 12.150
# 15      14.28333 11.650
# 16      20.91667 15.040
# 17      16.81000 11.360
# 18      19.50000 12.500
# 19      12.17333  9.020
# 20      18.20667 13.085
# 21      17.11000 12.485
# 22      13.38667 11.595
# 23      13.87000 10.850
# 24      10.76000  8.685
#
# Degrees of freedom: 72 total; 24 residual
# Residual standard error: 2.089769

```

So we can use `ddply()` instead.

```

lms <- ddply(w, c("plot", "trt"), function(x) coef(lm(yield ~ yr, data = x)))
lms

#   plot      trt (Intercept)      yr
# 1     1 Control    16.96667 14.660
# 2     2 ManureL    20.68333 16.885
# 3     3 Urea       14.31333 12.985
# 4     4 ManureH    21.32333 17.100
# 5     5 ManureL    18.52667 13.110
# 6     6 SludgeManure 15.77667 13.230
# 7     7 ManureH    20.86333 12.145
# 8     8 Sludge     17.16333 15.005
# 9     9 SludgeManure 19.32667 13.395

```

```

# 10   10     Sludge    11.56333 9.155
# 11   11     Control   17.50333 13.345
# 12   12     Sludge    15.51000 10.735
# 13   13     Urea      17.74667 11.790
# 14   14     ManureH   15.52667 12.150
# 15   15     Sludge    14.28333 11.650
# 16   16     Urea      20.91667 15.040
# 17   17     Control   16.81000 11.360
# 18   18     Urea      19.50000 12.500
# 19   19     ManureL   12.17333 9.020
# 20   20 SludgeManure 18.20667 13.085
# 21   21     ManureH   17.11000 12.485
# 22   22     ManureL   13.38667 11.595
# 23   23     Control   13.87000 10.850
# 24   24 SludgeManure 10.76000 8.685

```

```

int.rfa <- lm(`(Intercept)` ~ trt, data = lms)
anova(int.rfa)

```

```

# Analysis of Variance Table
#
# Response: (Intercept)
#           Df Sum Sq Mean Sq F value Pr(>F)
# trt        5  44.822  8.9645  0.9699 0.4622
# Residuals 18 166.374  9.2430

```

```

yr.rfa <- lm(yr ~ trt, data = lms)
anova(yr.rfa)

```

```

# Analysis of Variance Table
#
# Response: yr
#           Df Sum Sq Mean Sq F value Pr(>F)
# trt        5  8.676  1.7352  0.3148 0.8976
# Residuals 18 99.227  5.5126

```

The results of these hypothesis tests are close to the results we saw above using `aov()` and likelihood ratio tests.

The output from `lmer` has class `mer`. Some of the generic statistical model functions that we've used above do not have methods for the `mer` class (`plot()` is one example). Objects with this class are S4 objects,

```
isS4(mod2)
```

```
# [1] TRUE
```

```
isS4(mod2lm)
```

```
# [1] FALSE
```

This is a bit confusing, but an S4 object can have either S3 or S4 methods associate with it. To find them, we can look at the help file for ‘`class-mer`’, and we can use `methods()` and `showMethods()` to look for particular functions that might work.

```
showMethods(class = "mer")  
  
#  
# Function "asJSON":  
# <not an S4 generic function>  
#  
# Function "complete":  
# <not an S4 generic function>  
#  
# Function "coords":  
# <not an S4 generic function>  
#  
# Function ".DollarNames":  
# <not an S4 generic function>  
#  
# Function "formals<-":  
# <not an S4 generic function>  
#  
# Function "functions":  
# <not an S4 generic function>  
#  
# Function "plot":  
# <not an S4 generic function>  
#  
# Function "prompt":  
# <not an S4 generic function>  
#  
# Function "sumary":  
# <not an S4 generic function>  
  
methods(class = "mer")  
  
# no methods found
```

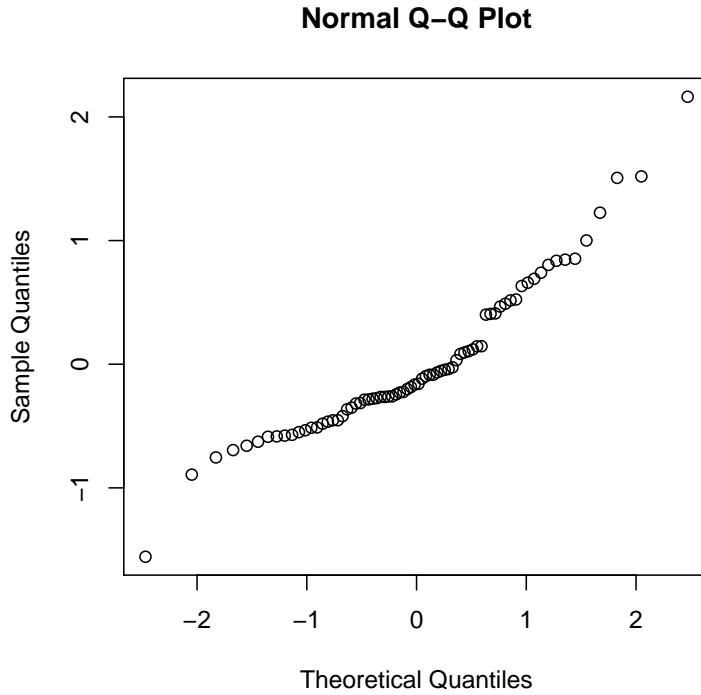
We can extract the fixed and random effects with `fixef()` and `ranef()`, respectively. And, `coef()`, `fitted()`, and `resid()` will return model coefficients, fitted values, and residuals, respectively. Note that there is no `predict()` method for `mer`.

Let’s work on model diagnostics. As described by [13, p. 174], we need to think about two assumptions related to error distribution:

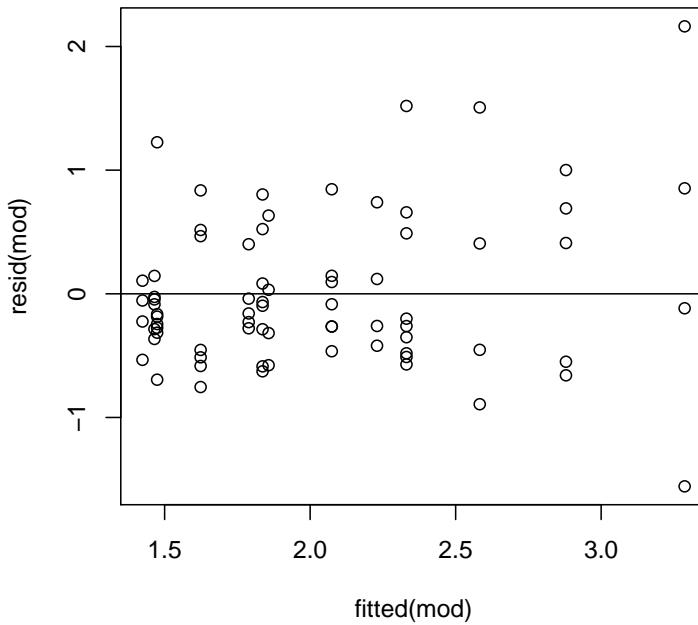
1. Within-group errors are normally distributed, with a mean of zero and a constant variance across all groups, and they are independent of random effects
2. Random effects are normally distributed and independent for different groups, with a mean of zero

The `n1me` package has some really nice plotting methods for assessing these assumptions—unfortunately, the `lme4` package is lacking in this area. But we can hobble something together.

```
qqnorm(resid(mod))
```



```
plot(fitted(mod), resid(mod))
abline(0, 0)
```



There don't seem to be any problems here. But let's repeat our analysis with `nlme` functions.

```
detach("package:lme4")
library(nlme)

mod2b <- lme(yield ~ (yr + trt)^2, random= ~ 1|plot, data = w)
mod3b <- lme(yield ~ (yr + trt)^2, random= ~ yr|plot, data = w)

# Error in lme.formula(yield ~ (yr + trt)^2, random = ~yr | plot, data = w): nlminb
problem, convergence error code = 1
# message = iteration limit reached without convergence (10)
```

Well, there is a very significant difference between the two packages. The summaries for both versions of `mod2` should look similar, but note the *P*-values present in this new summary.

```
summary(mod2b)

# Linear mixed-effects model fit by REML
# Data: w
#      AIC      BIC    logLik
# 368.4947 397.8156 -170.2474
#
# Random effects:
# Formula: ~1 | plot
#             (Intercept) Residual
# StdDev:     2.6146 2.687118
```

```

#
# Fixed effects: yield ~ (yr + trt)^2
#                               Value Std.Error DF   t-value p-value
# (Intercept)           16.287500 1.5201152 42 10.714649 0.0000
# yr                  12.553750 0.9500396 42 13.213922 0.0000
# trtManureH          2.418333 2.1497675 18  1.124928 0.2754
# trtManureL          -0.095000 2.1497675 18 -0.044191 0.9652
# trtSludge            -1.657500 2.1497675 18 -0.771014 0.4507
# trtSludgeManure     -0.270000 2.1497675 18 -0.125595 0.9014
# trtUrea              1.831667 2.1497675 18  0.852030 0.4054
# yr:trtManureH       0.916250 1.3435589 42  0.681957 0.4990
# yr:trtManureL       0.098750 1.3435589 42  0.073499 0.9418
# yr:trtSludge        -0.917500 1.3435589 42 -0.682888 0.4984
# yr:trtSludgeManure -0.455000 1.3435589 42 -0.338653 0.7366
# yr:trtUrea           0.525000 1.3435589 42  0.390753 0.6980
# Correlation:
#                               (Intr) yr      trtMnH trtMnL trtSld trtSlM trtUre
# yr                   0.000
# trtManureH         -0.707  0.000
# trtManureL         -0.707  0.000  0.500
# trtSludge           -0.707  0.000  0.500  0.500
# trtSludgeManure    -0.707  0.000  0.500  0.500  0.500
# trtUrea             -0.707  0.000  0.500  0.500  0.500  0.500
# yr:trtManureH      0.000 -0.707  0.000  0.000  0.000  0.000  0.000
# yr:trtManureL      0.000 -0.707  0.000  0.000  0.000  0.000  0.000
# yr:trtSludge        0.000 -0.707  0.000  0.000  0.000  0.000  0.000
# yr:trtSludgeManure 0.000 -0.707  0.000  0.000  0.000  0.000  0.000
# yr:trtUrea          0.000 -0.707  0.000  0.000  0.000  0.000  0.000
#                         yr:tMH yr:tML yr:trS yr:tSM
# yr
# trtManureH
# trtManureL
# trtSludge
# trtSludgeManure
# trtUrea
# yr:trtManureH
# yr:trtManureL      0.500
# yr:trtSludge        0.500  0.500
# yr:trtSludgeManure 0.500  0.500  0.500
# yr:trtUrea          0.500  0.500  0.500  0.500
#
# Standardized Within-Group Residuals:
#           Min        Q1        Med        Q3        Max
# -1.95094532 -0.51837238  0.01801092  0.52909672  2.04069011
#
# Number of Observations: 72
# Number of Groups: 24

```

Let's continue with other types of mixed-effects models. Multi-level data are often analyzed using mixed-effects models. Let's take a look at data on high schools in Texas, where we have two levels: school district and school<sup>145</sup>.

<sup>145</sup> These data are from <http://www.itsjustregression.com/datasets.php>.

```

k8 <- read.csv("../data/MLA_Kentucky_Eighth_Graders.csv")
dfsumm(k8)

#
# 48168 rows and 13 columns
# 48168 unique rows
#
#          dis_id sch_id stud_nm    sex ethnic reading
# Class      factor factor factor factor factor   factor
# Minimum    29,784.00 1,010 10,010     1     1    1.00
# Maximum    30,371.00 95,040 98,574 #NULL! #NULL! #NULL!
# Mean       30,057.00 305,040 30,594     1     1    52.00
# Unique (excl. NA) 132     235 48168     3     6   100
# Missing values 0       0     0     0     0     0
# Sorted      FALSE  FALSE  FALSE  FALSE  FALSE  FALSE
#
#          math sch_size sch_ses dis_size dis_ses locale
# Class      factor numeric numeric numeric numeric factor
# Minimum    1.00   -1.93  -49.9   -2.16  -47.1     2
# Maximum   #NULL!    1.89   40.4    3.71   42.6 #NULL!
# Mean       51.00   0.918  -15.6    1.09  -2.97     8
# Unique (excl. NA) 100     161   233     132   166     8
# Missing values 0       0     0     0     0     0
# Sorted      FALSE  FALSE  FALSE  FALSE  FALSE  FALSE
#
#          ethnic1
# Class      factor
# Minimum    0.00
# Maximum   #NULL!
# Mean       0.00
# Unique (excl. NA) 3
# Missing values 0
# Sorted      FALSE

```

It looks like missing values were coded as `#NULL!`. Let's try again.

```

k8 <- read.csv("../data/MLA_Kentucky_Eighth_Graders.csv", na.string = "#NULL!")
dfsumm(k8)

```

```

#
# 48168 rows and 13 columns
# 48168 unique rows
#
#          dis_id sch_id stud_nm    sex ethnic reading
# Class      factor factor factor integer integer numeric
# Minimum    29,784.00 1,010 10,010     1     1     1
# Maximum    30,371.00 95,040 98,574     2     5    99
# Mean       30,057.00 305,040 30,594     1     1    51.8
# Unique (excl. NA) 132     235 48168     2     5    99
# Missing values 0       0     0     249   1122    84
# Sorted      FALSE  FALSE  FALSE  FALSE  FALSE  FALSE
#
#          math sch_size sch_ses dis_size dis_ses locale
# Class      numeric numeric numeric numeric numeric integer
# Minimum    1       -1.93  -49.9   -2.16  -47.1     2
# Maximum    99      1.89   40.4    3.71   42.6     8
# Mean       49.8    0.918  -15.6    1.09  -2.97     6

```

```

# Unique (excl. NA)      99      161      233      132      166      7
# Missing values        110       0       0       0       0   38787
# Sorted                  FALSE     FALSE     FALSE     FALSE     FALSE     FALSE
#                                         ethnic1
# Class                      numeric
# Minimum                     0
# Maximum                     1
# Mean                      0.124
# Unique (excl. NA)        2
# Missing values        1122
# Sorted                  FALSE

```

That looks better. Note that district and school ids are already factors. But I'm not crazy about the commas and the unnecessary .00 in the levels, so let's remove them.

```

levels(k8$dis_id) <- gsub("\\.00", "", gsub(" ", " ", levels(k8$dis_id)))
levels(k8$sch_id) <- gsub(" ", " ", levels(k8$sch_id))
dfsumm(k8)

#
# 48168 rows and 13 columns
# 48168 unique rows
#
#          dis_id sch_id stud_nm    sex ethnic reading
# Class      factor factor factor integer integer numeric
# Minimum    29,784  1,010  10,010      1      1      1
# Maximum    30,371  95,040  98,574      2      5     99
# Mean       30,057 305,040  30,594      1      1    51.8
# Unique (excl. NA) 132      235    48168      2      5     99
# Missing values  0       0       0    249    1122     84
# Sorted      FALSE  FALSE  FALSE  FALSE  FALSE  FALSE
#                                         math sch_size sch_ses dis_size dis_ses locale
# Class      numeric numeric numeric numeric numeric integer
# Minimum      1    -1.93   -49.9    -2.16   -47.1      2
# Maximum      99     1.89    40.4     3.71    42.6      8
# Mean        49.8    0.918   -15.6     1.09   -2.97      6
# Unique (excl. NA) 99      161      233      132      166      7
# Missing values 110       0       0       0       0   38787
# Sorted      FALSE  FALSE  FALSE  FALSE  FALSE  FALSE
#                                         ethnic1
# Class                      numeric
# Minimum                     0
# Maximum                     1
# Mean                      0.124
# Unique (excl. NA)        2
# Missing values        1122
# Sorted                  FALSE

```

We need to make `sex` and `ethnic` factors.

