# An Introduction to Data Analysis and Graphics with R

A workbook for the 2010 University of Vermont R workshop

**Sasha Hafner and Adam Ryan**

**September 2010**

For more information on this workshop, please contact Sasha Hafner at sasha.hafner@dataanalysisworkshops.com.

# Contents

## Objective

The objective of this workshop is to introduce participants to data analysis and graphics with R. The range of analysis that can be completed, and the types of graphics that can be created in R are astounding. In addition to the wide variety of functions available in the "base" packages that are installed with R, more than 2400 contributed packages are available for download, each with its own suite of functions. Some of the individual packages are the subject of entire books. Obviously, this workshop will not cover every type of analysis or plot that R can be used for, or even every subtlety associated with each function covered in this workshop. However, after completing this workshop, you should be comfortable with the basic tools for carrying out typical data analyses and generating publication- and presentation-quality graphics in R. Given the inherent flexibility of R and of the functions that are covered in this workshop, we hope these basic tools will go a long way toward meeting your data analysis and data presentation needs. Furthermore, the experience that you gain in this workshop should give you a familiarity with the use of R, the Comprehensive R Network Archive (CRAN) site, and the R language, all of which will facilitate acquisition and use of other packages for specialized data analysis. Lastly, the brief introduction to some more advanced topics, such as writing functions and batch processing, can serve as a starting point for the development of time-saving procedures for data analysis and presentation. We hope you continue to use and learn R. Considering the number of people that are using and contributing to R, the number of books and other documents dedicated to R, and R's inherent flexibility, the data analysis and graphics development possiblities seem endless.

# 1. Introduction to R

*Crawley 2007: Chapter 1; Dalgaard 1997: Chapter 1; R-Intro: Sections 1 & 2, R-Lang: Section 2*

## *1.1. R overview and history*

R is a software system for computations and graphics. According to the R FAQ (http://cran.r-project.org/doc/FAQ/R-FAQ.html#R-Basics), "[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 R. Ihaka and R. Gentleman at the University of Auckland (New Zealand). The R language is a "dialect" of the S language[1], which was developed (principally) by J. Chambers at Bell Laboratories. This software is currently maintained by the R Development Core Team, which consists of more than a dozen people, and includes Ihaka, Gentleman, and Chambers. Additionally, many other people have contributed code to R since it was first released. The source code for R is available under the GNU General Public License, meaning that users can modify, copy, and redistribute the software or derivatives, as long as the modified source code is made available. R is regularly updated, however, changes are usually not major.

## *1.2. Finding and installing R*

R is available for Windows, Mac, and Linux operating systems. Installation files and instructions can be downloaded from the Comprehensive R Archive Network (CRAN) site at http://cran.r-project.org/. Although the graphical user interface (GUI) differs slightly across systems, the R commands do not.

## *1.3. Running R: GUI & scripts*

There are two basic ways to use R on your machine: through the GUI, where R evaluates your code and returns results as you work, or by writing, saving, and then running R script files[2]. R script files (or scripts) are just text files that contain the same types of R commands that you can submit to the GUI. Scripts can be submitted to R using the Windows command prompt, other shells, batch files, or the R GUI. All the code covered in this workbook will work if directy typed into the GUI, or it can be saved in a script file which can then be submitted to R[3]. Working directly in the R GUI is great for the early stages of code development, where a lot of experimentation and trial-and-error occurs. For any code that you want to save, rerun, and modify, you should consider working with R scripts. A useful approach is to work with both simultaneously—testing and perfecting code in the GUI before saving it in a script file.

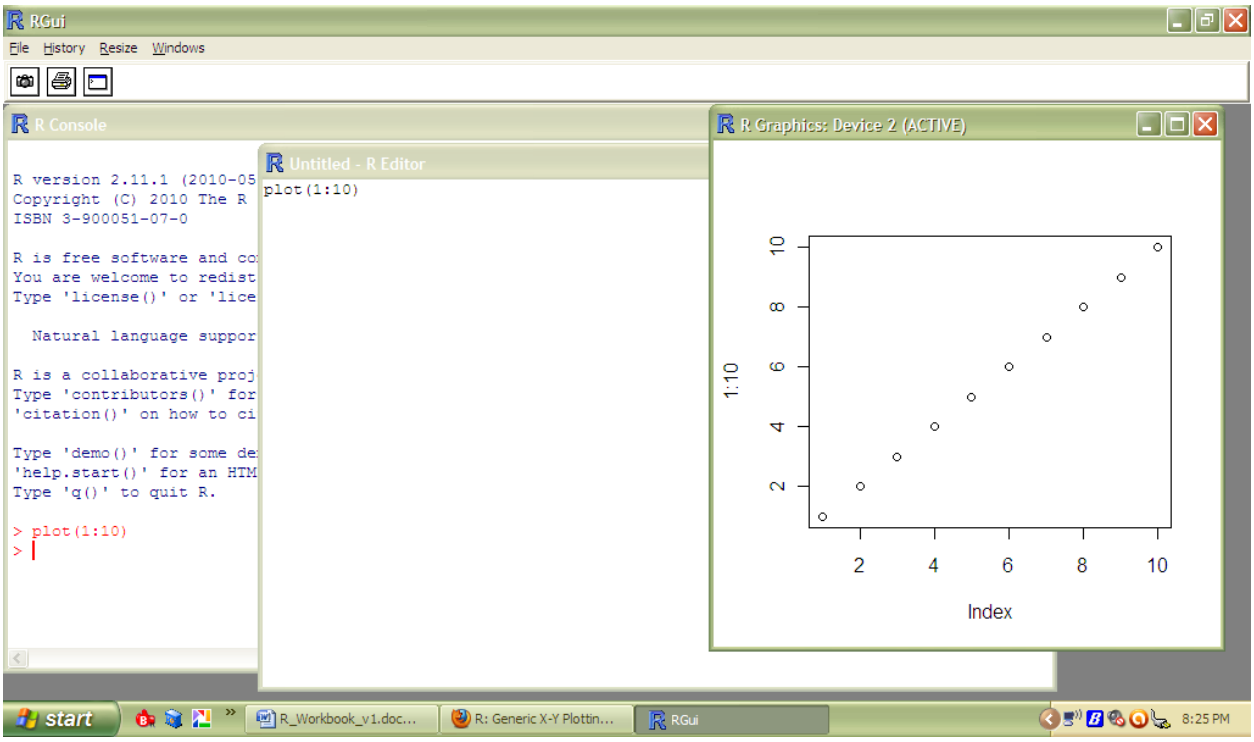
---

[1] The S language is also used in the commercial software S-PLUS, which is very similar to R.

[2] Note that the R GUI is command-line driven. 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, we recommend you stick to writing R commands yourself. The developer of R Commander, John Fox, wrote: "I must confess that I'm not terribly enamored of menu/dialog box interfaces to statistical software. . . One of my design goals was to wean users from the GUI to writing commands. . ."

[3] 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`.

So, how do you work with scripts? Any simple text editor works—you just need to be able to save text in the ASCII format, i.e. "unformatted" text. You can save your scripts and either call them up using the command `source("file_name.R")` in the R GUI, or, if you are using a shell (e.g. Windows command prompt) then type `R CMD BATCH file_name.R`[4]. The Windows and Mac versions of the R GUI come with a pretty basic script editor, shown below (the window in the center). This editor allows you to edit and create scripts, and also submit commands with the click of a button.




Unforunately, this editor isn't very good—the Windows version does not even have syntax highlighting[5].

There are some useful (in some cases free) text editors available that can be set up with R syntax highlighting and other features. TINN-R is a free text editor that is designed specifically for working with R script files[6]. Notepad++ is a general purpose text editor, but includes syntax highlighting and the ability to send code directly to R with the NppToR plugin[7]. For serious code

[4] To execute R scripts in batch mode, Windows needs to know where to find the R executable—you can do this by adding the file location to the Environment variable.
[5] The Mac version does have syntax highlighting, as well as some other nice features.
[6] You can find more information and download the program here: http://www.sciviews.org/Tinn-R/. Note that an old version of TINN-R (1.17.2.4) is more flexible (and easier to set up) than newer versions, but we encourage you to check out newer versions. TINN-R 1.17.2.4 seems to be buggy in Windows Vista.
[7] Notepad++ can be downloaded here: http://notepad-plus-plus.org/download. NppToR is available here: http://sourceforge.net/projects/npptor/.

writing, you might want to check out Emacs[8], which is a powerful text editor that becomes a tailor-made R code editor when you add the ESS plugin. A list of text editors that work well with R can be found here: http://www.sciviews.org/_rgui/projects/Editors.html.

## *1.4. R basics: commands, expressions, assignments, operators, objects*

The instructions you give R are called commands. The basic approach to using the R GUI is to type a command and hit enter—R evaluates what you typed and prints the result.

```
> 1+1
[1] 2
```

Notice a couple things about the above code. The > character is the prompt that will always be present in the GUI—it is used throughout this workbook to show which lines are commands (although where code gets complicated later on in this workbook, we leave out the prompt character and use different fonts for commands and R output). The line following the command starts with a `[1]`, which is simply the position of the adjacent element in the output—this will make more sense later.

For the above command, the result is printed to the screen and lost—there is no assignment involved[9]. 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 `<-`. When a command is used for assignment, the result is no longer printed to the GUI console[10].

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

Note that this is very different from:

```
> x< -1+1
[1] FALSE
```

In this case, putting a space between the two characters that make up the assignment operator causes R to interpret the command as an expression that asks if `x` is less than zero. However, spaces usually do not matter in R, as long as they do not separate a single operator or a variable name. This, for example, is fine:

---

[8] You can find more information, and a download here: http://vgoulet.act.ulaval.ca/en/ressources/emacs/. Or, for the more typical do-it-yourself approach, here: http://ess.r-project.org/index.php?Section=download. Note that there is a steeper learning curve for Emacs than for the other programs mentioned above.

[9] 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 screen with `.Last.value`.

[10] Unless you surround the entire command in parentheses.

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

Note that you can recall a previous command in the R GUI by hitting the up arrow on your keyboard. 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 R's 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: object "x" not found
```

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[11].

If you want to assign the same value to several symbolic variables, you can use the following syntax.

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

R is a case-sensitive language. This is true for symbolic variable names, function names, and everything else in R.

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

> X
Error: object "X" not found
```

In R, commands can be separated by moving onto a new line (i.e. hitting enter) or by typing a semicolon (`;`), which can be handy in scripts for condensing code. If a command is not completed in one line (by design or error), the typical R prompt > is replaced with a +.