```

k8$sex <- factor(k8$sex, labels = c("female", "male"))
k8$ethnic <- factor(k8$ethnic)

```

The potential response variables that we are interested in are standardized test results: `reading` and `math`. We might be interested if these scores are related to student sex or ethnicity. We need to consider the structure of the multilevel or nested structure of the data in any model that we fit—otherwise we are assuming independence that we don't have and will in general overestimate the significance of effects.

```
modm1 <- lmer(math ~ sex + ethnic + (1|dis_id) + (1|dis_id:sch_id), data = k8)
```

In this model we've included two random variables: `dis_id` (school district) and `sch_id`, nested within `dis_id`. Let's take a look at the results.

```
modm1
```

```
# Linear mixed model fit by REML ['lmerMod']
# Formula: math ~ sex + ethnic + (1 | dis_id) + (1 | dis_id:sch_id)
#   Data: k8
# REML criterion at convergence: 413188.2
# Random effects:
#   Groups      Name      Std.Dev.
#   dis_id:sch_id (Intercept) 5.413
#   dis_id        (Intercept) 3.099
#   Residual          19.959
# Number of obs: 46746, groups: dis_id:sch_id, 236; dis_id, 132
# Fixed Effects:
# (Intercept)    sexmale    ethnic2    ethnic3    ethnic4
#   50.827     -1.119     -15.056     -7.030     10.165
#   ethnic5
#   -2.895
```

We see estimates of the variance among districts and schools at the top. For fixed effects, there are some highly significant responses. As we saw with `lm()` above, `lmer()` used the first level of each factor as the default group in the model. We would expect the second column in the `X` matrix to be the `male` column.

```
model.matrix(modm1)[1:5, 2]

# 1 2 3 4 5
# 1 0 1 1 0

k8$sex[1:5]

# [1] male   female male   male   female
# Levels: female male
```

The model summary shows a small but very significant effect of sex—a difference of about 1.1 on a test score with an apparent maximum of 100. Apparent effects of ethnicity are much higher—from -15 to +10 points, compared to the first level, which is white. To test the overall effect of a factor, we can use `anova()`.

```

anova(modm1)

# Analysis of Variance Table
#          Df Sum Sq Mean Sq F value
# sex       1 15130  15130   37.98
# ethnic    4 795628 198907  499.31

```

We might be interested in interactions between sex and ethnicity.

```

modm2 <- lmer(math ~ sex*ethnic + (1|dis_id) + (1|dis_id:sch_id), data = k8)
modm2

```

```

# Linear mixed model fit by REML ['lmerMod']
# Formula: math ~ sex * ethnic + (1 | dis_id) + (1 | dis_id:sch_id)
#   Data: k8
# REML criterion at convergence: 413169.8
# Random effects:
# Groups      Name        Std.Dev.
# dis_id:sch_id (Intercept) 5.411
# dis_id         (Intercept) 3.100
# Residual      19.958
# Number of obs: 46746, groups: dis_id:sch_id, 236; dis_id, 132
# Fixed Effects:
#   (Intercept)      sexmale      ethnic2      ethnic3
#   50.7802       -1.0256     -14.9326     -4.9220
#   ethnic4       ethnic5  sexmale:ethnic2  sexmale:ethnic3
#   11.9498       -2.6289      -0.2513     -4.7761
# sexmale:ethnic4 sexmale:ethnic5
#   -3.6678       -0.5338

```

```

anova(modm2)

```

```

# Analysis of Variance Table
#          Df Sum Sq Mean Sq F value
# sex       1 15130  15130   37.9831
# ethnic    4 795631 198908  499.3553
# sex:ethnic 4   3158    790   1.9821

```

This interaction appears to be non-significant, or possible borderline. Let's use a LRT with `anova()`.

```

anova(modm1, modm2)

```

```

# refitting model(s) with ML (instead of REML)

# Data: k8
# Models:
# modm1: math ~ sex + ethnic + (1 | dis_id) + (1 | dis_id:sch_id)
# modm2: math ~ sex * ethnic + (1 | dis_id) + (1 | dis_id:sch_id)
#          Df      AIC      BIC  logLik deviance Chisq Chi Df Pr(>Chisq)

```

```
# modm1 9 413210 413289 -206596 413192
# modm2 13 413210 413324 -206592 413184 7.9293      4      0.0942 .
# ---
# Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Let's ignore it. Do the effects of sex and ethnicity depend on the school?

```
modm3 <- lmer(math ~ sex + ethnic + (1|dis_id) + (1|dis_id:sch_id) + (sex|dis_id:sch_id), data = k8)
anova(modm1, modm3)
```

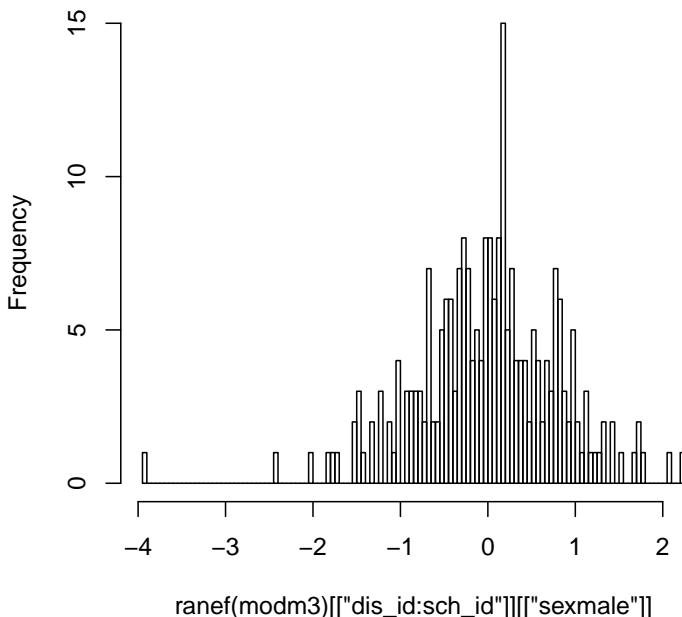
*# refitting model(s) with ML (instead of REML)*

```
# Data: k8
# Models:
# modm1: math ~ sex + ethnic + (1 | dis_id) + (1 | dis_id:sch_id)
# modm3: math ~ sex + ethnic + (1 | dis_id) + (1 | dis_id:sch_id) + (sex | 
# modm3:     dis_id:sch_id)
#       Df   AIC   BIC logLik deviance Chisq Chi Df Pr(>Chisq)
# modm1 9 413210 413289 -206596 413192
# modm3 12 413199 413304 -206588 413175 17.28      3 0.000619 ***
# ---
# Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

This LRT provides some strong evidence that they do—perhaps surprising considering the small size of the overall effect. Let's see how the effect varies among schools.

```
hist(ranef(modm3)[["dis_id:sch_id"]][["sexmale"]], breaks = 100)
```

**Histogram of ranef(modm3)[["dis\_id:sch\_id"]][["sexmale"]]**



We could do the same for district.

Let's compare predictions to observations.

```
k8$mathpred <- fitted(modm2)

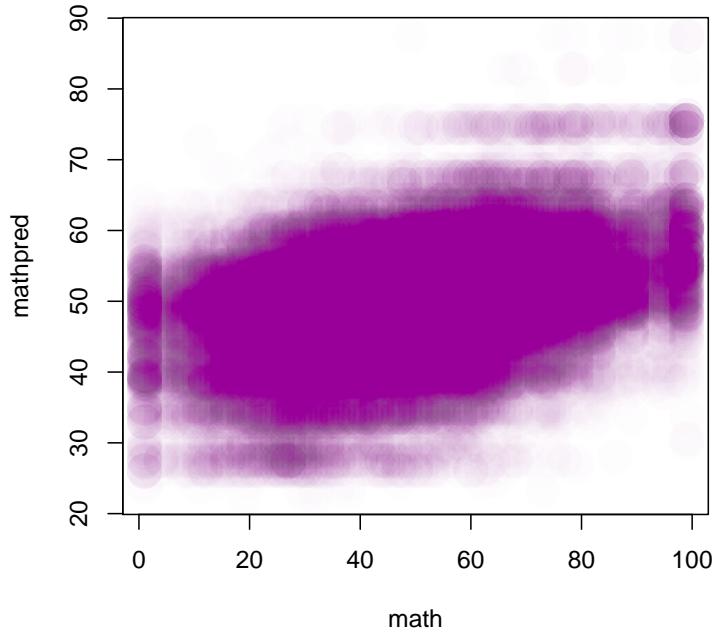
# Error in '$<-.data.frame'('*tmp*', mathpred, value = structure(c(47.3913685305297,
:   replacement has 46746 rows, data has 48168
```

This error is due to missing observations being removed by `lmer()`.

```
k8 <- k8[, c("dis_id", "sch_id", "sex", "ethnic", "math")]
k8 <- na.omit(k8)
```

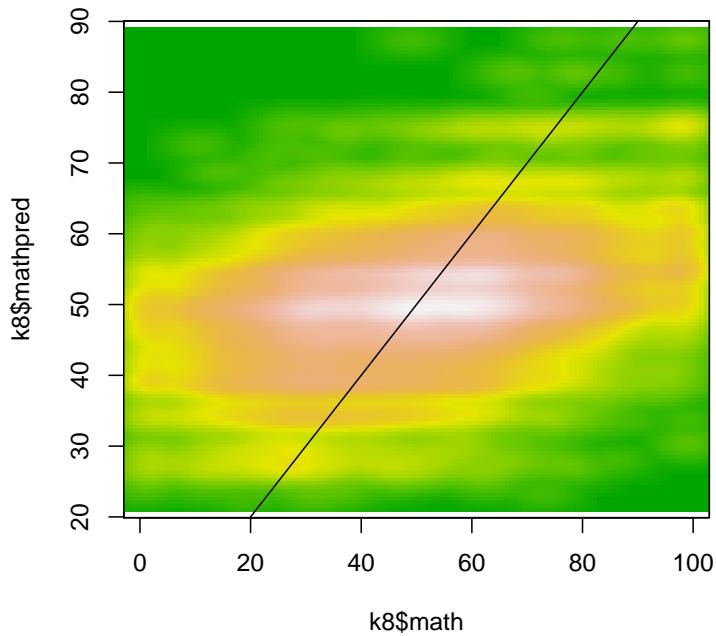
```
k8$mathpred <- fitted(modm2)
```

```
plot(mathpred ~ math, data = k8, pch = 19, cex = 3, col = "#99009902")
```



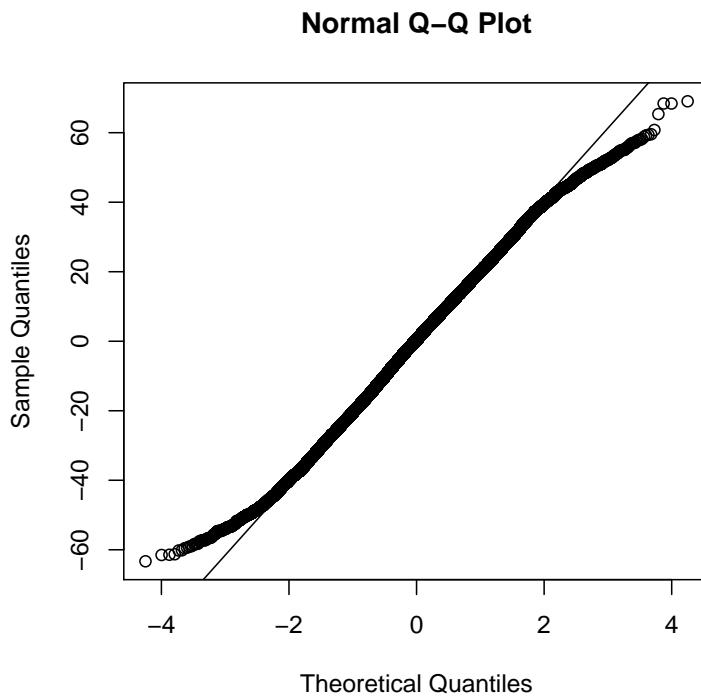
It is difficult to see much with all the overlap.

```
smoothScatter(k8$math, k8$mathpred, nrpoints = 0, colramp = terrain.colors)
abline(0, 1)
```



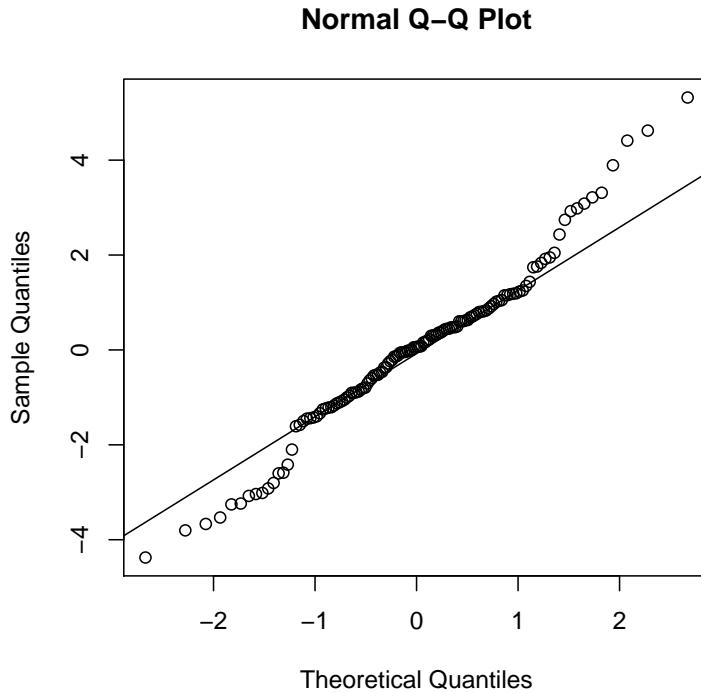
Let's take a look at the residuals.

```
qqnorm(resid(modm2))  
qqline(resid(modm2))
```



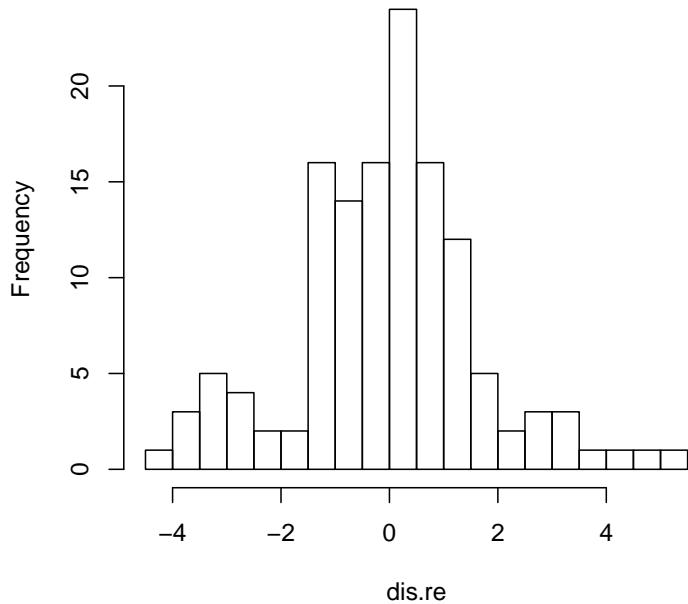
It looks like we've violated assumptions about normally distributed error and constant variance. Let's look at the random effects.

```
dis.re <- ranef(modm2)$dis_id[[1]]  
qqnorm(dis.re)  
qqline(dis.re)
```



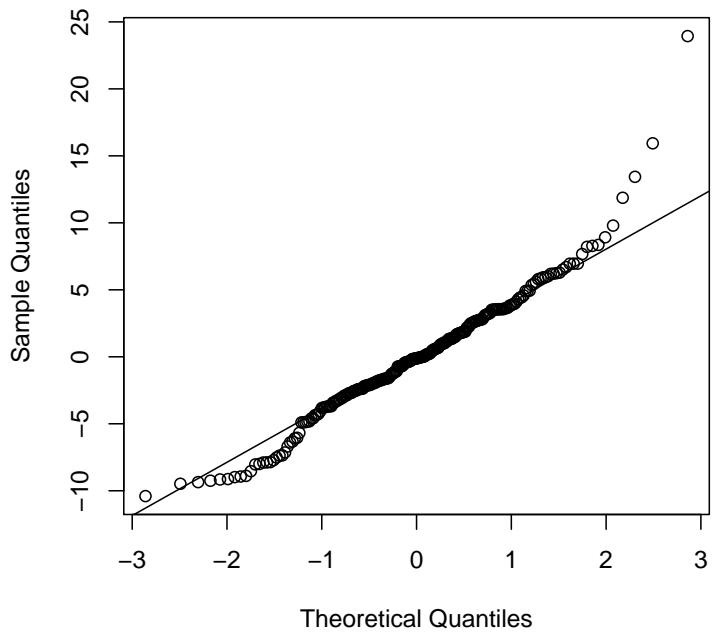
```
hist(dis.re, breaks = 25)
```

**Histogram of dis.re**

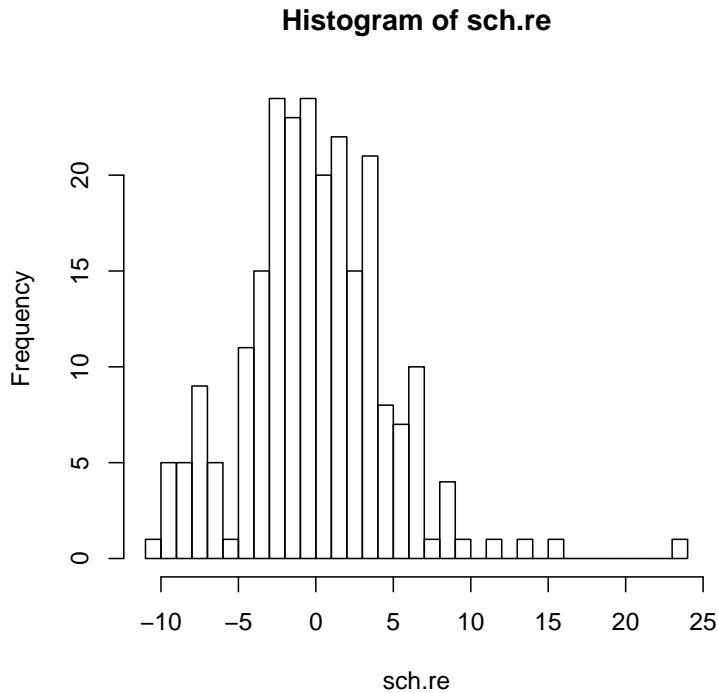


```
sch.re <- ranef(modm2)$`dis_id:sch_id`[[1]]  
qqnorm(sch.re)  
qqline(sch.re)
```

**Normal Q–Q Plot**

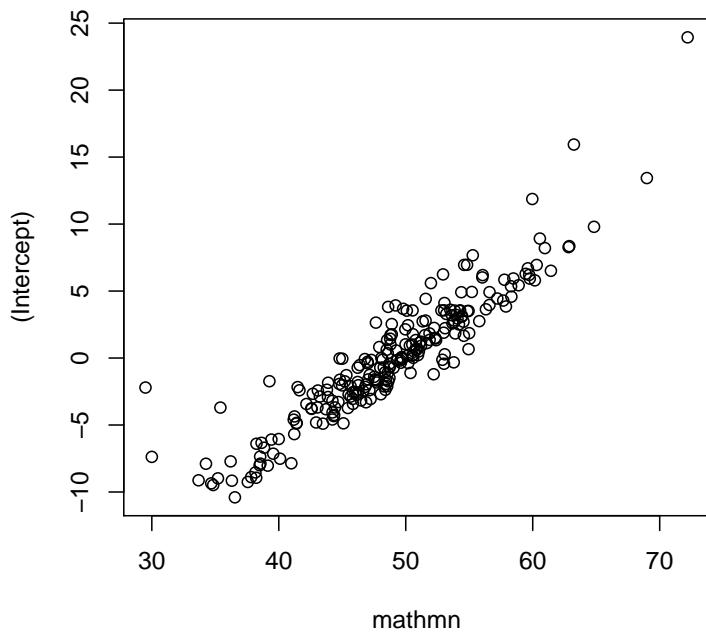


```
hist(sch.re, breaks = 25)
```



Following Faraway (2006), we can use our model to attempt to isolate the effectiveness of each school once scores are corrected for characteristics of the student body. The estimates of the random effects are estimates of these exact effects.

```
k8$dis_sch <- paste(k8$dis_id, k8$sch_id, sep = ":")  
ssumm <- ddply(k8, "dis_sch", summarise, mathmn = mean(math))  
ssumm <- merge(ssumm, ranef(modm2)$`dis_id:sch_id`, by.x = "dis_sch", by.y = "row.names")  
plot(`(Intercept)` ~ mathmn, data = ssumm)
```



It looks like those schools that do a better job educating students tend to have higher test scores—student ethnicity doesn't isn't the whole story.

## 23 Nonlinear regression

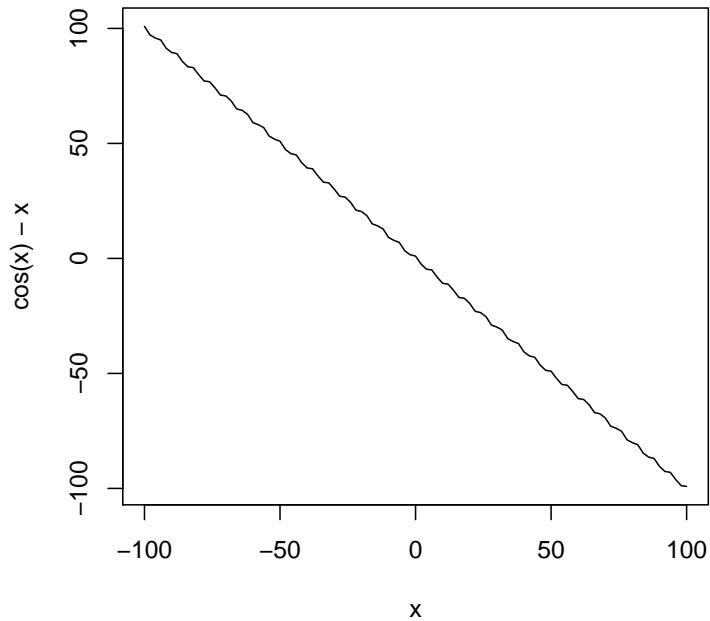
### 23.1 A little bit on optimization

R has multiple functions that can be used for optimization problems that are not like the `lm` and `glm` problems we've covered. These include one-parameter problems and more complex problems. You can find a list of R functions for these tasks in the Task View on Optimization and Mathematical Programming<sup>146</sup>. Non-linear regression is based on the same approaches, but it is generally not grouped with these solvers. However, I've grouped them together here.

For simple single-parameter optimization, `optimize` is the best bet. For example, say you want to find the root of an equation:

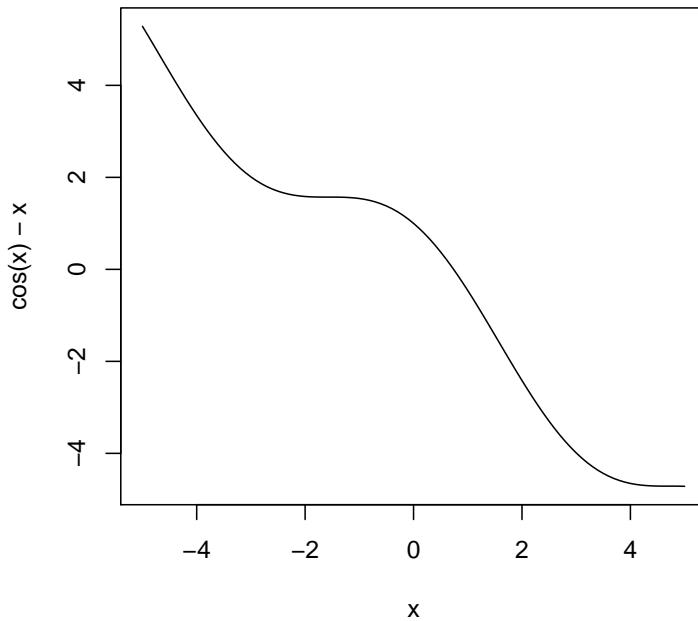
$$y = \cos x - x$$

```
curve(cos(x) - x, xlim = c(-100, 100))
```



```
curve(cos(x) - x, xlim = c(-5, 5))
```

<sup>146</sup> <http://cran.r-project.org/web/views/Optimization.html>



```

optimize(function(x) abs(cos(x) - x), interval = c(0, 3))

# $minimum
# [1] 0.7391
#
# $objective
# [1] 2.493002e-05

optimize(function(x) abs(cos(x) - x), interval = c(0, 3), tol = 1E-10)

# $minimum
# [1] 0.7390851
#
# $objective
# [1] 6.533913e-09

```

Let's try it.

```

cos(0.7390851) - 0.7390851

# [1] 5.558929e-08

```

The `optimize` function could be used for much more complicated functions. For multi-parameter optimization, there is a flexible `optim` function.

For nonlinear regression, there are two functions:

## 23.2 The nls function

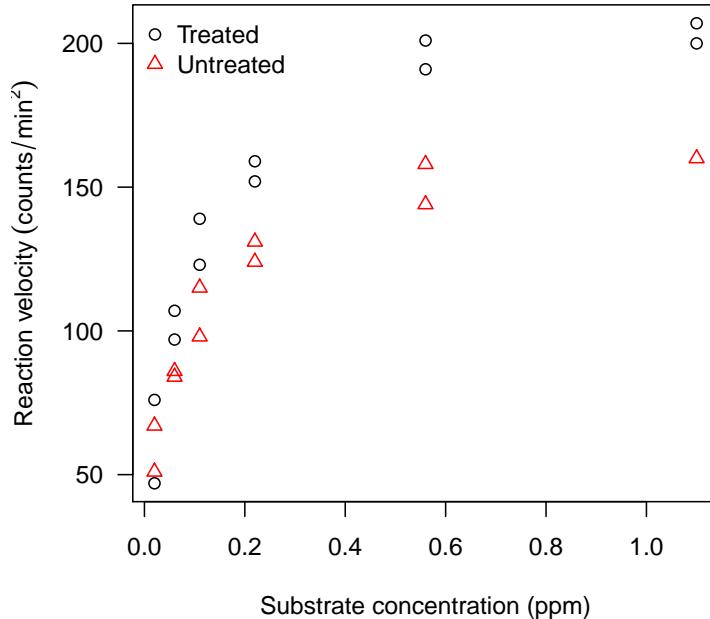
Let's demonstrate `nls()` with `Puromycin`, which is included in the `datasets` package. This data set contains data on the reaction rate of an enzymatic reaction with and without treatment with the antibiotic `puromycin`.

```
puro <- Puromycin
dfsumm(puro)

#
# 23 rows and 3 columns
# 23 unique rows
#
#          conc      rate     state
# Class    numeric  numeric   factor
# Minimum   0.02      47 treated
# Maximum   1.1       207 untreated
# Mean      0.312     127 treated
# Unique (excl. NA) 6       23      2
# Missing values 0       0       0
# Sorted      FALSE     FALSE    TRUE

plot(rate ~ conc, las = 1, xlab = "Substrate concentration (ppm)",
      ylab = expression("Reaction velocity"~("counts"/min^2)),
      pch = as.numeric(state), col = as.numeric(state),
      data = puro)

legend("topleft", c("Treated", "Untreated"), pch = 1:2, col = 1:2, bty = "n")
```



To model the response of reaction rate to substrate concentration, the Michaelis-Menten equation is suitable.

$$rate = \frac{V_{max}c}{1 + K_m c}$$

The asymptote  $V_{max}$  is the maximum reaction velocity, and  $K_m$  is the Michaelis-Menten constant, which turns out to be the value of `conc` where the rate reaches half of the maximum. We will need to provide starting estimates, and we can use this information to make some estimates from the plot. We can start by fitting a model to the “treated” group.

```
mod1 <- nls(rate ~ Vm*conc/(K + conc), data = puro, start = c(Vm = 250, K = 0.1),
             trace = TRUE, subset = state == "treated")
```

```
# 3993.976 : 250.0   0.1
# 1196.486 : 212.02378920   0.06342989
# 1195.456 : 212.63961237   0.06405238
# 1195.449 : 212.67946866   0.06411461
# 1195.449 : 212.68333082   0.06412064
# 1195.449 : 212.68370329   0.06412122
```

By specifying `trace = TRUE`, we get `nls` to print out its iterations as it runs. The first column gives the sum of residuals (sum of squares), and parameter estimates follow in the order they are specified in the `start` argument setting (headings would be nice though).

As with `lm` and other functions for fitting statistical models, there is a `summary()` method for `nls` objects.

```
summary(mod1)

#
# Formula: rate ~ Vm * conc/(K + conc)
#
# Parameters:
#   Estimate Std. Error t value Pr(>|t|)
#   Vm 2.127e+02 6.947e+00 30.615 3.24e-11 ***
#   K  6.412e-02 8.281e-03  7.743 1.57e-05 ***
#
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 10.93 on 10 degrees of freedom
#
# Number of iterations to convergence: 5
# Achieved convergence tolerance: 2.153e-06
```

And the extractor functions that we used earlier can also be used.

```
coef(mod1)
```

```
#           Vm            K
# 212.68370329  0.06412122
```

```

predict(mod1)

# [1] 50.56601 50.56601 102.81096 102.81096 134.36161 134.36161
# [7] 164.68469 164.68469 190.83292 190.83292 200.96882 200.96882

```

You can get pretty creative with `nls()`—you can fit just about any model that you write. For example, what if we wanted to fit the model to all our data, but just shift  $V_m$  up for the treated results?