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

[11] But, some serious R users stick with = instead of <− (e.g., Spector 2008).

If you find that you get stuck in a bad command, just hit the Esc key to get back to the regular prompt. For most of the commands in this workbook, we will include the prompt character > on the first line, and the continuation character + on following lines. If you want to select and copy a multi-line command from this workbook, you should be able to avoid copying the > and + characters by holding down the Alt key while you select.

There are several operators that are used in the R language. Some of the most common are listed below (more on these later):

**Arithmetic:**

| | |
|---|---|
| `+ - * / ^` | plus, minus, multiply, divide, power |

**Relational:**

| | |
|---|---|
| `a==b` | a is equal to b (do not confuse with =) |
| `a!=b` | a is not equal to b (the `!` symbol can be used to negate relational expressions in general) |
| `a<b` | a is less than b |
| `a>b` | a is greater than b |
| `a<=b` | a is less than or equal to b |
| `a>=b` | a is greater than or equal to b |

**Logical/grouping:**

| | |
|---|---|
| `!` | not |
| `&` | and |
| `\|` | or |

**Indexing**

| | |
|---|---|
| `$` | part of a data frame |
| `[]` | part of a data frame, array, list |
| `[[]]` | part of a list |

**Grouping commands**

`{}`

**Making sequences**

| | |
|---|---|
| `a:b` | returns the sequence a, a + 1, a + 2, . . . b |

**Others**

| | |
|---|---|
| `#` | commenting |
| `;` | alternative for separating commands |
| ~ | model formula specification |
| `()` | order of operations, function arguments |

Commands in R operate on objects, which can be thought of as anything that can be assigned to a symbolic variable. Objects include vectors, matrices, factors, lists, data frames, and functions. Excluding functions, these objects are also referred to as data structures or data objects.

When you close the R GUI, 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. It is good practice to not rely on a saved workspace. Instead, you should save the commands that created it as a script file, or save your output as a text file. One handy feature of the R GUI is the "Save History" option, which can be found in the File menu. This allows you to save all the commands you have submitted to the R GUI during your session as a script file.

## *1.5. R data types*

The term "data type" in R refers to the type of data that is present in a data structure, and does not describe the data structure itself. There are four common types of data in R: numerical, character, logical, and complex numbers. These are referred to as *modes* in R and are shown below.

Numerical data

```
> x<-10.2
> x
[1] 10.2
```

Character data

```
> name<-"John Smith"
> name
[1] "John Smith"
```

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

```
> name<-John
Error: object "John" not found
```

Either single or double quotes can be used in R (double quotes are used in this workbook, to avoid confusing single quotes with the accent character (`). When character data are read into R from a file, the quotes are not necessary[12].

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

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

[12] Unless spaces are present in individual elements, although even here quotes can be avoided by specifying a separator other than a space.

Note that there are no quotes around logical values—this 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
```

And finally, complex numbers, which will not be covered in this workbook, are the final data type in R.

```
> cnum1<-10 + 3i
> cnum1
[1] 10+3i
```

You can use the `mode` or `class` function to see what type of data is stored in any symbolic variable[13].

```
> class(name)
[1] "character"

> class(a)
[1] "logical"

> class(x)
[1] "numeric"

> mode(x)
[1] "numeric"
```

## *1.6. R data structures*

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. Data structures in R include vectors, matrices, arrays, data frames, lists, and factors. We will demonstrate how to make these different data structures in a following section; the examples below simply give you an idea of their structure.

[13] Mode and class are not identical—as Bill Venables writes: "'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).

Vectors are perhaps the most important type of data structure in R. A vector is simply an ordered collection of elements (e.g. individual numbers).

```
> x<-1:10
> x
 [1]  1  2  3  4  5  6  7  8  9 10
```

Matrices are similar to vectors, but have two dimensions.

```
> X<-matrix(1:30,nrow=3)
> X
     [,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 2 dimensions.

```
> Y<-array(1:90,dim=c(3,10,3))
> Y
, , 1

     [,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

, , 2

     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]   31   34   37   40   43   46   49   52   55    58
[2,]   32   35   38   41   44   47   50   53   56    59
[3,]   33   36   39   42   45   48   51   54   57    60

, , 3

     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]   61   64   67   70   73   76   79   82   85    88
[2,]   62   65   68   71   74   77   80   83   86    89
[3,]   63   66   69   72   75   78   81   84   87    90
```

One feature that is shared for vectors, matrices, and arrays is that they can only store one type of data at once, be it numerical, character, or logical. Technically speaking, these data structures can only contain elements of the same mode[14].

[14] Data structures that 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.

Data frames are similar to matrices—they are two-dimensional. However, a data frame can contain columns 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" or "experimental unit".

```
> dat<-data.frame(sp=c("Dog","Cat","Human"),sex=c("F","M","F"),
+ weight=c(75.2,186,8.72),living=c(T,F,T))

> dat
     sp sex weight living
1   Dog   F  75.20   TRUE
2   Cat   M 186.00  FALSE
3 Human   F   8.72   TRUE
```

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). Lists are important in the output from many different functions. In the code below, the variables defined above are used to form a list.

```
> summary.1<-list(1.2,x,Y,dat)
> summary.1
[[1]]
[1] 1.2

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

[[3]]
, , 1

     [,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

, , 2

     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]   31   34   37   40   43   46   49   52   55    58
[2,]   32   35   38   41   44   47   50   53   56    59
[3,]   33   36   39   42   45   48   51   54   57    60

, , 3

     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]   61   64   67   70   73   76   79   82   85    88
[2,]   62   65   68   71   74   77   80   83   86    89
[3,]   63   66   69   72   75   78   81   84   87    90
```

```
[[4]]
     sp sex weight living
1   Dog   F  75.20   TRUE
2   Cat   M 186.10  FALSE
3 Human   F   8.72   TRUE
```

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

```
> x<-NULL
```

## *1.7. Functions, arguments, and packages*

In R, you can carry out complicated or tedious procedures using functions. Functions require arguments, which include the object(s) that the function should act upon. For example, the function `sum` will calculate the sum of all its arguments:

```
> sum(1.0,4.214,2.3,8.145,-3.3)
[1] 12.359
```

The arguments in (most) R functions can 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). For example, for the function `plot`, the help file lists the following arguments.

```
plot(x, y, ...)
```

Therefore, we can call up this function with the following code.

```
> a<-1:10
> b<-a
> plot(x=a,y=b)
```

With named arguments, R recognizes the argument keyword (e.g. `x` or `y`) and assigns the given object (e.g. `a` or `b` above) to the correct argument. When using names arguments, the order of the arguments doesn't matter. We can also use what are called positional arguments, where R determines the meaning of the arguments based on their position.

```
> plot(a,b)
```

The expected position of arguments can be found in the help file for the function you are working with or by asking R to list the arguments using the `args` function.

```
> args(plot)
function (x, y, ...)
```

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. Many functions also have default argument values that will be used if values are not specified in the function call. These default

argument values can be seen by using the `args` function and can also be found in the help files. For example, for the function `rnorm`, the arguments `mean` and `sd` have default values.

```
> args(rnorm)
function (n, mean = 0, sd = 1)
```

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 include parentheses, R will return the function code (which can be useful).

Note that it is not necessary to use explicit numerical values as function arguments—symbolic variable names which represent appropriate data structures can be used. It is also possible to use functions as arguments within functions. R will evaluate such expressions from the inside outward. While this may seem trivial, this quality makes R very flexible. There is no explicit limit to the degree of nesting that can be used. You could use:

```
> plot(rnorm(10,sqrt(mean(c(1:5,7,1,8,sum(8.4,1.2,7))))),1:10)
```

which includes 5 levels of nesting (the sum of 8.4, 1.2, and 7 is combined with other values to form a vector, for which the mean value is calculated, then the square root of this value is taken and used as the standard deviation in a call to `rnorm`, and the output from this call is plotted). Of course, it is often easier to assign intermediate steps to symbolic variables. 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 the function `paste` in place of explicit character data.

Many functions (including `sum, plot and rnorm`) come with the R "base packages", i.e. they are loaded and ready to go as soon as you open R. These packages contain the most common functions[15]. While the base packages include many useful functions, for specialized procedures, you should check out the content that is available in the add-on packages. The CRAN website currently lists more than 2400 contributed packages that contain functions and data that users have contributed. You can find a list of the available packages at the CRAN website (http://cran.r-project.org/).

To utilize the functions in contributed R packages, you need to first install and then load the package. Packages can be installed via the Packages menu in the R GUI (select the "Packages" menu, then "Install packages", then select the closest mirror site, and finally, select the package you want to install). Or just use the command[16]:

---

[15] You can find a list of these packages here: http://cran.r-project.org/doc/FAQ/R-FAQ.html.

[16] This simple process can become frustrating if you don't have permission to write to the directory where R saves packages by default (e.g., if your user account does not include complete access). The best way around this is through specifying the locations where packages should be downloaded, say C:\Users\Joe\R_Lib. Then:

```
> install.packages("NADA",lib='C:/Users/Joe/R_Lib')
> library(NADA,lib.loc='C:/Users/Joe/R_Lib')
```

```
> 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("NADA")
```

Installation is a one-time process, but packages must be loaded each time you want to use them. This is very simple, e.g., to load the package NADA, use the following command.

```
> library(NADA)
Loading required package: survival
Loading required package: splines

Attaching package: 'NADA'


        The following object(s) are masked from package:stats :

         predict
```

Any package that you want to use that is not included as one of the "base" packages needs to be loaded every time you start R. (Alternatively, you can add code to the file Rprofile.site that will be executed every time you start R.)

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 should include links to source code, and a pdf manual. When working with a new package, it is a good idea to read the manual.

Some packages contain different functions with the same name, e.g. predict in the stats and NADA packages. The function in use will be the function from the package that was loaded last.

To "unload" functions, use the detach function:

```
> detach("package:NADA")
```

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 topic is covered in a later section.

## *1.8. Missing, indefinite, and infinite values*

Real data sets often contain missing values. R uses the marker NA (for "not available") to indicate a missing value. Any operation carried out on an NA will return NA.

```
> x<-NA
```

```
> x-2
[1] NA
```

Note that the `NA` used in R does not have quotes around it—this would make it character data[17]. To determine if a value is missing, use the `is.na` function (this function can also be used to set elements in a data object to `NA`.)

```
> is.na(x)
[1] TRUE
> !is.na(x)
[1] FALSE
```

Indefinite values are indicated with the marker `NaN`, for "not a number". Infinite values are indicated with the markers `Inf` or `-Inf`. You can find these values with the functions `is.infinite`, `is.finite`, and `is.nan`.