```

mod2 <- nls(rate ~ (Vm + a*(state == "treated"))*conc/(K + conc),
             data = puro, start = c(Vm = 250, K = 0.1, a = 50), trace = TRUE)

# 43320.93 : 250.0   0.1   50.0
# 2593.529 : 164.64141001   0.06387244  43.48831034
# 2241.449 : 166.88784786   0.05833299  42.14843806
# 2240.897 : 166.62490934   0.05801384  42.03629096
# 2240.892 : 166.60654417   0.05797677  42.02718551
# 2240.891 : 166.60438474   0.05797241  42.02611584
# 2240.891 : 166.6041307   0.0579719   42.0259904

```

The performance of the Gauss-Newton algorithm that does the work in `nls` can be dependent on the quality of initial guesses you give it. If you provide guesses reasonably close to the true values, `nls` will probably do a good job. If you provide very poor guesses, it may not be able to find least-squares estimates. For some common models that can be linearized, R has self-starting functions, which estimate starting values automatically using a transformation and linear regression. Here is an example of the models fit in the example above, but this time, a self-starting function (`SSmicmen`) is used:

```

mod3 <- nls(rate ~ SSmicmen(conc, Vm, K), data = puro, subset = state == "untreated")
summary(mod3)

#
# Formula: rate ~ SSmicmen(conc, Vm, K)
#
# Parameters:
#   Estimate Std. Error t value Pr(>|t|)
#   Vm 1.603e+02 6.480e+00 24.734 1.38e-09 ***
#   K  4.771e-02 7.782e-03   6.131 0.000173 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 9.773 on 9 degrees of freedom
#
# Number of iterations to convergence: 5
# Achieved convergence tolerance: 3.942e-06

```

It looks like `nls` actually couldn't improve on the initial guesses.

The `nls` function is very flexible, and can, in theory, be used with any model that you can specify within the function call. Common nonlinear models are given in Ritz and Streibig [15, Table B.1]. Several of these models have associated self-starting functions in R.

Use of the `nls` function is not limited to models that can be specified as an algebraic expression. The formula argument can actually be a call to another function, and so it is possible to use `nls` to calibrate a complicated numeric model, for example.

### 23.3 The `nls.lm()` function

As mentioned above, the `nls()` function uses (by default) the Gauss-Newton algorithm, which does not perform well for all data. A similar function, `nls.lm()`, available in the `minpack.lm` package, uses the Levenberg-Marquardt algorithm, which may be more reliable. I have had good luck with `nls.lm()` on problems that `nls()` has had difficulty with. This function is set up differently from `nls()`—instead of a formula, you need to provide a function that calculates residuals.

```
library(minpack.lm)
resid.f <- function(par) {
  puro$rate -
    (par$Vm+par$a*(puro$state == "treated"))*puro$conc/(par$K + puro$conc)
}

mod4 <- nls.lm(par = list(Vm = 250, K = 0.1, a = 50),
               fn = resid.f, control = nls.lm.control(nprint = 1))

# It.      0, RSS =   43320.9, Par. =       250       0.1       50
# It.      1, RSS =   2593.53, Par. =     164.641  0.0638724  43.4883
# It.      2, RSS =   2241.45, Par. =     166.888  0.058333  42.1484
# It.      3, RSS =   2240.9, Par. =     166.625  0.0580138  42.0363
# It.      4, RSS =   2240.89, Par. =     166.607  0.0579768  42.0272
# It.      5, RSS =   2240.89, Par. =     166.604  0.0579724  42.0261
# It.      6, RSS =   2240.89, Par. =     166.604  0.0579719  42.026

mod4

# Nonlinear regression via the Levenberg-Marquardt algorithm
# parameter estimates: 166.604130914708, 0.0579719014683139, 42.0259903947668
# residual sum-of-squares: 2241
# reason terminated: Relative error in the sum of squares is at most `ftol'.
```

### 23.4 Other functions

The `optim()` function may perform well, and other options exist. A search for simulated annealing functions in R will turn up some other options.

## 24 Basic R programming

With R, it is possible to write script files (scripts) that can be run later, and also to write functions that can be used for streamlining data analysis or graphics development. Programming in R benefits greatly from grouping, loops, and constructs for conditional execution. In some cases, interactive R sessions and simple scripts can also make use of these features.

### 24.1 Loops and grouping

Loops are a common feature in most programming languages—they allow you to repeat a command or a set of commands any number of times. In R it is possible to avoid using loops for many procedures where they would be required in other languages, by taking advantage of vectorized operations and indexing. Vectorized operations are generally more efficient than loops from both the standpoint of both writing and executing code. However, in some cases, loops may be more clear, or may be the only option.

R has three types of loops: `for`, `while`, and `repeat`. A `for` loop will repeat a set of commands a specified number of times and change the value of a counter variable with every pass.

```
for (i in 1:10) print(i)

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

The counter variable can be any type of vector—it does not need to be numeric. As you can see from the above examples, the command that follows the `for()` is executed with each pass of the loop. If you want to execute more than one command, simply group them using braces: `{}`. Any commands present together in braces are executed together—the entire block of code within the braces is referred to as a compound expression.<sup>147</sup> Here is a simple population model of rabbits with three age classes that uses a compound expression. Each mature female rabbit (doe) gives birth to 8 offspring in each cycle.

```
n.bucks <- 10
n.does <- 5
n.kits <- 0
for (season in 1:10) {
  n.does <- n.does + 0.5 * n.kits - round(0.1 * n.does)
  n.bucks <- n.bucks + 0.5 * n.kits - round(0.2 * n.bucks)
  n.kits <- (n.bucks > 0) * 8 * n.does
  n.total <- n.does + n.bucks + n.kits
  cat(season, n.total, "\n")
}
```

<sup>147</sup> Loops are not the only place where grouping is useful.

```
# 1 53
# 2 251
# 3 1228
# 4 6016
# 5 29478
# 6 144437
# 7 707740
# 8 3467922
# 9 16992814
# 10 83264780
```

Each iteration of our loop represents a breeding period—much shorter than a year for rabbits<sup>148</sup>.

One common use of loops is to repeat a set of commands for subsets of data. For many cases, however, vectorized operations can take the place of loops, resulting in simpler and faster code.

The `while` loop is used to repeat a set of commands until some condition is met. Here is an entirely useless example<sup>149</sup>.

```
answer <- sample(1:100, 1)
guess <- answer-1
while(guess != answer) {
  cat("\nGuess what the answer is")
  guess <- scan(n = 1)
  if (answer<guess) cat("\nLower. . .")
  if (answer>guess) cat("\nHigher. . .")
  if (answer == guess) cat("\nCorrect!")
}
```

```
Guess what the answer is
1: 50
Read 1 item
Lower. . .
Guess what the answer is
1: 25
Read 1 item
Lower. . .
....
Guess what the answer is
1: 7
Read 1 item
Correct!
```

A more interesting use of a `while` loop is in root-finding functions, where guesses are updated until some criterion is met. For example, say we want to find the root of  $\cos(x) - x$ . One approach is to use the secant method, which uses the slope between two points to make a guess of where the function is equal to zero. To apply it, we need to make successive guesses until the error is smaller than some threshold.

---

<sup>148</sup> According to a Wikipedia article, the gestation period for most rabbits is around 30 days, and young (kits) are generally weaned after five weeks. However, kits are not mature until after several months, so our model is not really accurate. Still, 83 million rabbits over 10 iterations is frightening.

<sup>149</sup> The `cat` function is used for printing data to the console. The functions `print` and `message` are alternatives that behave somewhat differently.

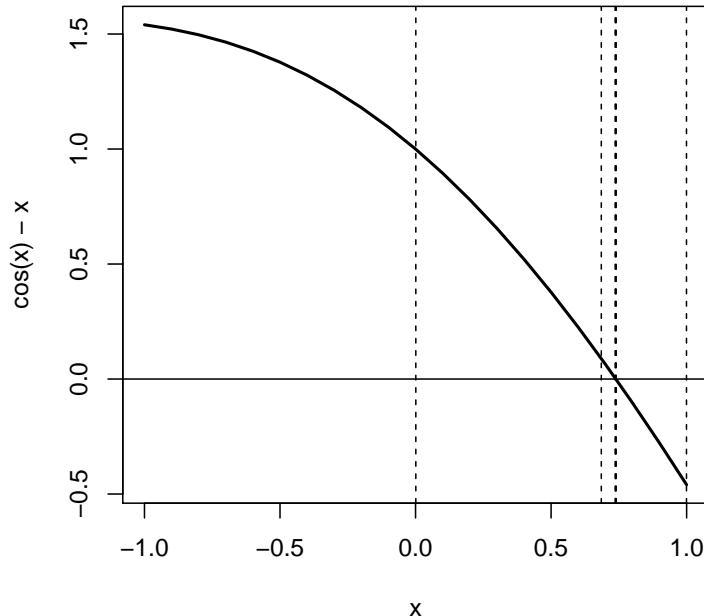
```

x<- -10:10/10
plot(x, cos(x) - x, type = "l", lwd = 2)
abline(h = 0)

x0 <- 0
x1 <- 0.001
y1 <- 10

while(abs(y1)>1E-10) {
  abline(v = x1, lty = 2)
  y0 <- cos(x0) - x0
  y1 <- cos(x1) - x1
  x2 <- x1
  s <- (y0 - y1)/(x0 - x1)
  x1 <- x0 - y0/s
  x0 <- x2
  cat(paste("\nx is ", signif(x1, 6), ", and y is ", signif(y1, 6), "\n", sep = ""))
}

```



```

#
# x is 0.9995, and y is 0.999
#
# x is 0.685262, and y is -0.458777
#
# x is 0.736314, and y is 0.0889914
#
# x is 0.739119, and y is 0.00463568
#

```

```
# x is 0.739085, and y is -5.67814e-05
#
# x is 0.739085, and y is 3.47985e-08
#
# x is 0.739085, and y is 2.6068e-13
```

A `repeat` loop continues until a `break` command is executed. To use them, incorporate the `break` command in a conditional statement (see the next subsection, section 24.2). Usually, one of the other loop types is a better choice.

## 24.2 Conditional statements

Conditional statements are another ubiquitous feature of programming languages. In R, there are two conditional statements, `if...else` and `ifelse`, and they are designed for two different applications. The `if` construct will execute a command if a specified condition is met.

```
x <- 0
if (10 > 3) x <- 101
x

# [1] 101

a <- 10
if (!is.na(a)) print(a)

# [1] 10
```

The condition in an `if` statement should always return a length-one logical vector.<sup>150</sup> Otherwise you will receive a warning, and perhaps at some point in the future, an error.

```
if (1:5 > a) print(a)

# Warning in if (1:5 > a) print(a): the condition has length > 1 and only the first
element will be used
```

If you would like several commands to be dependent on a single condition, then simply group the commands together using braces to make a compound command.

```
score <- 981423
record <- 750270
name <- "Alex Rogan"
if (score > record) {
  cat("\nYou broke the record!\n")
  recordholder <- name
  record <- score
}
```

<sup>150</sup> see `ifelse()` for a vectorized version, below.

```
#  
# You broke the record!
```

The `if` construct can include an `else`. The command that follows an `else` will be executed if the condition is not met.

```
score <- 56  
record <- 750270  
name <- "Sasha Hafner"  
if (score > record) {  
  cat("\nYou broke the record!\n")  
  recordholder <- name  
  record <- score  
} else {  
  cat("\n", name, ", you are terrible!\n", sep = "")  
  cat("\nThe current record is ", record, "\n", sep = "")  
  cat("\nThe current record holder is ", recordholder, "\n", sep = "")  
}  
  
#  
# Sasha Hafner, you are terrible!  
#  
# The current record is 750270  
#  
# The current record holder is Alex Rogan
```

And, of course, an `if` can follow an `else`.

```
score <- 56  
record <- 750270  
name <- "Sasha Hafner"  
if (score > record) {  
  cat("\nYou broke the record!\n")  
  recordholder <- name  
  record <- score  
} else if (score < 0.5*record) {  
  cat("\n", name, ", you are terrible!\n", sep = "")  
  cat("\nThe current record is ", record, "\n", sep = "")  
  cat("\nThe current record holder is ", recordholder, "\n", sep = "")  
} else {  
  cat("\n", name, ", you aren't that bad--keep trying!\n", sep = "")  
  cat("\nThe current record is ", record, "\n")  
  cat("\nThe current record holder is ", recordholder, "\n", sep = "")  
}  
  
#  
# Sasha Hafner, you are terrible!  
#  
# The current record is 750270  
#  
# The current record holder is Alex Rogan
```

If you want to apply an `if...else` type construct to multiple elements in a data structure, use `ifelse`.

```
grades <- c(82, 64, 95, 54, 96, 96, 92, 90, 99, 72)
result <- ifelse(grades > 65, "pass", "fail")
result

# [1] "pass" "fail" "pass" "fail" "pass" "pass" "pass" "pass" "pass"
# [10] "pass"
```

The `ifelse` construct is handy for adding to data frames new columns which contains values that are dependent on the value of some other variable(s) in the same data frame.

```
flow <- read.csv("../data/flow_2006_summary.csv")
names(flow)

# [1] "site.id"    "month"      "discharge"

unique(flow$site)

# [1] 1509000 4232730

flow$name <- ifelse(flow$site == 1509000, "Tioughnioga", "Seneca")
head(flow)

#   site.id month discharge      name
# 1 1509000   Jan  37237.615 Tioughnioga
# 2 1509000   Feb  24522.010 Tioughnioga
# 3 1509000   Mar  27158.420 Tioughnioga
# 4 1509000   Apr  16505.757 Tioughnioga
# 5 1509000   May  9830.955 Tioughnioga
# 6 1509000   Jun  27116.135 Tioughnioga

tail(flow)

#   site.id month discharge      name
# 19 4232730   Jul  24587.14 Seneca
# 20 4232730   Aug  11300.95 Seneca
# 21 4232730   Sep       NA Seneca
# 22 4232730   Oct  26051.04 Seneca
# 23 4232730   Nov  49824.33 Seneca
# 24 4232730   Dec  36907.15 Seneca
```

Much of what can be done using `ifelse` can also be done using indexing<sup>151</sup>. One or the other approach may make more sense or use less code in some cases.

<sup>151</sup> In some cases, the `switch` function is a better option than `ifelse`. It can be especially useful within functions. And `merge` may be a better bet for more complicated examples.

## 24.3 Writing simple functions

The capacity to define new functions is a very useful feature of R. Once you define a new function in a console, it is stored in your workspace environment, and effectively does not differ from the functions that come with any R packages. Functions can be entered directly into the R GUI, saved in a script file that can be later loaded, or made into a package that can be loaded. It is also possible to store your function calls (or a source call to a script with your functions) in the file `Rprofile.site` so that they are loaded every time you start R<sup>152</sup>.

To define a function in R, use the following syntax.

```
function_name <- function(arg1, arg2, arg3) expression1
```

If your function requires more than one command (all but the simplest functions will), you can use braces to group them.

Here is a function for calculating the geometric mean of a set of numbers.

```
geomean <- function(x) 10^mean(log10(x))
```

Here our function name is `geomean`, and it expects a single argument `x`. Braces are not needed for grouping since there is only one expression. Also note that the result of the function is not assigned to anything—it is automatically returned when you call up the expression. The last expression in a function is what the function will return. For clarity, you can nest this object or expression in a call to the `return` function.

```
geomean <- function(x) return(10^mean(log10(x)))
```

Let's test out this function.

```
x <- c(100, 1000, 10000)
geomean(x)

# [1] 1000
```

Pretty easy, no?

Moving on to vectorized functions. It is very easy to make a vectorized function—simply use expressions that carry out vectorized operations and return a vector. Here is a simple example for converting lb (mass) to kg.

```
lb2kg <- function(x) 0.453592*x
```

Let's test it.

```
weights <- rnorm(10, mean = 175, sd = 9)
weights
```

---

<sup>152</sup> See the footnotes on p. 80.

```

# [1] 163.9062 167.5148 167.1240 171.2038 179.6039 170.6991 178.2132
# [8] 167.0058 192.6705 160.6145

lb2kg(weights)

# [1] 74.34655 75.98335 75.80610 77.65668 81.46690 77.42776 80.83610
# [8] 75.75250 87.39381 72.85347

```

The next example is more complicated, but the two points I'm trying to make with it are simple: for functions that contain multiple expressions, group them together with braces to make one compound expression, and for complex output, lists are a good fit.

This function is one I wrote for finding peaks in time series data<sup>153</sup>.

```

peaks <- function(x, y, d.min) {
  step <- x[2]-x[1]
  span <- min(length(y), floor(d.min/step))
  if (span%%2 == 0) span <- span+1
  halfspan <- span/2
  y.extended <- c(rep(0, halfspan), y, rep(0, halfspan))
  lagmat <- embed(y.extended, span)
  result <- max.col(lagmat) == 1 + halfspan
  x.pks <- x[result]
  y.pks <- y[result]
  pks <- data.frame(x = x.pks, y = y.pks)
  n.pks <- sum(result)
  return(list(n.pks = n.pks, x = x.pks, y = y.pks, pks = pks))
}

```

Let's test it.

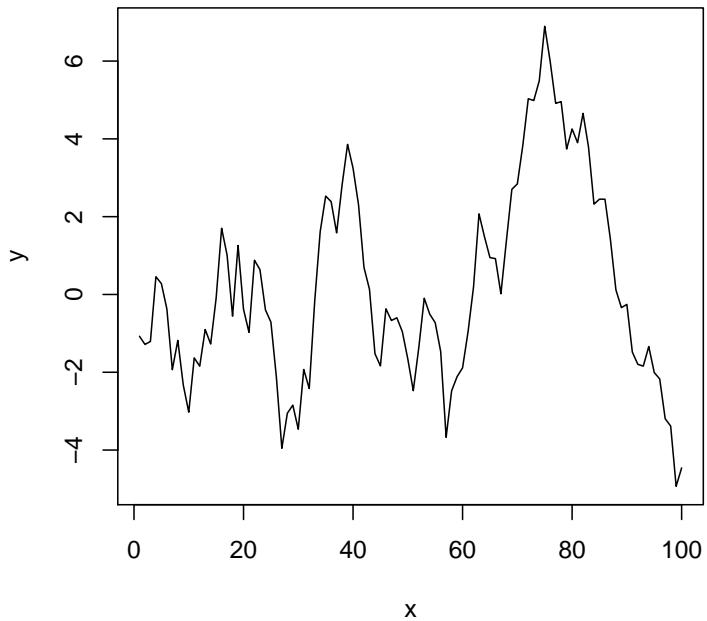
```

dat <- data.frame(x = 1:100, y = cumsum(rnorm(100)))
plot(y ~ x, data = dat, type = "l")

```

---

<sup>153</sup> It is based on the `peaks` function in the `PR0cess` package.



```
pks <- peaks(x = dat$x, y = dat$y, d.min = 5)
```

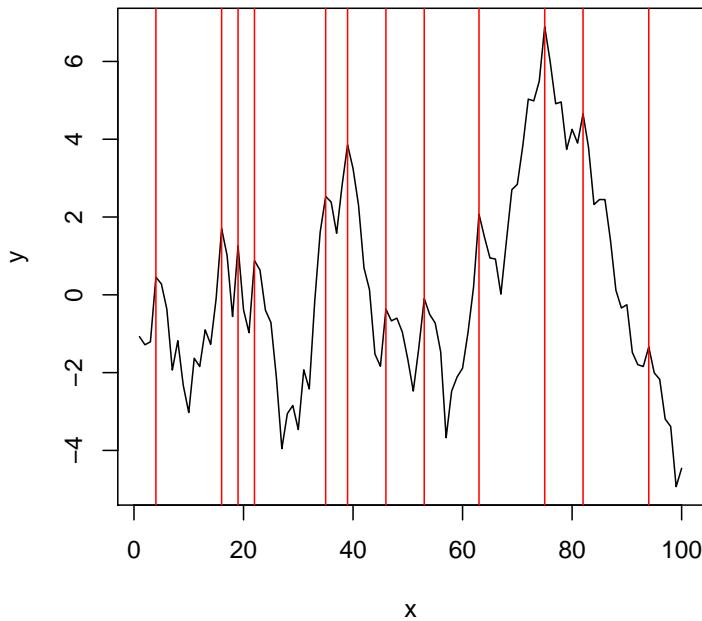
And here is the output.

```
pks
```

```
# $n.pks
# [1] 12
#
# $x
# [1] 4 16 19 22 35 39 46 53 63 75 82 94
#
# $y
# [1] 0.45593045 1.70170851 1.26011183 0.87959530 2.52854840
# [6] 3.85794682 -0.36799874 -0.09692556 2.07282036 6.89396411
# [11] 4.65497569 -1.33865243
#
# $pks
#      x          y
# 1    4  0.45593045
# 2   16  1.70170851
# 3   19  1.26011183
# 4   22  0.87959530
# 5   35  2.52854840
# 6   39  3.85794682
# 7   46 -0.36799874
# 8   53 -0.09692556
# 9   63  2.07282036
```

```
# 10 75  6.89396411
# 11 82   4.65497569
# 12 94  -1.33865243
```

```
plot(y ~ x, data = dat, type = "l")
abline(v = pk$y, col = "red")
```



One of the challenges in developing new functions is that when a function runs, it has its own environment, and therefore will behave differently than if you were to submit each line of function code directly in the console (i.e., in your workspace). Being able to “see” a function as it runs can be helpful for debugging, and you can do this with either the `debug` or `browser` functions. These functions can be handy even for solving problems with functions you did not write. The example below tags the `read.table` function for debugging.

```
debug(read.table)
```

The next time you call `read.table`, you will have access to the function’s environment, and can check variable values at each step as the function is evaluated. To exit, just type `Q`. And when you are done, be sure to submit:

```
undebug(read.table)
```

For getting into a function’s environment at a particular point in the function code, just insert `browser()` at that location, and save, submit, and call the function.

Say we want a function that will calculate the fraction of sample variance explained by a model (a pseudo-r<sup>2</sup> function).

```

pr2 <- function(meas, mod) {
  tss <- sum((meas - mean(meas))^2)
  e <- (mod - meas)^2
  1.0 - sum(e)/tss
}

```

To test this function out, let's use the example of a simple linear regression from above.

```

hard <- read.csv("../data/janka.csv")

mod1 <- lm(hardness ~ density, data = hard)

hard.pred <- predict(mod1)

pr2(meas = hard$hardness, mod = hard.pred)

# [1] 0.9493278

```

Note that the variables defined within the function are not available outside the function they are available only within the function environment and are lost when R exits the function.

```

tss

# Error in eval(expr, envir, enclos): object 'tss' not found

```

As with all the functions we covered, you can use both named and positional arguments.

```

pr2(hard$hardness, hard$hard.pred)

# [1] 1

```

As with other functions, you can use the `args` function to see the arguments.

```

args(pr2)

# function (meas, mod)
# NULL

```

Separation of environments is required for recursive functions, which are allowed in R. For example, this simple-looking function

```

fact <- function(x) {
  if (x>1) out <- x*fact(x-1) else out <- 1
  out
}

```

returns the factorial of any integer by repeatedly calling itself. If it isn't clear to you how this function works (I always have trouble writing and understanding recursive functions, and it probably took me 30 minutes to get this one to work), don't worry, but be aware that recursive functions are allowed and used in R.

```
fact(4)
```

```
# [1] 24
```

But, to get back to the more general idea, that it is very common to call up functions from within other functions, let's look at a more straight-forward example. Let's say we want a function for producing partial regression plots<sup>154</sup>.

```
prPlot <- function(mod, which.pred) {  
  x <- model.matrix(mod) [, which.pred]  
  y <- mod$coef[which.pred]*x + mod$res  
  plot(x, y)  
  abline(0, mod$coef[which.pred])  
}
```

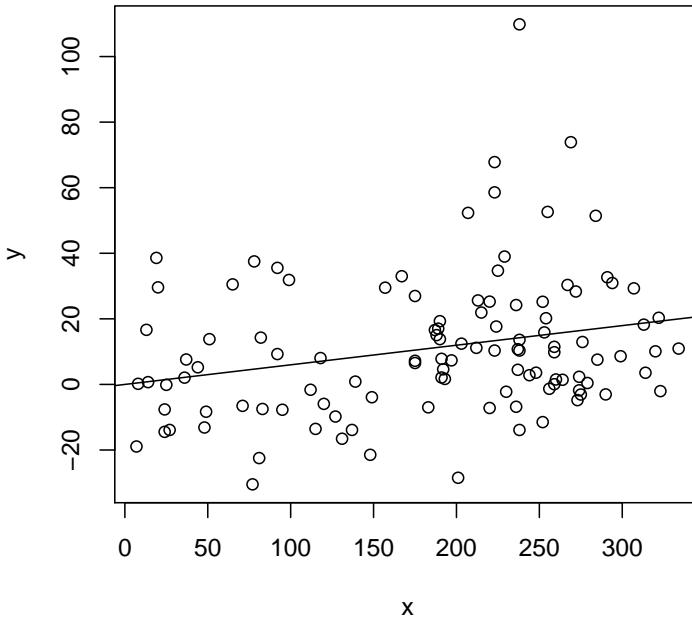
Note that this function doesn't actually return any value (actually it returns `NULL`). It produces a plot as a side effect, but that is it. We can test out this function with a multiple regression example described above.

```
ozone <- read.csv("../data/ozone.csv")
```

```
mod <- lm(ozone~rad + temp + wind, data = ozone)  
prPlot(mod, "rad")
```

---

<sup>154</sup> This function is based on one by Julian Faraway.



This function works, but we certainly could improve it. For starters, let's add axis labels, and include some default values.

```
prPlot <- function(mod, which.pred, xlab = which.pred, ylab = "Partial residuals") {
  x <- model.matrix(mod)[, which.pred]
  y <- mod$coef[which.pred]*x + mod$res
  plot(x, y, xlab = xlab, ylab = ylab)
  abline(0, mod$coef[which.pred])
}
```

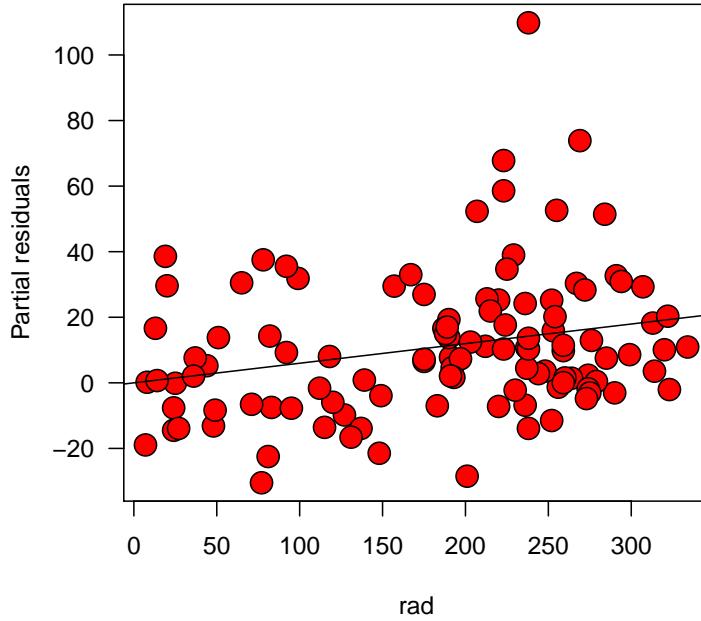
Notice how the function will now use the argument value for `which.pred` as the default value for the x axis label? Of course, a user could override that value by simply specifying a value for the `xlab` argument when calling up `prPlot`. And, notice how the same argument name is used in the `prPlot` function as in the `plot` function for the axis labels? This isn't necessary, but I think it is the easiest approach. Perhaps this function should allow for users to modify additional graphical parameters, say the plotting symbol color or shape. We could add additional arguments, but a better approach is to use an ellipsis, that is, ....

```
prPlot <- function(mod, which.pred, xlab = which.pred, ylab = "Partial residuals", ...){
  x <- model.matrix(mod)[, which.pred]
  y <- mod$coef[which.pred]*x + mod$res
  plot(x, y, xlab = xlab, ylab = ylab, ...)
  abline(0, mod$coef[which.pred])
}
```

This version of the function will accept additional arguments that are not listed in the function definition. Any additional arguments will be passed onto `plot` (since its function call contains a `...`).