## *1.9. Getting help*

It is usually easy to find the answer to questions about specific functions or about R in general. There are several good introductory books on R, some of which are listed at the end of this workbook. You can also find free detailed manuals on the CRAN website (http://cran.r-project.org/, then select the "manuals" link at the lower left). 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).

Each function in R has a help file associated with it that explains the syntax and usually includes an example. Help files are concisely written. You can bring up a help file by typing `?` and then the function name.

```
> ?aov
```

This will bring up the help file for the `aov` function in your default internet browser. Once the help file is opened, you can of course search within it using your brower's search function (usually accessable with ctrl+f on Windows and apple+f on Macs). But, what if you aren't sure what function you need for a particular task? How can you know what help file to open? In addition to the sources given below, you should try `help.search("keyword")` or `??keyword`, both of which search the R help files for whatever keyword you put in. For example

```
> ??Kruskal
```

returns

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

[17] For character data, missing values are given as `<NA>` to distinguish them from `"NA"`.

```
fuzzy matching:


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


Type '?PKG::FOO' to inspect entry 'PKG::FOO TITLE'.
```

To see a particular help file, follow the instructions given, e.g.,

```
> ?stats::kruskal.test
```

In this case, you can leave out the `stats::` part of the command, since the `stats` package is automatically loaded when R is started.

```
> ?kruskal.test
```

There is an R help mailing list (http://www.r-project.org/mail.html), which can be very helpful. 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 obscure questions, but they are generally not shy about telling users to go back and read the posting guide.

One of the best sources of help on R functions is the mailing list archives (http://cran.r-project.org/, then select "Search" at the upper left, then "Searchable mail archives"). Here you can find suggestions for functions for particular problems, help on using specific functions, and all kinds of other information. A quick way to search the mailing list archives by entering `RSiteSearch("keyword")` into the console. For the most comprehensive search, a good bet is Google—http://www.google.com. To limit the results to R-related pages, adding "cran" seems to work well.

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

```
> apropos("mean")
 [1] "colMeans"         "kmeans"           "mean"
 [4] "mean.data.frame"  "mean.Date"        "mean.default"
 [7] "mean.difftime"    "mean.POSIXct"     "mean.POSIXlt"
[10] "rowMeans"         "weighted.mean"
```

For much more powerful searching capabilities that you can access through the GUI, check out the `sos` package (Graves et al. 2009).

## *Exercises*

1. You can use R for magic tricks: Pick any number. Double it, and then add 12 to the result. Divide by 2, and then subtract your original number. Did you end up with 6.0?

2. If you want to work with a set of 10 numbers in R, something like this:

11.0 8.3 9.8 9.6 11.0 12.0 8.5 9.9 10.0 11.0

what type of data structure should you use to store these in R?

What if you want to work with a data set that contains weight, age, and an categorical assessment of health for 50 whitetail deer—what type of data structure should you use to store these in R?

3. Install and load a package—take a look at the list of available packages, and pick one. To make sure you've loaded it correctly, try to run an example from the package reference manual. Identify the arguments required for calling up the function. Detach the package when you are done.

4. Assign your full name (or someone else's full name) to a variable called `my.name`. Print the value of `my.name` to the GUI. Try to subtract 10 from `my.name`. Finally, determine the type of data stored in my.name and 10 using the `class` function. If you are unsure of what `class` does, check out the help file.

5. Pretend you are interested in seeing what functions R has for generalized additive models (or some other topic). Can you figure out how to search for relevant functions? Are you able to identify a function or two that may do what you want?

# 2. Vectors, matrices, and arrays

*Crawley 2008: Chapter 2, Dalgaard 2008: Chapter 1.2, R-Intro: Sections 2 & 5, Short 2005*

## *2.1. Creating and working with vectors*

There are several ways to create a vector in R. 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
```

or even

```
> 1:10000
    [1]     1     2     3     4     5     6     7     8
    [9]     9    10    11    12    13    14    15    16
...
[9985]  9985  9986  9987  9988  9989  9990  9991  9992
 [9993]  9993  9994  9995  9996  9997  9998  9999 10000
```

The function `seq` (for sequence) is more flexible. Its typical arguments are `from`, `to`, and `by` (or, in place of `by`, you can specify `length.out`).

```
> seq(-10,10,2)
 [1] -10  -8  -6  -4  -2   0   2   4   6   8  10
```

Note that 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
```

For other cases, use `c` (for concatenate or combine).

```
> 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 an assignment operator.

```
> v1<-c(2,1,5,100,2)
> v1
[1]   2   1   5 100   2
```

Variable names can be any combination of letters, numbers, and the symbols . and _, but, they cannot start with a number or with _.

```
> 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
[16] 2.5 2.6 2.7 2.8 2.9 3.0
```

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 substituted into functions calls in place of numeric values.

```
> 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[18].

Logical vectors are very useful in R for subsetting data, i.e., for isolating some part of an object that meets certain criteria. For relational commands, 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, note that when logical vectors are used in arithmetic, they are changed (coerced in R terms) into a vector of binary elements: 1 or 0. Continuing with the above example:

```
> a<-x>5
> a
 [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE

> a*1.4
```

[18] This does not mean you cannot comibine data of different modes into a single vector using `c`—you actually can, but R will coerce all the elements to the same mode.

```
 [1] 0.0 0.0 0.0 0.0 0.0 1.4 1.4 1.4 1.4 1.4
```

One function that is commonly used on character data is `paste`. It concatenates character data (and can also work with numerical 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, 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.

```
> month<-"March"
> day<-12
> year<-2009
> paste("Today is the ",day,"th day of ",month,", ",year,sep="")
[1] "Today is the 12th day of March, 2009"
```

This is especially useful with loops, when a variable with a changing value is combined with other data. Loops will be discussed in a later section.

```
> group<-1:10
> id<-LETTERS[1:10]
> for(i in 1:10) {
+ print(paste("group =",group[i],"id =",id[i]))
+ }
[1] "group = 1 id = A"
[1] "group = 2 id = B"
[1] "group = 3 id = C"
[1] "group = 4 id = D"
[1] "group = 5 id = E"
[1] "group = 6 id = F"
[1] "group = 7 id = G"
[1] "group = 8 id = H"
[1] "group = 9 id = I"
[1] "group = 10 id = J"
```

Note that the separator can be specified as well using the `sep` argument (default is a single space " "). `LETTERS` is actually a constant (one of only a few) that is built into R—it is a vector of uppercase letters A through Z (different from `letters`).

## *2.2. Vector arithmetic, some common functions, and vectorized operations*

In R, vectors can be used directly in arithmetic expressions. 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 is a quality that makes R a very efficient programming language[19].

```
> x<-6:10
> x
[1]  6  7  8  9 10
> x+2
[1]  8  9 10 11 12
```

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

```
> y<-c(4,3,7,1,1)
> y
[1] 4 3 7 1 1

> z<-x+y
> z
[1] 10 10 15 10 11
```

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:

```
> x<-1:10
> m<-0.8
> b<-2
> y<-m*x + b
> y
 [1]  2.8  3.6  4.4  5.2  6.0  6.8  7.6  8.4  9.2 10.0
```

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

```
> x<-1:10
> m<-0.8
> b<-c(2,1,1)
> y<-m*x + b
Warning message:
longer object length
        is not a multiple of shorter object length in: m * x + b
```

[19] Efficient for code-writers, that is.

Some arithmetic operators that are available in R are:

| | |
|---|---|
| `+` | addition |
| `-` | subtraction |
| `*` | multiplication |
| `/` | division |
| `^` | exponentiation |
| `%/%` | integer division |
| `%%` | modulo (remainder) |
| `log(a)` | natural log of a |
| `log10(a)` | base 10 log of a |
| `exp(a)` | $e^a$ |
| `sin(a)` | sine of a |
| `cos(a)` | cosine of a |
| `tan(a)` | tangent of a |
| `sqrt(a)` | square root of a |

Some simple functions that are useful for vector math include:

| | |
|---|---|
| `min` | minimum value of a set of numbers |
| `max` | maximum of a set of numbers |
| `pmin` | parallel minima (compares multiple vectors "row-by-row") |
| `pmax` | parallel maxima |
| `sum` | sum of all elements |
| `length` | length of a vector (or the number of columns in a data frame) |
| `NROW` | number of rows in a vector or data frame |
| `mean` | arithmetic mean |
| `sd` | standard deviation |
| `rnorm` | generates a vector of normally-distributed random numbers |
| `signif, ceiling, floor` | rounding |

Many, many other functions are available.

R also has a few built in constants, including `pi`.

```
> pi
[1] 3.141593
```

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

```
> 7 - 2*4
[1] -1
```

is different from:

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

```
[1] 20
```

and

```
> 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 (or even data frames and arrays, lists) 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
 [7] 2.645751 2.828427 3.000000 3.162278
```

or

```
> sqrt(1:10)
 [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490
 [7] 2.645751 2.828427 3.000000 3.162278
```

The previous expressions are also equivalent to:

```
> sqrt(c(1,2,3,4,5,6,7,8,9,10))
 [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490
 [7] 2.645751 2.828427 3.000000 3.162278
```

But they are not the same as 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)
Error in sqrt(1, 2, 3, 4, 5, 6, 7, 8, 9, 10) :
  10 arguments passed to 'sqrt' which requires 1
```

There are also some functions designed for making vectorized (or list-ized?) operations on lists, matrices, and arrays: these include `apply` and `lapply`. We will cover these in a later section.

## *2.3. Matrices and arrays*

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 arrays or matrices, but these data structures become are useful for some tasks.

To generate a matrix, the `matrix` function can be used. For example:

```
> X<-matrix(1:15,nrow=5,ncol=3)
> X
     [,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.

```
> as.vector(X)
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

If, for any reason, you want to change the filling order, you can use the `byrow` argument:

```
> X<-matrix(1:15,nrow=5,ncol=3,byrow=T)
> X
     [,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:

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

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

Arithmetic with matrices and arrays that have the same dimensions is straightforward, and is done on an element-by-element basis. This is true for all the arithmetic operators listed in earlier sections.

```
> Z<-matrix(1,nrow=5,ncol=3)
> Z
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    1    1    1
[3,]    1    1    1
[4,]    1    1    1
[5,]    1    1    1
> X + Z
     [,1] [,2] [,3]
[1,]    2    7   12
[2,]    3    8   13
[3,]    4    9   14
[4,]    5   10   15
[5,]    6   11   16
```