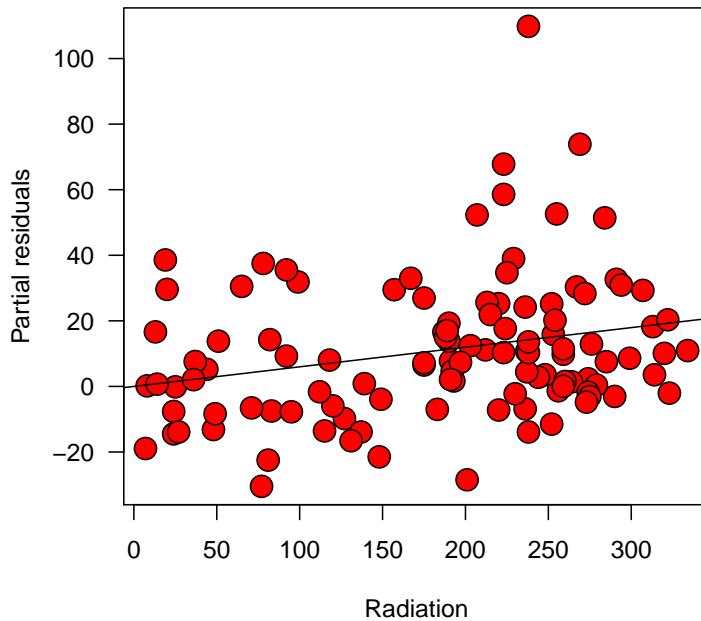
So, we can include any number of additional graphical parameters. Let's try out the complete new function.

```
prPlot(mod = mod, which.pred = "rad", col = "black", bg = "red", pch = 21, cex = 2, las = 1)
```



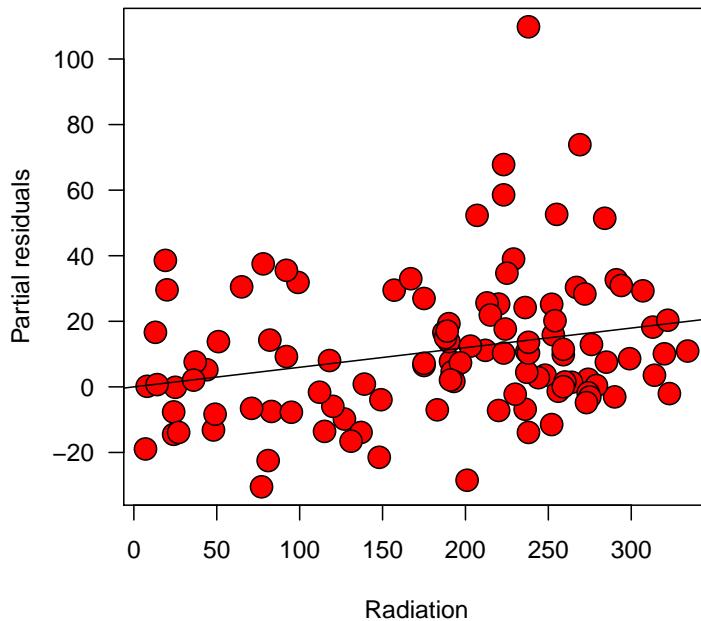
And if we wanted to change the x axis label, we could do so with this

```
prPlot(mod = mod, which.pred = "rad", xlab = "Radiation", col = "black",
       bg = "red", pch = 21, cex = 2, las = 1)
```



which is effectively the same as this

```
prPlot(mod, "rad", "Radiation", col = "black", bg = "red", pch = 21, cex = 2, las = 1)
```



The flow control structures that were discussed earlier can, of course, be used in functions, and can be used, e.g., to apply different algorithms depending on the value or class of some argument, or to stop execution and return an error message if needed. Here is a simple function that accepts a body temperature (for humans), and returns a message. It uses both `if` and `ifelse` statements, and also uses `stop` to return an error message.

```
checkTemp <- function(temp, units = "C") {  
  
  if (units == "F") {  
    temp <- (temp-32)*5/9  
  } else if (units != "C") {  
    stop("Expected C or F for units, got ", units)  
  }  
  
  cond <- ifelse(temp<35, "Hypothermia", ifelse(temp>37.5, "Fever", "OK"))  
  
  return(cond)  
}
```

Let's test it out for a few conditions.

```
checkTemp(37)  
  
# [1] "OK"  
  
checkTemp(37, units = "F")  
  
# [1] "Hypothermia"  
  
checkTemp(98.6, units = "F")  
  
# [1] "OK"  
  
checkTemp(98.6, units = "K")  
  
# Error in checkTemp(98.6, units = "K"): Expected C or F for units, got K
```

To return a warning, but not stop execution of a function, use the `warning()` function. And to have a function simply return a message, but no warning or error, use `message()` or `cat()`.

## 25 Base graphics, part II

### 25.1 Arranging multiple plots per page

There are two common ways of putting multiple plots on one page when using the base graphics functions. One way is to modify the graphic parameter `mfrow` (or `mfcol`), with the `par()` function. This is a simple method for setting the number of plot regions per page, but its one disadvantage is that all of the plot regions are equally sized. More flexibility is available through use of the `layout` function. Before I start describing each of these methods, let's discuss the `par` function, which can be used to set or check graphical parameters. With no arguments, `par` will return the current values of all graphical parameters.

```
par()

# $xlog
# [1] FALSE
#
# $ylog
# [1] FALSE
#
# $adj
# [1] 0.5
#
# $ann
# [1] TRUE
#
# $ask
# [1] FALSE
#
# $bg
# [1] "transparent"
#
# $bty
# [1] "o"
#
# $cex
# [1] 1
#
# $cex.axis
# [1] 1
#
# $cex.lab
# [1] 1
#
# $cex.main
# [1] 1.2
#
# $cex.sub
# [1] 1
#
# $cin
# [1] 0.15 0.20
```

```
#  
# $col  
# [1] "black"  
#  
# $col.axis  
# [1] "black"  
#  
# $col.lab  
# [1] "black"  
#  
# $col.main  
# [1] "black"  
#  
# $col.sub  
# [1] "black"  
#  
# $cra  
# [1] 10.8 14.4  
#  
# $crt  
# [1] 0  
#  
# $csi  
# [1] 0.2  
#  
# $cxy  
# [1] 0.03787879 0.05952381  
#  
# $din  
# [1] 5.2 5.2  
#  
# $err  
# [1] 0  
#  
# $family  
# [1] ""  
#  
# $fg  
# [1] "black"  
#  
# $fig  
# [1] 0 1 0 1  
#  
# $fin  
# [1] 5.2 5.2  
#  
# $font  
# [1] 1  
#  
# $font.axis  
# [1] 1  
#  
# $font.lab
```

```
# [1] 1
#
# $font.main
# [1] 2
#
# $font.sub
# [1] 1
#
# $lab
# [1] 5 5 7
#
# $las
# [1] 0
#
# $lend
# [1] "round"
#
# $lheight
# [1] 1
#
# $ljoin
# [1] "round"
#
# $lmitre
# [1] 10
#
# $lty
# [1] "solid"
#
# $lwd
# [1] 1
#
# $mai
# [1] 1.02 0.82 0.82 0.42
#
# $mar
# [1] 5.1 4.1 4.1 2.1
#
# $mex
# [1] 1
#
# $mfcol
# [1] 1 1
#
# $mfg
# [1] 1 1 1 1
#
# $mfrow
# [1] 1 1
#
# $mgp
# [1] 3 1 0
#
```

```

# $mkh
# [1] 0.001
#
# $new
# [1] FALSE
#
# $oma
# [1] 0 0 0 0
#
# $omd
# [1] 0 1 0 1
#
# $omi
# [1] 0 0 0 0
#
# $page
# [1] TRUE
#
# $pch
# [1] 1
#
# $pin
# [1] 3.96 3.36
#
# $plt
# [1] 0.1576923 0.9192308 0.1961538 0.8423077
#
# $ps
# [1] 12
#
# $pty
# [1] "m"
#
# $smo
# [1] 1
#
# $srt
# [1] 0
#
# $tck
# [1] NA
#
# $tcl
# [1] -0.5
#
# $usr
# [1] 0 1 0 1
#
# $xaxp
# [1] 0 1 5
#
# $xaxs
# [1] "r"

```

```

#
# $xaxt
# [1] "s"
#
# $xpd
# [1] FALSE
#
# $yaxp
# [1] 0 1 5
#
# $yaxs
# [1] "r"
#
# $yaxt
# [1] "s"
#
# $ylbias
# [1] 0.2

```

Since we have not yet modified these settings, these are the defaults. To set graphical parameters, specify the name and value of the parameter as an argument.

Note that the default setting for `mfcoll` and `mfrow` is the two element vector `1, 1`, i.e., `c(1, 1)`. This means that plots are arranged on a page in a one row-by-one column fashion, i.e., one plot per page. To create a plot layout that consists of four equal sized plot regions, `mfrow` (or `mfcoll`) would be set to `c(2, 2)`. The difference between `mfrow` and `mfcoll` is that with `mfrow`, the plots are first organized by row, and with `mfcoll`, the plots are first organized by column.

```

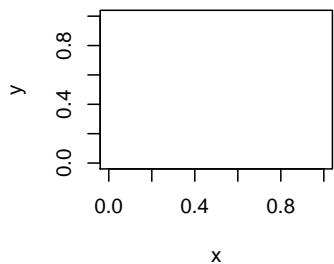
x <- 0:1
y <- 0:1

par(mfrow = c(2, 2))

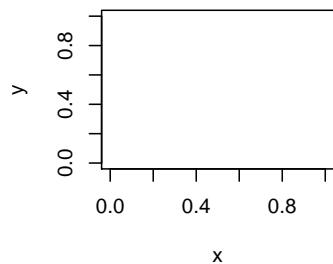
plot(x, y, type = "n", main = "Plot 1")
plot(x, y, type = "n", main = "Plot 2")
plot(x, y, type = "n", main = "Plot 3")
plot(x, y, type = "n", main = "Plot 4")

```

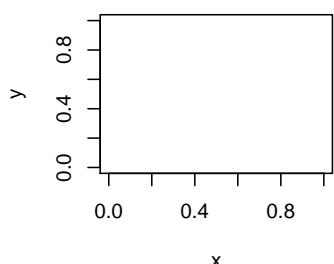
**Plot 1**



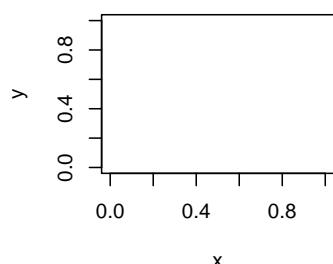
**Plot 2**



**Plot 3**



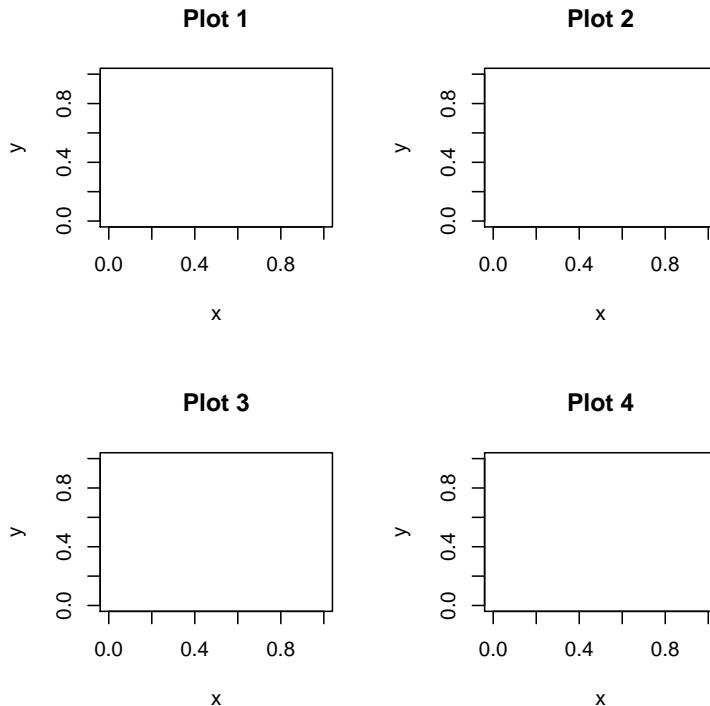
**Plot 4**



Or, with a loop:

```
par(mfrow = c(2, 2))

for (i in 1:4) plot(x, y, type = "n", main = paste("Plot", i))
```

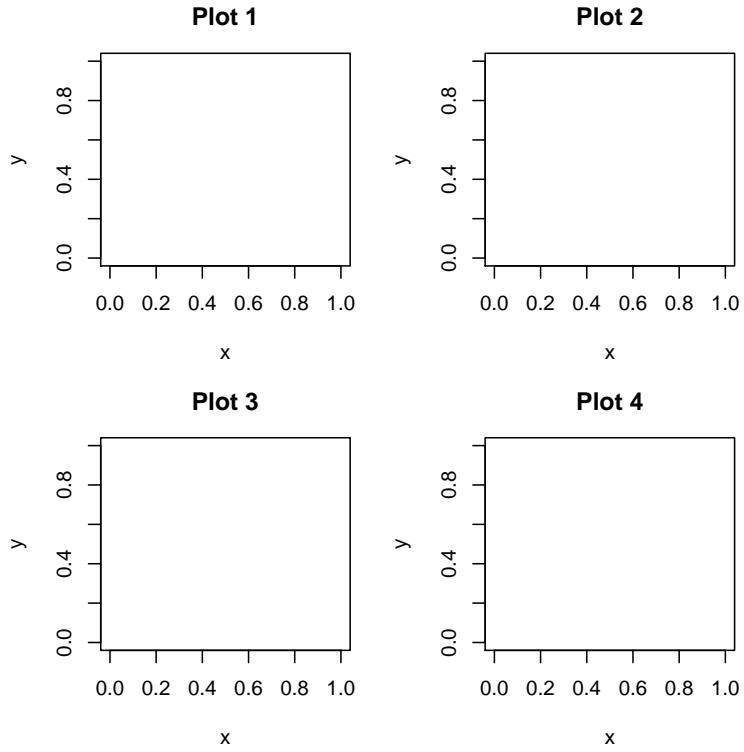


Notice the order of plots as they are arranged on the page. The plotting starts with the upper left region of the page, fills out the top row, continues to the second row, and eventually ends in the bottom right region of the page (i.e., left to right and then top to bottom). If another plot had been specified, it would have been placed in the upper left region of a new page. With `mfcol`, plot order goes from top to bottom before left to right.

There is a lot of blank space around the plots in the above example, and the plots are small. The amount of space around individual plots and around the edge of the page as a whole can be adjusted by changing the values for the parameters `mar` (margins) and `oma` (outer margins) (outer margins really only make sense when you are creating a figure in a graphics file, which is covered later). The default setting is `mar = c(5.1, 4.1, 4.1, 2.1)`, which are the margins in lines from the bottom moving clockwise. In the example below, the margins have been reduced.

```
par(mfrow = c(2, 2), mar = c(4, 4, 3, 1.5))

for (i in 1:4) plot(x, y, type = "n", main = paste("Plot", i))
```

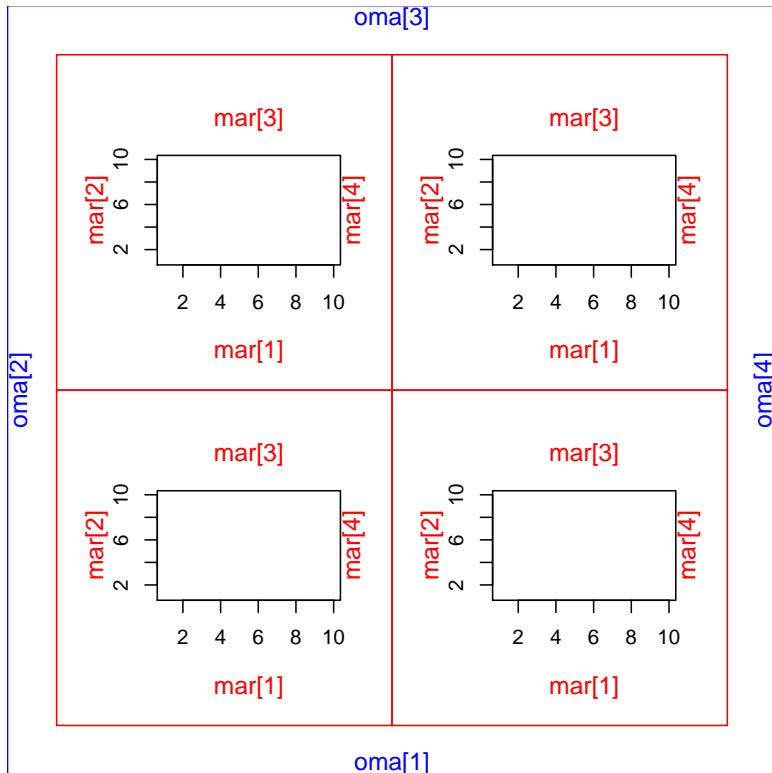


The following code displays the areas controlled by mar and oma.

```

mars <- c(5.1, 4.1, 4.1, 2.1)
omas <- c(2, 2, 2, 2)
par(mfrow = c(2, 2), mar = mars, oma = omas)
for(i in 1:4) {
  plot(NULL, xlim = c(1, 10), ylim = c(1, 10), xlab = "", ylab = "")
  box("figure", col = "red")
  for(j in 1:4) mtext(paste("mar[", i, "]"), sep = ""), side = j, line = 4-j, col = "red")
}
for(i in 1:4) mtext(paste("oma[", i, "]"), sep = ""), side = i, line = 1,
  col = "blue", outer = TRUE)
box("outer", col = "blue")

```



The layout function provides more flexibility with it, plot regions with different sizes can be specified. First, let's reset `mfrow`, `mar`, and `oma` to the defaults:

```
par(mfrow = c(1, 1), mar = c(3.5, 3.5, 3, 1)+0.1, oma = c(0, 0, 0, 0))
```

The only required argument to the layout function is `mat`, which must be a matrix that shows where each panel should go. For example, take a look at the following matrix.