This doesn't work when dimensions don't match:

```
> Z<-matrix(1,nrow=3,ncol=3)
> X + Z
Error in X + Z : non-conformable arrays
```

For mixed vector/array arithmetic, vectors are recycled if needed.

```
> Z
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    1    1    1
[3,]    1    1    1

> x<-1:9
> Z+x
     [,1] [,2] [,3]
[1,]    2    5    8
[2,]    3    6    9
[3,]    4    7   10

> y<-1:3
> Z+y
     [,1] [,2] [,3]
[1,]    2    2    2
[2,]    3    3    3
```

```
[3,]    4    4    4
```

R also has operators for matrix algebra. The operator `%*%` carries out matrix multiplication, and the function `solve` can invert matrices.

```
> X<-matrix(c(1,2.5,6,3.2,4,5,6,4,9),nrow=3)
> X
     [,1] [,2] [,3]
[1,]  1.0  3.2    6
[2,]  2.5  4.0    4
[3,]  6.0  5.0    9

> solve(X)
            [,1]        [,2]        [,3]
[1,] -0.33195021 -0.02489627  0.23236515
[2,] -0.03112033  0.56016598 -0.22821577
[3,]  0.23858921 -0.29460581  0.08298755
```

A useful function for working with matrices is the `outer` function. In its simplest usage, it will apply a specified function to all combinations of the elements in two vectors given as arguments. This can be handy for, e.g., contour plots. Note that the function specified as the `FUN` argument must be a vectorized function. For example, let's make a multiplication table[20]:

```
> x<-y<-1:11
> outer(x,y,"*")

      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
 [1,]    1    2    3    4    5    6    7    8    9    10    11
 [2,]    2    4    6    8   10   12   14   16   18    20    22
 [3,]    3    6    9   12   15   18   21   24   27    30    33
 [4,]    4    8   12   16   20   24   28   32   36    40    44
 [5,]    5   10   15   20   25   30   35   40   45    50    55
 [6,]    6   12   18   24   30   36   42   48   54    60    66
 [7,]    7   14   21   28   35   42   49   56   63    70    77
 [8,]    8   16   24   32   40   48   56   64   72    80    88
 [9,]    9   18   27   36   45   54   63   72   81    90    99
[10,]   10   20   30   40   50   60   70   80   90   100   110
[11,]   11   22   33   44   55   66   77   88   99   110   121
```

[20] This may not be the best example, since the `fun` argument is actually an operator and not a function. To use an operator here, note that you have to enclose it in quotes. You can try using a function, such as `pmax`, instead.

## *Exercises*

1. Generate a vector of numbers that contains the sequence 1, 2, 3,. . .10 (try to use the least amount of code possible to do this). Assign this vector to the variable `x`, and then carry out the following vector arithmetic. Make sure your answers match the values given below.

$\log_{10} x$ (=0, 0.301, 0.477. . .)

$\ln x$ (=0, 0.69, 1.099. . .)

$\frac{\sqrt{x}}{2-x}$ (=1, Inf, -1.7. . .)

2. Use an appropriate function to generate a vector of 100 numbers that go from 0 to 2π, with a constant interval. Assuming this first vector is called x, create a new vector that contains sine(2x - 0.5π). Determine the minimum and the maximum of sine(2x - 0.5π). Does this match what you expect?

3. Create 5 vectors, each containing 10 random numbers. Give each vector a different name. Creacte a new vector where the 1st element contains the sum of the 1st elements in your original 5 vectors, the 2nd element contains the sum of the 2nd elements, etc. Determine the mean of this new vector. (Hint: this should be a very easy set of operations.)

4. Create the following matrix using the least amout of code (ir should be only around 30 characters):

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

5. If you are bored, try this. Given the following set of linear equations:

27.2x + 32y – 10.8z = 401.2
x – 1.48y = 0
409.1x +13.5z = 2.83

Solve for x, y, and z using matrix algebra.

# 3. Data frames, data import, and data export

*Crawley 2007: Chapter 4, Dalgaard 2008: Sections 1.2.10 & 2.4, R-Data: Section 1.2, R-Intro: Sections 6 & 7*

## *3.1. Reading data from files*

As described above, a data frame is a type of data structure in R with rows and colums, 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.table`, which works with ASCII text files. Data can be read in from other files as well, using different functions, but `read.table` is the most commonly used approach. R is very flexible in how it reads in data from text files. Typically, organization will be as follows

```
agency     site       date       discharge  flag.discharge
USGS       4232730    2006-01-01      75         P
USGS       4232730    2006-01-02     493         P
USGS       4232730    2006-01-03    1380         P
USGS       4232730    2006-01-04    1910         P
USGS       4232730    2006-01-05    1940         P
. . .
```

Note that the column labels in the header have to be compatable with R's variable naming convention, or else R will make some changes as they are read in (or won't read the data in correctly). With this option, R will assign row numbers based on the order of the observations. So, for example, the data shown above without row labels are in the text file River_flow.txt, and they can be read in and assigned to the data frame `flow.dat` using the following command.

```
> flow.dat<-read.table("River_flow.txt",header=TRUE)
> flow.dat
   agency    site       date discharge flag.discharge
1    USGS 4232730 2006-01-01        75              P
2    USGS 4232730 2006-01-02       493              P
3    USGS 4232730 2006-01-03      1380              P
4    USGS 4232730 2006-01-04      1910              P
...
```

Note that you must specify `header=TRUE`, or else R will interpret the row of labels as data. If the file you are trying to load is not in the directory that R is working in (the working directory, which can be checked with `getwd()` and changed with `setwd(file= "filename")` or through the File menu [after selecting "File", select "Change dir…"]) you can include a file path, but note that the path should have forward, not backward, slashes (or double backward slashes, if you prefer).

If you do not specify a field separtor (the `sep` argument) R assumes that any spaces or tabs separate the data in your text file. In this case, the number of white space 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 white space as different elements).

Alternately, other separators can be used. If you specify a separator (say `sep="\t"` for tabs or `sep=","` for commas) two consecuative 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, e.g. `read.csv`, which is identical to `read.table`, except default arguments differ. Also, 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. A few other handy bits of information for using `read.table` follow. You can include comments at the end of rows in your data file—just preceed them with a `#`. Also, R will recognize `NaN`, `Inf`, and `-Inf` in input files.

Probably the easiest approach to handling missing values is to indicate their presence with `NA` in the text file. R will automatically recognize these as missing values. Since the file River_flow.txt uses `NA` for missing values, they should have been read in properly[21].

```
> which(is.na(flow.dat$discharge))
[1] 273
> flow.dat[271:275,]
    agency    site       date discharge flag.discharge
271   USGS 4232730 2006-09-28      1430              P
272   USGS 4232730 2006-09-29      1430              P
273   USGS 4232730 2006-09-30        NA           <NA>
274   USGS 4232730 2006-10-01      1290              A
275   USGS 4232730 2006-10-02       649              A
```

In most cases, it makes the most sense to put your data into a text file for reading 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. For data that are in a spreadsheet program such as Excel you have at least two options. Data can be copied and pasted into a text editor and saved as a text file—this creates a tab-delimited file. Alternatively, data can be saved directly from spreadsheet programs, e.g. as "Formatted Text (Space delimited)" (*.prn) in Excel (although this can be problematic for spreadsheets with a large number of columns), or as comma-separated values (*.csv).

If it is not possible to convert your data file into a text file, e.g. if the original software is not available for opening the file, it is likely that you can find a function for reading the file directly. R has the capability to handle many different formats[22].

---

[21] The tricks used in these two commands are described in section 5.2.

[22] See the `foreign` package.

This may all seem confusing, but it is really not that bad. Your best bet is to play around with the different options, find one that you like, and stick with it. Let's work on another example. First, 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 present. So, it should be easy to read in:

```
> pop.dat<-read.table("US_pop.txt",header=TRUE)
> pop.dat
   year       pop
1  1790   3929214
2  1800   5308483
3  1810   7239881
...
21 1990 248709873
22 2000 281421906
```

Note that we did not need to specify a separator, since one or more white space characters (spaces, in this case) are interpreted as a separator[23]. This type of `read.table` statement, that includes only the file name and `header=TRUE` is the approach that we recommend. Unless your file has unquoted character strings that contain white space 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.

Data frames can actually be edited interactively in R using the `edit` function. This is really only useful for small data sets.

```
> pop.dat<-edit(pop.dat)
```

## *3.2. Creating data frames manually*

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

```
> date<-c("1-FEB-2008","17-APR-2008","20-JUN-2008","19-SEPT-2008")
> mass<-c(1.8,3.4,6.3,7.8)
> dat<-data.frame(sample.date=date,mass.mean=mass)
> dat
```

[23] In fact, if we had specified `sep=" "` R would have returned an error, since the number of spaces between entries is not consistent in this file.

```
   sample.date mass.mean
1   1-FEB-2008       1.8
2  17-APR-2008       3.4
3  20-JUN-2008       6.3
4 19-SEPT-2008       7.8
```

While this approach is not an efficient way to enter data that could be read in directly, it can be very handy for some applications, e.g. creating customized summary tables. Note that column names are specified using an equal sign. It is also possible to specify (or change, or check) column names for an existing data frame using the function `names`.

```
> names(dat)<-c("Date","Mass")
> dat
          Date Mass
1   1-FEB-2008  1.8
2  17-APR-2008  3.4
3  20-JUN-2008  6.3
4 19-SEPT-2008  7.8
```

Row names (1:4 above) can be specified in the `data.frame` function with the `row.names` argument.

```
> dat<-data.frame(sample.date=date,mass.mean=mass,
+ row.names=c("D","E","A","B"))
> dat
   sample.date mass.mean
D   1-FEB-2008       1.8
E  17-APR-2008       3.4
A  20-JUN-2008       6.3
B 19-SEPT-2008       7.8
```

Specifying row names can be useful if you want to index data, which will be covered later. Row names can also be specified for an existing data frame with the `rownames` function (not to be confused with the `row.names` argument).

## *3.3. 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. It it usually necessary to be able to select and identify specific columns (i.e., vectors) within data frames. There are two ways to specify a given column of data from within a data frame. The first is to use the `$` notation. To demonstrate, let's read in some data on biological hydrogen production from glucose:

```
> h2.dat<-read.table("Biohydrogen.txt",header=TRUE)
```

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

```
> names(h2.dat)
[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 $H_2$ and $CO_2$ in this case) produced by a particular reactor[24].