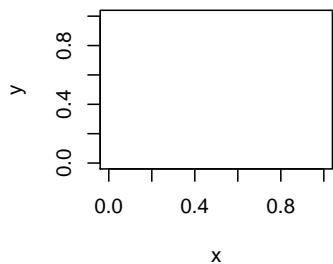
```
matrix(1:4, nrow = 2)

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

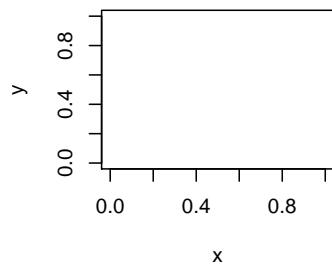
You can probably guess what kind of a layout this will give when you submit it to layout:

```
layout(matrix(1:4, nrow = 2))
for (i in 1:4) plot(x, y, type = "n", main = paste("Plot", i))
```

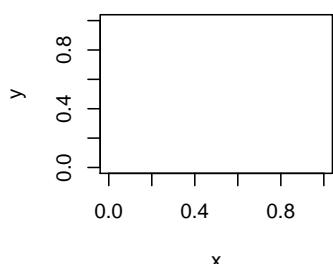
**Plot 1**



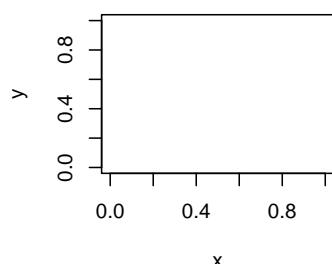
**Plot 3**



**Plot 2**

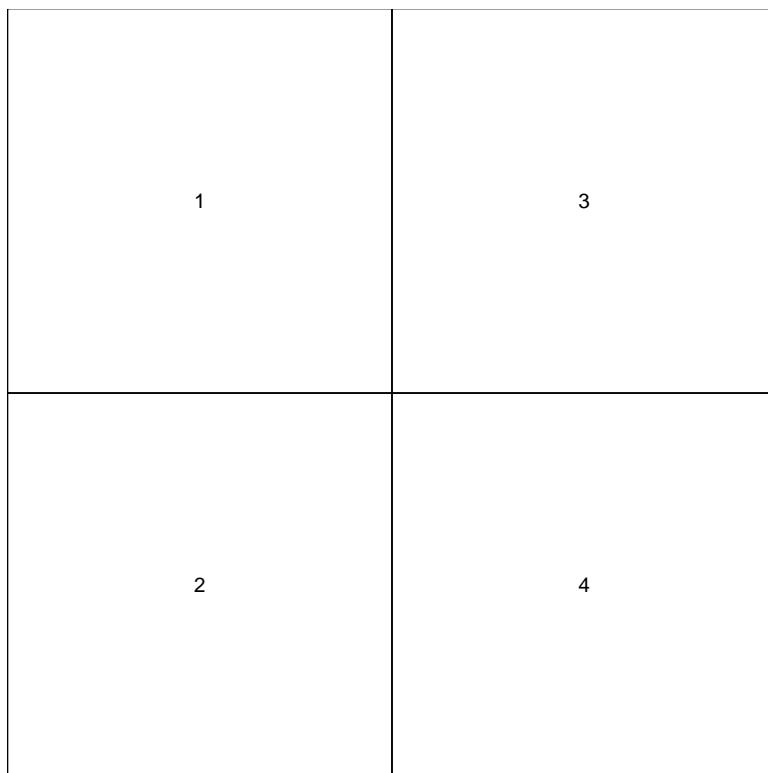


**Plot 4**



An easier way to quickly show where your plots will show up is to use the `layout.show` function:

```
layout(matrix(1:4, nrow = 2))
layout.show(4)
```



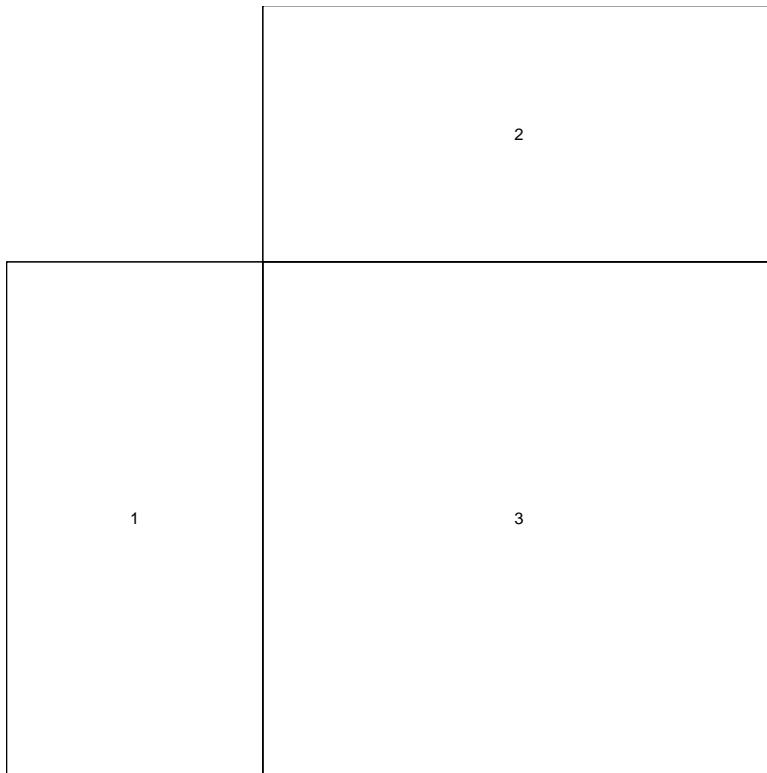
The only challenge in using layout is in figuring out a matrix that accurately reflects what you want. How about this:

```
matrix(c(0, 1, 1, 2, 3, 3, 2, 3, 3), nrow = 3)

#      [,1] [,2] [,3]
# [1,]    0    2    2
# [2,]    1    3    3
# [3,]    1    3    3
```

Note that you can just put a 0 where you dont want any plot.

```
layout(matrix(c(0, 1, 1, 2, 3, 3, 2, 3, 3), nrow = 3))
layout.show(3)
```



Additional flexibility comes from the heights argument in layout, which allows you to set the relative height of each row in your matrix. For example, compare the two cases below.

```
layout(matrix(1:9, nrow = 3, byrow = TRUE))
layout.show(9)
```

1	2	3
4	5	6
7	8	9

```
layout(matrix(1:9, nrow = 3, byrow = TRUE), heights = c(0.5, 2,1))
layout.show(9)
```

1	2	3
4	5	6
7	8	9

Once you get the hang of using layout, it is very easy to manipulate the number, arrangement, and

size of plots in graphical output. However, if multiple plots of equal size are to be created, specifying `mfrow` (or `mfcol`) is an easier option.

## 25.2 More on the plot function: arguments and values

R allows the user to have fine control over essentially all aspects of graphical output. Some of the most common optional arguments to the `plot` function are given in the table below, along with common values. These arguments primarily allow users to control axis limits and appearance.

Argument	Common values	Notes
<code>cex</code>	0 to 3 (or greater)	Numeric value. Size of plotting text and symbols.
<code>mgp</code>	<code>c(0, 0, 0)</code> through <code>c(3, 2, 1)</code>	Location of axis titles, labels, and lines. Default if <code>c(3, 1, 0)</code>
<code>xaxs</code>	" <code>r</code> " or " <code>i</code> "	Style of axis interval. Default (" <code>r</code> ") extends a bit beyond extreme values.

The following code and resulting graphics demonstrate some of the options listed in the table above. (We will cover the `text` function, which is used below to annotate the plots, in the next subsection.)

```
par(mfrow = c(2, 2), mar = c(4.5, 4.5, 2, 1))

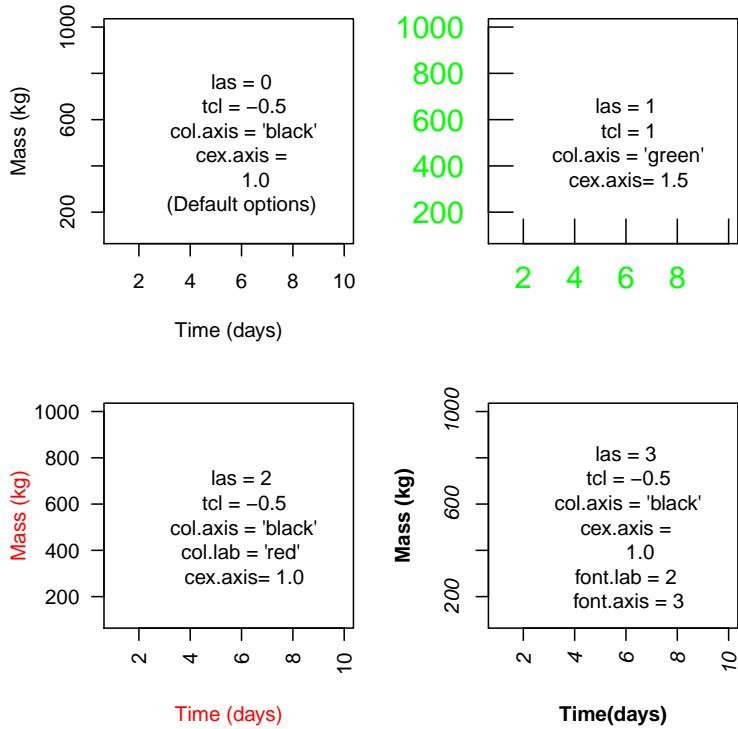
x <- 1:10
y <- 100*x

plot(x, y, type = "n", xlab = "Time (days)", ylab = "Mass (kg)")
text(6, 500, "las = 0\n tcl = -0.5\n col.axis = 'black'\n cex.axis =
1.0\n(Default options)")

plot(x, y, type = "n", xlab = "", ylab = "", las = 1, tcl = 1, col.axis = "green",
cex.axis = 1.5)
text(6, 500, "las = 1\n tcl = 1\n col.axis = 'green'\n cex.axis= 1.5")

plot(x, y, type = "n", xlab = "Time (days)", ylab = "Mass (kg)", las = 2, col.lab = "red")
text(6, 500, "las = 2\n tcl = -0.5\n col.axis = 'black'\n col.lab = 'red' \n cex.axis= 1.0")

plot(x, y, type = "n", xlab = "Time(days)", ylab = "Mass (kg)", las = 3, font.lab = 2, font.axis = 3)
text(6, 500, "las = 3\n tcl = -0.5\n col.axis = 'black'\n cex.axis =
1.0\n font.lab = 2 \n font.axis = 3")
```



A complete description of the various plot parameter settings that can be specified as arguments to plot can be viewed in the help file for `par`. This will provide a list of the plot parameter settings that can either be set by specifying values in a `par` command, or by specifying values as arguments through the `plot` function. Note that any parameters that are specified in `par` will remain set at the specified value until another value is specified. For example, future plots will be displayed based on the specified settings. If settings are set in a `plot` command, or `axis`, etc., they are only set for that individual plot.

## 26 Dynamic documents

“Dynamic documents” contain components that depend on operations in R. They can be automatically updated using R. This book is an example. It is written as a plain text file with a mix of text (like this sentence you are reading) and R code (like the call below).

```
rnorm(10)
```

The output from the R code is generated and combined with the text in an intermediate file, which is then “compiled” to make the final pdf. You can also see this output below.

```
rnorm(10)
```

```
# [1] 0.06614578 0.45955474 2.02972284 -0.85729121 0.11499186
# [6] 0.02906988 0.27872294 -1.03097236 2.78295703 -0.45573274
```

When I change a data file or some code used in some examples I can quickly update the entire book.

Depending on what you want to do and your preferences, dynamic documents can be made using the `sweave()` function from the `utils` package (one of the base packages), functions from the `knitr` package, or functions from the `rmarkdown` package. For typesetting, in this book I use L<sup>A</sup>T<sub>E</sub>X, which is feature-rich but not the easiest markup language to learn. For this approach, I work in an “Rnw” file<sup>155</sup>, which is a text file that has a mixed of L<sup>A</sup>T<sub>E</sub>X commands, descriptive text, and “chunks” of R code, identified by special delimiters.

Why use a dynamic document instead of a simple R script? First, it is a convenient way to combine notes or descriptive text, R code, and output in a single document. They provide all important parts of data analysis in a single document, which facilitates interpreting results and eliminates some mistakes (operations and output are linked). Second, the dynamic bit means manual steps are bypassed and re-analysis or re-evaluation of your results after some change is both easy and repeatable. Try to do this with manually copying and pasting, followed by some manipulations in Microsoft Excel, and manual creation of a table in Microsoft Word.

What I and many R users recommend is not Rnw but Rmd files, that is, using R Markdown. Markdown is a markup language that can be compiled to pdf, html, and even Microsoft Word files<sup>156</sup>. Why recommend it? It is simple to learn, and can be easily understood in its original form (the Markdown file).

R Markdown can be learned in a few minutes. In RStudio, you can use the **File** menu to create a new R Markdown file, and go from there. Or, to start manually, just create a new text file, add a YAML header as in the example below, and save it with the Rmd extension.

```
---
title: "Homework assignment 3"
author: "Sasha D. Hafner"
date: "July 7, 2019"
output: html_document
---
```

<sup>155</sup> RNW stands for “R Noweb”, where “noweb” is a reference to the “noweb” literate programming tool developed by Norman Ramsey, which is based on the WEB system developed by Donald E. Knuth.

<sup>156</sup> Yihui Xie, the author of the knitr package, wrote in this “a Word document? No! but my boss wants it... okay...”.

In Markdown (and therefore, R Markdown), the `#` symbol is used for headers.