```
> h2.dat$vol
  [1]  0.00  0.00 19.50 14.25  9.10 24.20 17.50  4.00  4.00  0.00
 [11]  0.00 21.40 16.20  9.90 25.50 17.40  4.00  0.00  0.00  0.00
 [21] 22.50 17.00 10.50 26.60 17.10  2.00  0.00  0.00  0.00 21.20
 [31] 15.30 10.40 23.60 16.80  7.00  4.00  0.00  0.00 20.40 12.70
 [41]  9.30 21.20 20.70  2.60    NA  0.00  0.00  0.00  4.20  0.00
 [51]  0.00  0.00  1.85  1.70  0.00  0.00  0.00  3.60  0.00  0.00
 [61]  0.00  0.00  2.50  0.00  0.00  0.00  3.70  0.00  0.00  0.00
 [71]  2.00  0.80  0.00  0.00 17.80 14.40 14.00 24.60  5.10  2.60
 [81]  5.50  0.00  0.00 22.75 20.00 16.50 14.90  7.00  4.00  3.00
 [91]  0.00  0.00 21.90 19.20 16.60 17.50  6.30  4.40  1.50  0.00
[101]  0.00 25.20 19.80 17.30 17.30  6.60  2.70  2.80  0.00  0.00
[111] 24.60 21.20 18.80 15.10  9.00  2.40  0.00  0.00  0.00 19.60
[121] 18.30 16.30 23.00  5.00  2.10  6.20  0.00  0.00 11.35  0.00
[131]  0.00  0.00  0.00  2.00  0.00
```

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

```
> mean(h2.dat$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):

```
> mean(na.omit(h2.dat$vol))
[1] 7.719403
```

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

```
> attach(h2.dat)
> vol
  [1]  0.00  0.00 19.50 14.25  9.10 24.20 17.50  4.00  4.00  0.00
 ...
```

---

[24] With the `$` notation, you don't even need to specify the complete name of the column you want—just enough to distinguish it from other columns is sufficient, so `h2.dat$v` would also work here.

```
[131]  0.00  0.00  0.00  2.00  0.00
```

Note that when you are done using the individual columns, it is good practice to `detach` your data frame. Once the data frame is `detach`ed, R will no longer know what you mean when you specify the name of a column alone:

```
> detach(h2.dat)
> vol
Error: object "vol" not found
```

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.

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

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

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

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

Here is what our new data frame looks like:

```
> h2.dat
    reactor      date  time   vol conc.h2        date.time   vol.L
1      G172 9/18/2006 11:12  0.00      NA 9/18/2006, 11:12 0.00000
2      G172 9/18/2006 14:00  0.00    0.00 9/18/2006, 14:00 0.00000
3      G172 9/19/2006  9:26 19.50    7.73  9/19/2006, 9:26 0.01950
...
135    G171 9/21/2006 12:40  0.00   30.26 9/21/2006, 12:40 0.00000
```

Both the `$` notation and the `attach` and `detach` functions can be used to specify data vectors within any other function. However, there are other options when using functions. For some functions, you can specify the data frame that should be used with the `data` argument, e.g. `data=h2.dat`, and then refer to the column(s) within the data frame directly by name. This argument can be used in many different functions. 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.

Many data frames that contain real data will have some missing observations. R has several tools for working with these observations. For starters, the `na.omit` function can be used for removing `NA`s from a vector. Let's work with the `conc.h2` column, which contains the concentration of $H_2$ (% volume) in the produced biogas.

```
> h2.dat$conc.h2
  [1]    NA  0.00  7.73 10.72 15.69 21.64 23.53 26.52 25.06    NA
```

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

> na.omit(h2.dat$conc.h2)
 [1]  0.00  7.73 10.72 15.69 21.64 23.53 26.52 25.06  0.00  8.39
[11] 11.89 17.84 24.24 27.20 26.23 27.79  0.00  7.85 11.54 18.72
[21] 22.38 24.77 28.46 27.06  0.00  7.38 10.66 11.89 20.97 25.36
[31] 17.01 25.22  0.00  6.57  9.29 13.08 16.82 22.88 22.10  0.00
[41]  6.67 10.51 14.48 21.20 23.61 18.96 26.25  0.00  8.31 13.55
[51] 21.26 23.11 25.78 26.40 27.13  0.00  7.50 11.82 17.21 21.08
[61] 22.86 22.00 26.49 31.20 35.22 30.93 30.26
attr(,"na.action")
 [1]   1  10  19  28  37  45  46  47  48  49  50  51  52  53  54  55
[17]  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71
[33]  72  73  82  91 100 101 102 103 104 105 106 107 108 109 110 111
[49] 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127
[65] 130 131 132 133
attr(,"class")
[1] "omit"
```

Although the result does contain more than just the non-NA values, only the non-NA values will be used in subsequent operations[25]. Note that the result of `na.omit` contains more information than just the non-NA values. This function can also be applied to complete data frames. In this case, any row with an NA is removed.

```
> h2.clean.dat<-na.omit(h2.dat)
```

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.dat$conc.h2)
  [1]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
 [11] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
 [21] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
```

[25] 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 number of values that were removed.

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

## *3.4. Writing data to files*

With R, it is easy to write data to files. The function `write.table` is usually the best function for this purpose. 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 given in the associated help file:

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

For example, if we wanted to write out the entire contents of the `h2.dat` dataframe we could use the following code:

```
> write.table(h2.dat,"h2.out")
```

Setting the `append` argument to `TRUE` lets you add data to a file that already exists. Note that there are several other arguments for changing the appearance of the output.

The `write.table` function cannot be used with all data structures in R (e.g. lists), but it is possible to send all R output to a file using the `sink` function. The follow command would send all the susequent 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()
```

## *Exercises*

1. The file Thakali_Ni_EC50s.txt contains nickel EC50s for barley root elongation in soils, along with relevant soil chemistry. Open the file to see how it is formatted (if you don't have a text editor that shows tabs and spaces, you can use Microsoft Word, and click the "show paragraph marks" button, ¶), and then read the data into R using the function `read.table`. Print the resulting data frame to the screen to make sure the data were read in correctly.

Did you include `sep` in your command? If not, do so now, if so, try the command without the `sep` argument. Is it needed? Delete the quotes that are present in the data file around the soil names (use the search and replace feature of your text editor) and save the file with a new name. Now do you need the `sep` argument to read the data in? Why?

2. Determine the minimum and maximum soil pH (ph.soil) in your new data frame that contains the Thakali data. Next, add a new column to the data frame that contains the $\log_{10}$ of the EC50 (`ec50.ni`).

3. Create a new data frame that contains the mean Ni EC50, soil pH, and soil organic carbon (`oc`) from the Thakali data. Write out the summary to a new file using the default options. Write the data out again, but this time, see if you can eliminate the row labels. Finally, try changing the separator to a tab.

4. Open the Excel file Carion_beetles.xls, which contains data on the presence of several species of carion beetles in 25 locations in NY state. Can you think of more than one way to get these data into R? Save these data to a text file using your chosen method, and try reading them into R using `read.table`. Try adding a space between the genus abbreviation and the species name for one or two species and repeat the process. Does this cause any problems?

# 4. Graphics, part I

*Dalgaard 2008: Section 2.2, R-Intro: Section 12, Murrell 2005*

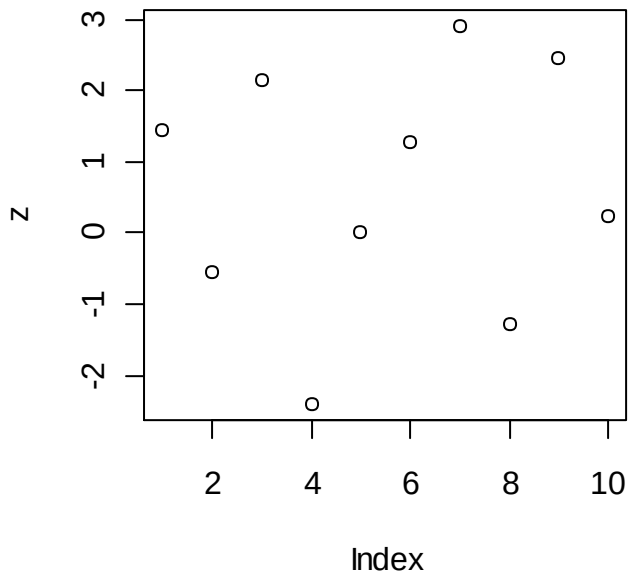
## *4.1. Introduction to the plot function*

It is easy to produce publication-quality graphics in R. However, in this first section on graphics, we will focus on the simplest plots that can be produced with the `plot` function. This function produces a plot as a side effect, but the type of plot produced depends on the type of data submitted. The basic plot arguments, as given in the help file for plot.default are:

```
plot(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, ...)
```

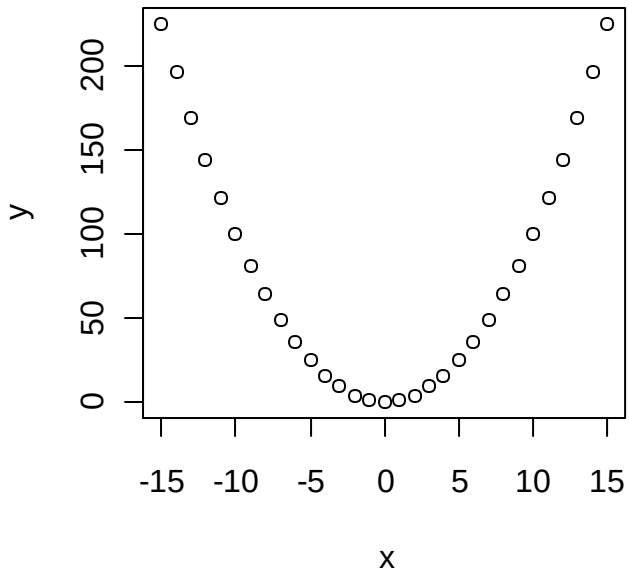
To plot a single vector, all we need to do is supply that vector as the only argument to the function:

```
> z<-rnorm(10)
> plot(z)
```




In this case, R simply plots the data in the order they occur in the vector. To plot one variable versus another, just specify the two vectors for the first two arguments:

```
> x<- -15:15
> y<-x^2
> plot(x,y)
```



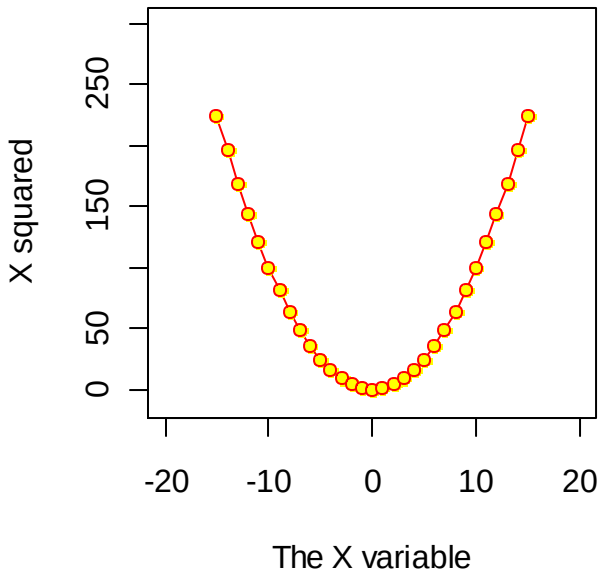

And that is all it takes to generate plots in R, as long as you like the default settings. Of course, the default settings generally won't be sufficient for publication- or presentation-quality graphics. Fortunately, plots in R are very flexible. The table below shows some of the more common arguments to the `plot` function, and some of the common settings. For many more arguments, see the help file for `par`.

| Argument | Common options | Additional information |
|---|---|---|
| `bg` | `"red"`<br>`"blue"`<br>many more | Color of fill for some plotting symbols (see below) |
| `col` | `"red"`<br>`"blue"`<br>`1`<br>Through<br>`657` | Color of plotting symbols and lines. Type `colors()` to get list. You can also mix your own colors. See "Color Specification" in the help file for par. |
| `las` | `0`<br>`1`<br>`2`<br>`3` | Rotation of numeric axis labels. |
| `log` | `"x"`<br>`"y"`<br>`"xy"` | For making log axes. If you have multiple decades, you may have to use axis function (or our function log.axis) for nice-looking axes |

| | | |
|---|---|---|
| lty | 0<br>1 or "solid"<br>2 or "dashed"<br>3 or "dotted"<br>through<br>6 | Line types. Can also specify custom types, e.g, 23 would have a 2-unit dash followed by a 3-unit space. Note that lines simply connect points—there is no automatic "smoothing". |
| main | Any character string, e.g., "Plot 1" | Adds a main title at the top. |
| pch | 0<br>through<br>25 | Plotting symbols. See below for symbols. Can also use any single character, e.g., "v" or "@". |
| type | "p" for points<br>"l" for line<br>"b" for both<br>"o" for over<br>"n" for none | "n" can be handy for setting up a plot that you later add data to. |
| xlab, ylab | Any character string, e.g., "Income (US$)" | For specifying axis labels. |
| xlim, ylim | Any 2-element vector, e.g.,<br>c(0,100)<br>c(-10,10)<br>c(55,0) | List higher value first to reverse axis |

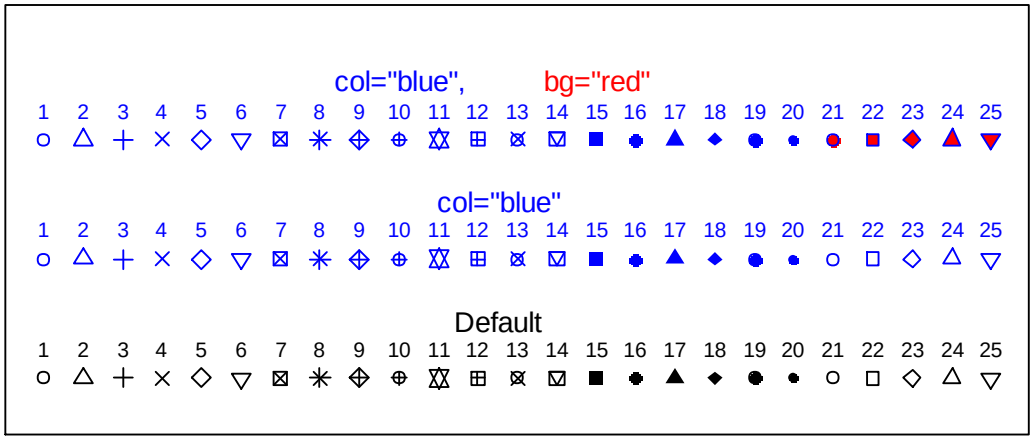
Use of some of the above arguments is shown in the following example.

```
> plot(x,y,type="o",xlim=c(-20,20),ylim=c(-10,300),pch=21,col="red",
+ bg="yellow",xlab="The X variable",ylab="X squared")
```




The `plot` function is effectively vectorized. It accepts vectors for the first two arguments (which specify the x and y position of your observations), but can also accept vectors for some of the other arguments, including `pch` or `col`. Among other things, this provides an easy way to produce a reference plot demonstrating R's plotting symbols and lines. If you use R regularly, you may want to print a copy out (or make your own).

```
> 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()
```




## *Exercises*

1. Produce a data frame with two columns: x, which ranges from -2π to 2π and has a small interval between values (for plotting), and cosine(x). Plot cosine(x) vs. x as a line. Repeat, but try some different line types or colors.

2. Read in the data in the file Oxychem.txt, which contains the concentration of the chemical Dechlorane Plus in tree bark from western NY State and surrounding areas. Plot the Dechlorane Plus concentration (`dechlor`) vs. the distance (`dist`—in km) from the suspected source (OxyChem). Try using linear and logarithmic axes, and be sure to add appropriate axis labels and a heading. Try a few plotting symbols and colors.

# 5. Manipulating data, part I

*Crawley 2007: Chapters 2 & 4, Dalgaard 2008: Sections 1.2 & 10.1.3, R-Intro: Section 5.4, R-Lang: Section 3.4, Venables & Ripley 2002: Chapter 2. Spector 2008*

## *5.1. Modes, classes, attributes, length, and coercion*

As described above, the mode of an object describes the type of data that it contains. In R, mode is an object attribute. All objects have at least two attributes: mode and length, but many objects have more.

```
> x<-1:10

> mode(x)
[1] "numeric"

> length(x)
[1] 10
```

It is often necessary to change the mode of a data structure, e.g. to have your data displayed differently, or to apply a function that only works with a particular type of data structure. In R this is called coercion. There are many functions in R that have the structure `as.something` that change the mode of a submitted object to "something". For example, say you want to treat numeric data as character data.

```
> x<-1:10
> as.character(x)
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
```

Or, you may want to turn a matrix into a data frame.

```
> X<-matrix(1:30,nrow=3)
> as.data.frame(X)
  V1 V2 V3 V4 V5 V6 V7 V8 V9 V10
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
```

If you are unsure of whether or not a coercion function exists, give it a try—two other common examples are `as.numeric` and `as.vector`.

Attributes are important internally for determining how objects should be handled by various functions. In particular, the `class` attribute determines how a particular object will be handled by a given function. For example, output from a linear regression has the class `"lm"`, and will be handled differently by the `print` function than will a data frame, which has the class `"data.frame"`. The utility of this object-oriented approach will become more apparent later on.

It is often necessary to know the length of an object. Of course, length can mean different things. Three useful functions for this are `nrow`, `NROW`, and `length`.

The function `nrow` will return the number of rows in a two-dimensional data structure.

```
> X<-matrix(1:30,nrow=3)
> X
     [,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
> nrow(X)
[1] 3
```

The vertical analog is `ncol`.

```
> ncol(X)
[1] 10
```

You can get both of these at once with the `dim` function.

```
> dim(X)
[1]   3  10
```

For a vector, use the function `NROW or length`.

```
> x<-1:10
> NROW(x)
[1] 10
```

The value returned from the function `length` depends on the type of data structure you submit, but for most data structures, it is the total number of elements.

```
> length(X)
[1] 30
> length(x)
[1] 10
```

## *5.2. Indexing, sub-setting, splitting, sorting, and locating data*

Subsetting and indexing are ways to select specific parts of a data structure (such as specific rows within a data frame) within R. Indexing (also known as subscripting) is done using square brackets in R:

```
> v1<-c(5,1,3,8)
> v1
[1] 5.0 1.0 3.0 8.0
```

Say we want the 3rd observation:

```
> v1[3]
[1] 3.2
```

R is very flexible in terms of what can be selected or excluded.

This returns the 1st through 3rd observation:

```
> v1[1:3]
[1] 5.0 1.0 3.2
```

While this returns all but the 4th observation:

```
> v1[-4]
[1] 5.0 1.0 3.2
```

This bracket notation can also be used with relational constraints. For example, if we want only those observations that are < 5.0:

```
> v1[v1<5]
[1] 1.0 3.2
```

This may seem a bit confusing, but if we evaluate each piece separately, it becomes more clear:

```
> v1<5
[1] FALSE  TRUE  TRUE FALSE

> v1[c(FALSE,TRUE,TRUE,FALSE)]
[1] 1.0 3.2
```

While we are on the topic of subscripts, we should note that, 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. To increase the size of a vector, just assign a value to a position that doesn't currently exist:

```
> length(v1)
[1] 4

> v1[8]<-10
> length(v1)
[1] 8
> v1
[1]  5.0  1.0  3.2  8.0   NA   NA   NA 10.0
```

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 subscripting as applied to data frames, let's read in a file:

```
> flow.dat<-read.table("River_flow.txt",header=TRUE)
```

This data frame has 730 rows and five columns:

```
> dim(flow.dat)
[1] 730   5

> flow.dat
    agency    site       date discharge flag.discharge
1     USGS 4232730 2006-01-01        75              P
2     USGS 4232730 2006-01-02       493              P
3     USGS 4232730 2006-01-03      1380              P
...
729   USGS 1509000 2006-12-30       672              A
730   USGS 1509000 2006-12-31       628              A
```

Let's say we want only the first five rows and first two columns:

```
> flow.dat[1:5,1:2]
  agency    site
1   USGS 4232730
2   USGS 4232730
3   USGS 4232730
4   USGS 4232730
5   USGS 4232730
```

If an index is left out, R returns all values in that dimension (you need to include the comma).

```
> flow.dat[1:5,]
  agency    site       date discharge flag.discharge
1   USGS 4232730 2006-01-01        75              P
2   USGS 4232730 2006-01-02       493              P
3   USGS 4232730 2006-01-03      1380              P
4   USGS 4232730 2006-01-04      1910              P
5   USGS 4232730 2006-01-05      1940              P
```

You can also specify row or column names directly within the brackets—this can be very handy when order may change in future versions of your code.

```
> flow.dat[1:5,"site"]
[1] 4232730 4232730 4232730 4232730 4232730
```

You can also specify multiple column names using the `c` function[26].

[26] Since you are using character data to identify columns in the last two examples (both `"site"` and `c("agency","site")` are chacter vectors), it is also possible to use the function `paste`. This can be handy, for example, when you are working in a loop, and want to select a different row or column with each iteration.