```
# Overview  
This report generates and plots 10 random numbers.
```

You can find more details on Markdown from RStudio and elsewhere. A “cheat sheet” from RStudio is a good place to start: <https://www.rstudio.com/wp-content/uploads/2016/03/rmarkdown-cheatsheet-2.0.pdf>. A few tips are given below.

Markdown syntax	Description
<code>#</code> Header	Header 1
<code>##</code> Sub-header	Header 2
<code>###</code> Sub-sub-header	Header 3
<code>*bold*</code>	Bold text
<code>*italic*</code>	Italicized text
<code>x^2^</code>	Superscript
<code>CO~2~</code>	Subscript
<code>\$y = x^2 \$</code>	In-line math

R code is placed between the delimiters shown below.

```
```{r}  
x <- rnorm(10)  
```  
  
```{r}  
plot(x)  
```
```

The code you enter and the associated delimiters is called a “code chunk”. Most editors for R scripts include a shortcut for adding the delimiters. In RStudio it is **ctrl + alt + i**.

Exactly how code chunks are handled can be specified in “chunk options”. The options `echo` (to display the code) and `eval` (to evaluate the code and show the result) are two of the most useful. Both default to `TRUE`, but could be set to `FALSE` using the code shown below.

```
```{r, echo = FALSE}  
x <- rnorm(10)  
x  
```  
  
```{r, eval = FALSE}  
x <- 10  
```  
  
```{r}  
x  
```
```

Do you know what value `x` would have in the third chunk? You can find more chunk options in the cheat sheet mentioned above (<https://www.rstudio.com/wp-content/uploads/2016/03/rmarkdown-cheatsheet-2.0.pdf>).

To more really integrate R output into your text, you can use single backtick quotes for inline code. For example:

```
```{r}
x <- rnorm(10)
```

The mean of these values is `r mean(x)`.
```

This would give you something like this:

“The mean of these values is 0.15391”

in the resulting document. Remember to start with `r` here, otherwise you will get unevaluated code in the document. This may also be useful, e.g.,

The call used above was '`rnorm(10)`'.

How does this all fit together? See the block below for an example.

```
---
title: "Example dynamic report with some random numbers"
author: "Sasha D. Hafner"
date: "July 13, 2019"
output: html_document
---


# Overview
This report is an example of a dynamic report based on R Markdown.
In it, random numbers are generated and used in a few operations.

# The actual code
First we will generate some random numbers.

```{r}
x <- rnorm(10)
```

These can be plotted with the `plot()` function.

```{r}
plot(x)
```

The mean value is `r mean(x)`.
```

## 27 Common mistakes

In order to use R, it is necessary to write commands. While very flexible and generally more efficient than “selection and response” software, this approach means that an infinite number of mistakes can be made. This section lists some of the more common mistakes.

### 1. Forgetting that R is case sensitive.

```
VEC <- 1:10
vec

# Error in eval(expr, envir, enclos): object 'vec' not found
```

### 2. Trying to change the value of a variable without using assignment.

```
x <- 1:10
x

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

Can multiply by 10:

```
x*10

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

But this doesn't change the value of x. See:

```
x

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

This does:

```
x <- x*10
x

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

### 3. Expecting R to update commands without resubmitting them.

```
x <- 1:10
y <- x*100
y

# [1] 100 200 300 400 500 600 700 800 900 1000
```

```
x <- rnorm(10)
y

# [1] 100 200 300 400 500 600 700 800 900 1000
```

Why didn't y change?

#### 4. Trying to open a file that doesn't exist (or isn't in working directory).

```
dat <- read.csv("../data/typo.csv")

# Warning in file(file, "rt"): cannot open file '../data/typo.csv': No such file or
# directory

# Error in file(file, "rt"): cannot open the connection
```

Check directory

```
getwd()
```

```
# [1] "C:/Workshop files"
```

And list contents

```
list.files()
```

```
[1] "ave_wind_us.csv"          "biohydrogen_bad.csv"
[3] "biohydrogen.csv"           "biohydrogen_v1.csv"
[5] "biohydrogen_v2.csv"         "biohydrogen.xlsx"
...
.
```

#### 5. Using backward slashes instead of forward slashes in path.

```
#dat <- read.csv("C:\Workshop files\us_pop.csv")
```

```
Error: '\u' used without hex digits in character string starting "C:\u"
```

#### 6. Forget header = TRUE argument in a read.table expression.

```
dat <- read.table("../data/us_pop.txt")
dat
```

```
#      V1          V2
#1  year        pop
#2  1790    3929214
#3  1800    5308483
#4  1810    7239881
#5  1820    9638453
#....
```

```
summary(dat)
```

```
#      V1          V2
# 1790 : 1  106021537: 1
# 1800 : 1  123202624: 1
# 1810 : 1  12866020 : 1
# 1820 : 1  132164569: 1
# 1830 : 1  151325798: 1
# 1840 : 1  17069453 : 1
# (Other):17 (Other) :17
```

## 6. Not noticing extra whitespace in a file.

File contents:

```
reactor date time vol conc.h2
G171 9/18/2006 11:12 0 NA
G171 9/18/2006 14:00 0 31.200
G171 9/19/2006 9:26 11.35 35.220
G 171 9/19/2006 12:51 0 NA
....
```

The problem is in the last entry in the first column: G 171.

```
dat <- read.csv("../data/biohydrogen_bad.csv")
```

```
#Error in scan(file, what, nmax, sep, dec, quote, skip, nlines, na.strings, :
# line 1 did not have 6 elements
```

## 7. Presence of extra characters in numeric data.

File contents:

```
site achlor
'magee Marsh' 1.25
'Magee Marsh' 1.55
'Magee Marsh' 1.74
'Magee Marsh' 1.7.7           Problem is here
'Davis Besse-Toussaint' 1.82
....
```

```
dat <- read.csv("../data/eagles_bad.csv")
mean(dat$achlor)

#[1] NA
#Warning message:
#In mean.default(dat$achlor) :
#  argument is not numeric or logical: returning NA

class(dat$achlor)
```

#[1] "factor"

**8. Not specifying a data frame when it is needed, or using a data argument for functions that don't have one.**

```
dat <- read.csv("../data/eagles.csv")
plot(achlor, data = dat)
```

```
# Error in plot(achlor, data = dat) : object 'achlor' not found
```

All of the options below are fine.

```
plot(dat$achlor)
with(dat, plot(achlor))
plot(achlor ~ site, data = dat)
```

**9. Trying to load a package that hasn't been installed.**

```
library(beeswarm)
```

**10. Using the wrong argument name in a function that accepts optional arguments.**

Because the function accepts optional arguments, you won't get an error or warning.

```
library(reshape2)
mites <- read.csv("../data/mites.csv")
mites.m <- melt(mites, id.variables = 1:4)

# Using date, treat as id variables
```

But I told it to use four variables. What it did instead was make guesses. Why? Correct argument name used below.

```
mites.m <- melt(mites, id.vars = 1:4)
```

## **11. Not paying attention to error and warning messages.**

When R gives you a message, read it. They are short but usually make sense.

## **12. Not reading help files.**

You can find a lot of useful information in help files, both on the function(s) itself and on related functions.

## 28 Where to go next

If you have made it to this point of the book, and are still interested in R, this section will provide you with a bit of advice of how to proceed. If you do not feel comfortable with most of the material in the book, I recommend reviewing the book, and working through examples and exercises until you do. Most of the topics covered in the book will be useful for a wide range of data analysis and graphics tasks, and should give you a good start in R.

If you found this book lacking, first, please let me know what you don't like (send an email to me), and then check out some alternatives. A good introductory book on R and basic statistical analyses in R is Peter Dalgaard's *Introductory Statistics with R* [4]. CRAN also has free documents that cover several R topics. Start at the list of manuals at <http://cran.r-project.org/manuals.html>, where you can download *An Introduction to R* by Bill Venables et al. [25], which provides a good introduction. Other documents can be found at <http://cran.r-project.org/other-docs.html>.

If you are comfortable with the material in this book, and want to learn more about specific topics, there are many books that can be helpful. CRAN lists 160 books on R: <http://www.r-project.org/doc/bib/R-books.html>, but this list is not complete, as a search for R in Amazon will show. Also, books on statistical models and not on R *per se* often include R code through companion websites. Unfortunately, not all books on R are *good* books on R, so look for book reviews or find a copy to check out before buying.

I've listed some of my favorite R books below, organized by topics. You can find the complete citations for these books in the References section, below. All the R manuals can be downloaded for free from <http://cran.r-project.org/manuals.html>.

### R basics

Dalgaard 2008<sup>157</sup> [4]: Chapters 1 & 2

Manuals: An Introduction to R [25]: Sections 1 & 2

### Vectors, matrices, and arrays

Dalgaard 2008 [4]: Chapter 1

Chambers 2008 [3]: Chapter 6

Manuals: An Introduction to R [25]: Sections 2 & 5

Manuals: The R Language Definition [18]: Section 2

### Data import and data export

Manuals: R Data Import/Export [17]

Manuals: An Introduction to R [25]: Section 7

Dalgaard 2008 [4]: Chapter 1

Spector 2008 [21]: Chapters 2 & 3

### Indexing

Manuals: An Introduction to R [25]: Sections 2 & 5

Spector 2008 [21]: Chapter 6

### Data manipulation

Spector 2008 [21]

<sup>157</sup> This and some of the other books in this list are published by Springer as part of its Use R! series. Many university libraries have a subscription to Springer Link, which means students and employees may be able to download pdf versions of the Use R! books, as well as buy black-and-white paperback versions for a reduced price (\$25 as of early 2012). If you have access to a university library system, check it out.

## **Programming**

Chambers 2008 [3]

Venables & Ripley 2000 [23]

## **Graphics**

Murrell 2005 [11]

Wickham 2009 [26]

Sarkar 2008 [20]

## **Linear models**

Faraway 2005 [6]

Faraway 2002 [5]

Venables and Ripley 2002 [24] Zar 1999 [29]

## **Generalized linear models**

Faraway 2006 [7]

Venables and Ripley 2002 [24]

## **Random- and mixed-effects models**

Pinheiro and Bates 2000 [13]

Bates 2005 [2]

## **Generalized additive models**

Wood 2006<sup>158</sup> [28]

## **Nonlinear regression**

Ritz & Streibig 2008 [15]

Pinheiro and Bates 2000 [13]

## **Nonparametric models**

Faraway 2006 [7]

## **tidyverse packages**

See the tidyverse website (<http://tidyverse.org/>)

Grolemund et al 2017 [8] (You can find a free version online at <http://r4ds.had.co.nz/>)

## **Acknowledgements**

This book developed from an earlier version that I originally wrote with Adam Ryan (HDR | HydroQual, <http://www.hydroqual.com/>). Adam and I learned R together when we both worked for HydroQual, and I think he came up with the idea of offering R workshops. We wrote our original book in 2007 and 2008 for workshops at Clemson University and SUNY ESF. James Gibbs (SUNY ESF) kindly provided several data sets that are used in this workshop, and also provided advice on organizing workshops which has proved to be very helpful. Other individuals have shared data as well: Ben Hirsch, Elizabeth Jordan Reverri, Amy Roe, Mariam Lekveishvili, and Joe Besessi. Lastly, feedback from participants in the previous workshops and courses has been very helpful in improving the selection, organization, and delivery of material.

---

<sup>158</sup> I haven't read this book, but it looks good.

## 29 References

- [1] Agresti, A. 2007. *An Introduction to Categorical Data Analysis*. New York: Wiley.
- [2] Bates, D.M. 2005. Fitting linear mixed models in R. *R News* 5(1): 27-30.
- [3] Chambers, J. 2008. *Software for Data Analysis: Programming with R*. New York: Springer.
- [4] Dalgaard, Peter. 2008. *Introductory Statistics with R*. 2nd ed. New York: Springer.
- [5] Faraway, J. 2002. *Practical Regression and ANOVA using R*. <http://cran.r-project.org/other-docs.html>.
- [6] Faraway, J. 2005. *Linear Models with R*. New York: Chapman and Hall/CRC.
- [7] Faraway, J. 2006. *Extending the Linear Model with R: Generalized Linear, Mixed Effects and Nonparametric Regression Models*. New York: Chapman and Hall/CRC.
- [8] Grolemund, G., Wickham, H. 2017. R for Data Science. Sebastopol, CA: O'Reilly. <http://r4ds.had.co.nz/>
- [9] Hafner, S.D. 2007. Biological hydrogen production from nitrogen-deficient substrates. *Biotechnology and Bioengineering* 97: 435-437.
- [10] McCullagh, P., Nelder, J.A. 2006. *Generalized Linear Models*. 2nd ed. New York: Chapman and Hall/CRC.
- [11] Murrell, P. 2005. *R Graphics*. London: CRC Press.
- [12] NOAA 2009. Comparative Climate Data, Tables. <http://ols.nndc.noaa.gov/plolstore/plsql/olstore.prodspecific?prodnum=C00095-PUB-A0001#TABLES>
- [13] Pinheiro, J.C., Bates, D.M. 2000. *Mixed-effects models in S and S-PLUS*. New York: Springer.
- [14] Qui, X., R. Hites. 2008. Dechlorane plus and other flame retardants in tree bark from the northeastern United States. *Environmental Science and Technology* 42: 31-36.
- [15] Ritz, C., Streibig, J. 2008. *Nonlinear Regression with R*. New York: Springer.
- [16] Schimel J , Nadelhoffer K , Shaver G , Giblin A , and Rastetter E. 1995. Methane and carbon dioxide emissions were monitored in control, greenhouse, and nitrogen and phosphorus fertilized plots of three different plant communities, Toolik Field Station, North Slope Alaska, Arctic LTER 1993. <https://metacat.lternet.edu:443/knb/metacat/knb-lter-arc.1427.4/lter>
- [17] R Development Core Team. 2008. *R Data Import/Export*.
- [18] R Development Core Team. 2008. *R Language Definition*.
- [19] Ryan A.C. 2005. *Influence of Dissolved Organic Matter Source and Chemical Characteristics on Acute Copper Toxicity*. PhD Dissertation. Clemson University, Clemson, South Carolina, USA.
- [20] Sarkar, D. 2008. *Lattice: Multivariate Data Visualization with R*. New York: Springer.
- [21] Spector, Phil. 2008. *Data Manipulation with R*. New York: Springer.
- [22] Thakali, S., H.E. Allen, D.M. Di Toro, A.A. Ponizovsky, C.P. Rooney, F.J. Zhao, and S.P. McGrath. 2006. A terrestrial biotic ligand model. 1. Development and application of Cu and Ni toxicities to barley root elongation in soils. *Environmental Science and Technology* 40: 7085-7093.
- [23] Venables, W.N., Ripley, B.D. 2000. *S Programming*. New York: Springer.
- [24] Venables, W.N., Ripley, B.D. 2002. *Modern Applied Statistics with S*. New York: Springer.

- [25] Venables, W. N., D. M. Smith, and the R Development Core Team. 2008. *An Introduction to R*.
- [26] Wickham, H. 2009. *ggplot2: Elegant Graphics for Data Analysis*. New York: Springer.
- [27] Wilcock, R. J., C. D. Stevenson, and C. A. Roberts. 1981. An interlaboratory study of dissolved oxygen in water. *Water Research* 15: 321-325.
- [28] Wood, S. 2006. Generalized Additive Models: An Introduction with R. Boca Raton: Chapman & Hall.
- [29] Zar, Jerrold H. 1999. *Biostatistical Analysis*. 4th ed. Upper Saddle River, NJ: Prentice Hall.