```
> flow.dat[1:5,c("agency","site")]
  agency    site
1   USGS 4232730
2   USGS 4232730
3   USGS 4232730
4   USGS 4232730
5   USGS 4232730
```

Relational constraints can also be used in indexes.

```
> flow.dat[flow.dat$discharge<60,]
    agency    site       date discharge flag.discharge
50    USGS 4232730 2006-02-19        55              P
77    USGS 4232730 2006-03-18        31              P
78    USGS 4232730 2006-03-19        52              P
80    USGS 4232730 2006-03-21        59              P
92    USGS 4232730 2006-04-02        50              P
100   USGS 4232730 2006-04-10        58              P
106   USGS 4232730 2006-04-16        51              P
116   USGS 4232730 2006-04-26        56              P
NA    <NA>      NA       <NA>        NA           <NA>
```

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[27]. 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[28].

```
> subset(flow.dat,discharge<60)
    agency    site       date discharge flag.discharge
50    USGS 4232730 2006-02-19        55              P
77    USGS 4232730 2006-03-18        31              P
78    USGS 4232730 2006-03-19        52              P
80    USGS 4232730 2006-03-21        59              P
92    USGS 4232730 2006-04-02        50              P
100   USGS 4232730 2006-04-10        58              P
106   USGS 4232730 2006-04-16        51              P
116   USGS 4232730 2006-04-26        56              P
```

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.dat,discharge>4000 & site!=4232730)
    agency    site       date discharge flag.discharge
```

[27] A note about the `subset` function that may or may not ever be relevant: by default, a subsetted object retains all the levels of a factor (see following sections for information on factors). If you want to remove unused levels, use `drop=TRUE`.

[28] Well, the result is almost identical to the previous result—note the different handling of `NA` values.

```
438   USGS 1509000 2006-03-14      4540             A
544   USGS 1509000 2006-06-28      6050             A
545   USGS 1509000 2006-06-29      4760             A
```

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 where `flag.discharge` is `A`, `Ae`, or `P`. We could use this:

```
> subset(flow.dat,flag.discharge=="A" | flag.discharge=="Ae" |
flag.discharge=="P")
    agency    site       date discharge flag.discharge
1     USGS 4232730 2006-01-01        75              P
2     USGS 4232730 2006-01-02       493              P
3     USGS 4232730 2006-01-03      1380              P
4     USGS 4232730 2006-01-04      1910              P
...
```

But, this is an easier way:

```
> subset(flow.dat,flag.discharge %in% c("A","Ae","P"))
    agency    site       date discharge flag.discharge
1     USGS 4232730 2006-01-01        75              P
2     USGS 4232730 2006-01-02       493              P
3     USGS 4232730 2006-01-03      1380              P
4     USGS 4232730 2006-01-04      1910              P
...
```

Indexing matrices and arrays follows what we have just covered. For example:

```
> X<-matrix(1:30,nrow=3)
> X
     [,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

> X[3,8]
[1] 24

> X[,3]
[1] 7 8 9

> Y<-array(1:90,dim=c(3,10,3))
> Y[3,1,1]
[1] 3
```

Indexing is a little trickier for lists—you need to use double square brackets, `[[i]]`, to specify an element within a list[29]. Of course, if the element within the list has multiple elements, you could use indexing to select specific elements within it.

```
> list.1<-list(1:10,X,Y)
> list.1[[1]]
 [1]  1  2  3  4  5  6  7  8  9 10
```

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.

```
> list.1[[2]][3,2]
[1] 6
```

Indexing, either alone or in combination with the `subset` function, can be used for some pretty sophisticated subsetting. For example, taking the hydrogen production data:

```
> h2.dat<-read.table("Biohydrogen.txt",header=TRUE)
> h2.dat
    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
...
134    G185 9/20/2006 13:41  2.10      NA
135    G185 9/21/2006 12:40  6.20      NA
```

Let's say we want to select only data from those reactors that do not have missing $H_2$ concentrations, with an exception for the first observation (11:12 on 9/18/2006, for which no measurements were made). This exception makes the code more complicated, but see if you can figure out how this works:

```
> subset(h2.dat,!reactor %in% h2.dat$reactor[h2.dat$date!="9/18/2006"
+ & h2.dat$time!="11:12"][is.na(h2.dat$conc.h2[h2.dat$date!=
+ "9/18/2006" & h2.dat$time!="11:12"])])
    reactor      date  time   vol conc.h2
10     G172 9/18/2006 11:12  0.00      NA
11     G172 9/18/2006 14:00  0.00    0.00
12     G172 9/19/2006  9:26 19.50    7.73
...
107    G182 9/20/2006 13:41  4.40   22.00
108    G182 9/21/2006 12:40  1.50   26.49
```

[29] You can use single brackets too, but doing so will return an object that is still a list.

An easy way to divide data into groups is to use the `split` function. This function will divide a data structure (typically a vector or a data frame) into one subset for each level of the variable you would like to split by. The subsets are stored together in a list.

```
h2.splt<-split(h2.dat,h2.dat$reactor)
```

This function can be handy for viewing data, especially when working with data frames with many groups.

```
> h2.splt
$G171
  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
7    G171 9/20/2006  8:52  0.00      NA
8    G171 9/20/2006 13:41  2.00   30.93
9    G171 9/21/2006 12:40  0.00   30.26
...
$G185
    reactor      date  time  vol conc.h2
127    G185 9/18/2006 11:12  0.0      NA
128    G185 9/18/2006 14:00  0.0      NA
129    G185 9/19/2006  9:26 19.6      NA
130    G185 9/19/2006 12:51 18.3      NA
131    G185 9/19/2006 16:00 16.3      NA
132    G185 9/19/2006 22:52 23.0      NA
133    G185 9/20/2006  8:52  5.0      NA
134    G185 9/20/2006 13:41  2.1      NA
135    G185 9/21/2006 12:40  6.2      NA
```

If you apply `split` to individual vectors, the resulting list can be used directly in some plotting or summarizing functions to give you results for each separate group. (There are usually other ways to arrive at this type of result.) The `split` function can also be handy for manipulating and analyzing data by some grouping variable, as we will see later.

It is often necessary to sort data. For a single vector, this is done with the function `sort`.

```
> x<-rnorm(5)
> x
[1]  0.08396616  0.60914099  0.93206451 -1.29747244  0.87950418
> y<-sort(x)
> y
[1] -1.29747244  0.08396616  0.60914099  0.87950418  0.93206451
```

But what if you want to sort an entire data frame by one column? In this case it is necessary to use the function `order`, in combination with indexing.

```
> h2.dat[order(h2.dat$vol),]
> h2.dat[order(h2.dat$vol),]
    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
...
108    G182 9/21/2006 12:40  1.50   26.49
63     G177 9/21/2006 12:40  1.70      NA
62     G177 9/20/2006 13:41  1.85      NA
...
111    G183 9/19/2006  9:26 25.20      NA
24     G173 9/19/2006 22:52 25.50   24.24
33     G174 9/19/2006 22:52 26.60   22.38
54     G176 9/21/2006 12:40    NA      NA
```

The function `order` returns a vector that contains the row positions of the ranked data:

```
> order(h2.dat$vol)
  [1]   1   2   4   5   6   7   9  10  11  19  20  27  28
...
```

Note 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 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 resides in its 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 specific values reside in a data structure, while the `which` function will return the locations of values that meet certain criteria.

```
> match(1.85,h2.dat$vol)
[1] 62
```

Note that this function matches the first observation only. This function is vectorized.

```
> match(c(1.85,2.0,25.5),h2.dat$vol)
[1] 62  8 24
```

Let's check this result:

```
> h2.dat[c(62,8,24),]
   reactor      date  time   vol conc.h2
62    G177 9/20/2006 13:41  1.85      NA
8      G171 9/20/2006 13:41  2.00   30.93
24    G173 9/19/2006 22:52 25.50   24.24
```

The `match` function is useful for finding the location of the unique values, such as the maximum.

```
> match(max(h2.dat$vol),h2.dat$vol)
[1] 54
```

Let's take a look at the value.

```
> h2.dat$vol[54]
[1] NA
```

Oops. Try again.

```
> match(max(na.omit(h2.dat$vol)),h2.dat$vol)
[1] 33
> h2.dat$vol[33]
[1] 26.6
```

The `which` function, on the other hand, will return all locations that meet the criteria.

```
> which(h2.dat$vol<1)
 [1]   1   2   4   5   6   7   9  10  11  19  20  27  28
[14]  29  36  37  38  46  47  55  56  57  59  60  61  64
[27]  65  66  68  69  70  71  73  74  75  77  78  79  81
[40]  82  83  91  92 100 101 109 110 118 119 126 127 128
```

Of course, you can specify multiple constraints.

```
> which(h2.dat$vol<10 & h2.dat$vol>0)
  8  14  17  18  23  26  35  44  45  50  53  58  62
[14]  63  67  72  76  80  81  88  89  90  97  98  99 106
[27] 107 108 115 116 117 124 125 133 134 135
```

The `which` function can be useful for locating missing values.

```
> which(is.na(h2.dat$vol))
[1] 54
```

## *5.3. Factors*

For many analyses, it is important to distinguish between quantitative (i.e., continuous) and categorical (i.e., discrete) variables. Categorical data are called factors in R. Internally, factors are stored as numeric data (as a check with `mode` will tell you), but they are handled as categorical data in statisitical analyses. Factors are a class of data in R. R automatically

recognizes non-numerical data as factors when data are read in[30], but if numerical 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 specified explicitly. In R, the function `factor` does this.

```
> a<-c(rep(0,4),rep(1,4))
> a
[1] 0 0 0 0 1 1 1 1
> a<-factor(a)
> a
[1] 0 0 0 0 1 1 1 1
Levels: 0 1
```

The levels that R assigns to your factor are by default the unique values given in your original vector. This is often fine, but you may want to assign more meaningful levels. Levels can be specified for a factor using the `levels` function.

```
>levels(a) <- c("F","M")
> a
[1] F F F F M M M M
Levels: F M
```

The order in which a factor is sorted can be important in some cases. For example, say you have a vector that contains height categories.

```
> heights<-c("short","short","tall","medium","medium","tall")
```

If you designate this as a factor, the default levels will be sorted alphabetically.

```
> height.1<-factor(heights)
> height.1
[1] short  short  tall   medium medium tall
Levels: medium short tall

> as.numeric(height.1)
[1] 2 2 3 1 1 3
```

If you specify `levels` as an argument of the function factor, you can control the order of the levels.

```
> height.2<-factor(heights,levels=c("short","medium","tall"))
> as.numeric(height.2)
[1] 1 1 3 2 2 3
```

This can be useful for obtaining a logical order in statistical output or summaries.

---

[30] Factors take up less storage space than do character data. Automatic conversion of read-in data to factors can be suppressed by specifying `stringsAsFactors=FALSE` when using `read.table`.

Sometimes it is necessary to combine multiple factors to make a new factor that includes all combinations of the original factors. This can be done using a colon (`:`), as described below in the discussion on model formulae.

## *Exercises*

1. Read in the data in US_GDP.txt which contains US gross domestic product (in millions of $) from 1790 to 2009. Using subscripting, return the following: 1) The first 10 rows of the of the colums `year` and `gdp.real`, 2) Nominal GDP (gdp.nom) for the 1800s only.

Now use the subset function to create a new data frame that has only data for the 1800s.

2. Still working with this data set, find the location of the maximum real GDP. Lastly, sort the entire data frame by real GDP.

3. Read in the data in the file Cacti_v_tort.txt. Try creating a subset of this data frame that contains only observations where tortoises were present (where the variable tortoise equals Yes) using the `subset` function.

# 6. Manipulating data, part II

## *6.1. Combining data*

Data frames (or vectors or matrices) often need to be combined for analysis or plotting. Three R functions that are very useful for combining data are `rbind`, `cbind`, and `merge`. The function `rbind` simply "stacks" objects on top of each other to make a new object ("row bind"). The function `cbind` ("column bind") carries out an analogous operation with columns of data. The `merge` function is used to merge data frames by some common variable or variables.

```
> stuff.dat<-data.frame(ID=c("A","B","C"),response=1:3)
> stuff.dat
  ID response
1  A        1
2  B        2
3  C        3

> a.row<-c("C",4)
> more.stuff.dat<-rbind(stuff.dat,a.row)
> more.stuff.dat
  ID response
1  A        1
2  B        2
3  C        3
4  C        4

> a.column<-c(3,1,5)
> wide.stuff.dat<-cbind(stuff.dat,y=a.column)
> wide.stuff.dat
  ID response y
1  A        1 3
2  B        2 1
3  C        3 5
```

To demonstrate the `merge` function, let's read in some data.

```
> pop.dat<-read.table("US_pop.txt",header=TRUE)
> pop.dat
   year       pop
1  1790   3929214
2  1800   5308483
3  1810   7239881
...
21 1990 248709873
22 2000 281421906

> gdp.dat<-read.table("US_GDP.txt",header=T)
> gdp.dat
    year  gdp.nom gdp.real
1   1790      187     4027
```

```
2   1791      204     4268
3   1792      223     4583
...
219 2008 14441400 13312200
220 2009 14256300 12987400
```

To merge these two data frames by year, we can use the following command:

```
> us.dat<-merge(pop.dat,gdp.dat,by="year")
> us.dat
   year       pop gdp.nom gdp.real
1  1790   3929214     187     4027
2  1800   5308483     476     7398
3  1810   7239881     699    10626
...
21 1990 248709873 5800500  8033900
22 2000 281421906 9951500 11226000
```

Notice that `us.dat` does not have rows for all the years that are present in the `gdp.dat` data frame, because most of them didn't have a matching year in `pop.dat`. If you want to keep all rows regardless, use `all=TRUE`.

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 with the same names, but not the variables you want to merge by, R will add extensions to the variable names (`.x` and `.y` by default) in the new data frame. Note that you can merge by multiple variables by specifying, e.g., `by=c("var1", "var2")`. Check out the help file for `merge` for even more options.

## *6.2. Aggregating and summarizing data*

R has some powerful functions for aggregation of data. Some operations could be carried out with multiple functions. In this section, we will start with some simple operations using the `table` function, and then discuss more advanced aggregation before exploring the functions that can be used to carry the aggregation out. The `table` function is handy for summarizing counts of factor data. To demonstrate, let's read in some data from the `ISwR` package.

```
> install.packages("ISwR")
> library(ISwR)
> juul.dat<-juul
> names(juul.dat)
[1] "age"      "menarche" "sex"      "igf1"     "tanner"   "testvol"
```

Let's make sure sex and menarche are factors.

```
> juul.dat$sex<-factor(juul.dat$sex,labels=c("M","F"))
> juul.dat$menarche<-factor(juul.dat$menarche,labels=c("No","Yes"))
```

```
> table(juul.dat$sex)

  M   F
621 713

> table(juul.dat$sex,juul.dat$menarche)

     No Yes
  M   0   0
  F 369 335
```

For more advanced data aggregation, R offers the following functions: `apply`, `lapply`, `aggregate`, `by`, and `tapply` (among others). Spector (2008: 106) presents some handy information on selecting an appropriate function. We will focus on only the most common operations.

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, our data on $H_2$ production by bacteria in a set of laboratory reactors (serum bottles):

```
> h2.dat<-read.table("Biohydrogen.txt",header=TRUE)
> h2.dat
    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
7      G171 9/20/2006  8:52  0.00      NA
8      G171 9/20/2006 13:41  2.00   30.93
9      G171 9/21/2006 12:40  0.00   30.26
...
135    G185 9/21/2006 12:40  6.20      NA
```

Let's say we want to know the total volume of biogas produced by each reactor. The function `tapply` is a good place to start (others would also work). The arguments we need to specify are `X` (the vector we want to summarize), `INDEX` (the grouping variable), and `FUN` (the function we want to apply to each group).

```
> tapply(h2.dat$vol,h2.dat$reactor,FUN = sum)
 G171  G172  G173  G174  G175  G176  G177  G178  G179  G180  G181
13.35 92.55 94.40 95.70 98.30    NA  7.75  6.10  6.50 84.00 88.15
 G182  G183  G184  G185
87.40 91.70 91.10 90.50
```

The function `aggregate` is another good choice here. Its syntax is similar, but note that the `by` argument (analogous to `INDEX` in `tapply`) needs to be a list. This function returns a data frame, so use it if that is what you need.

```
> aggregate(h2.dat$vol,list(reactor=h2.dat$reactor),FUN=sum)
   reactor     x
1     G171 13.35
2     G172 92.55
3     G173 94.40
4     G174 95.70
5     G175 98.30
6     G176    NA
7     G177  7.75
8     G178  6.10
9     G179  6.50
10    G180 84.00
11    G181 88.15
12    G182 87.40
13    G183 91.70
14    G184 91.10
15    G185 90.50
```

What if we need to do something more complicated, like fit a spline, carry out interpolation, or fit a model to each group within a larger data set?

```
> tox.dat<-read.table("Ogeechee_tox_summary.txt",header=TRUE)
> tox.dat
    test n.doc n.ph rep dom.source   ph   doc   lc50
1      2     2    6   1          1 6.25  2.09   9.27
2      3     2    6   1          2 6.22  2.10   6.06
...
126  144    15    8   2          7 8.08 14.60 475.01
```

Let's say we want to fit a regression model to each group of data that has a particular DOM source[31]:

```
> by(tox.dat,tox.dat$dom.source,FUN=function(x)
+ lm(lc50 ~ doc + ph,data=x))
tox.dat$dom.source: 1

Call:
lm(formula = lc50 ~ doc + ph, data = x)

Coefficients:
(Intercept)          doc           ph
```

[31] This example requires you to write your own function, and it introduces the `lm` function for fitting linear models. Both are discussed in later sections.

```
   -1239.47        14.93       178.15

------------------------------------------------------
tox.dat$dom.source: 2

Call:
lm(formula = lc50 ~ doc + ph, data = x)

Coefficients:
(Intercept)          doc           ph
    -705.70        14.38        98.20

------------------------------------------------------
...
```

If you need to process multiple variables at once, you can use `aggregate`, which can handle entire data frames. In this example, we just use part of a data frame to calculate mean pH, DOC concentration, and LC50 for each DOM source.

```
> aggregate(tox.dat[,6:8],list(dom.source=tox.dat$dom.source),mean)
  dom.source       ph      doc     lc50
1          1 7.198333 7.766667 158.8883
2          2 7.156111 8.016667 112.3661
3          3 7.150556 7.785000 105.0778
4          4 7.153889 8.216111 118.4539
5          5 7.140556 8.265556 115.0394
6          6 7.140000 7.897778 138.6472
7          7 7.145000 7.857778 136.1128
```

Let's give this data frame a name for use later on.

```
> tox.summ.dat<-aggregate(tox.dat[,6:8],
+ list(dom.source=tox.dat$dom.source),mean)
```

We will readily admit that using `aggregate`, `by`, and `tapply` can be a bit confusing. If you can't see an advantage of one function over another, there may not be an advantage to using one or the other. The differences between these functions mostly relates to output format (although there are some other important differences). Try one function, and move onto another if needed.

If your data are organized as a list, and you would like to apply some function to each element in the list (e.g., individual vectors or data frames or. . .), use `lapply` or `sapply`. These functions provide a flexible option for some of the more difficult problems: convert your data into a list using `split`, which will divide your data based on some grouping variable, and then use `lapply` or `sapply` to carry out the necessary operation. And, if the operation you want to carry out just seems too complicated for use with one of these or similar functions (e.g., maybe the operation you need to carry out requires multiple steps), you always have the (old school) option of using loops, which are described in a later section.

Let's move on to some simpler material. Sometimes it is necessary to carry out some operation on complete rows or columns within arrays or data frames. The function `apply` can be used for this.

```
> apply(tox.summ.dat,2,range)
     dom.source       ph      doc     lc50
[1,]          1 7.140000 7.766667 105.0778
[2,]          7 7.198333 8.265556 158.8883
```

The above functions are useful for applying a function to specific groups within a data set and producing one result (or set of results) for each group. In some cases, you may want to add a new column to a data frame that contains the result for the group that each row belongs to. Thinking again about the Ogeechee toxicity data, suppose that you want to normalize the DOC concentration by the mean concentration for each DOM source. There are a couple code-intensive ways that you could do this: create a summary with e.g., use `tapply`, and `merge` the result back with the original data frame, or you could apply `split` to the data frame, use `lapply` to apply the required function to each list within the result, and then use `unsplit` to put the pieces back together. However, R also has a single function for this specific purpose: `ave`.

```
> tox.dat$doc.mean<-ave(tox.dat$doc,tox.dat$dom.source,FUN=mean)
> tox.dat
    test n.doc n.ph rep dom.source   ph   doc   lc50 doc.mean
1      2     2    6   1          1 6.25  2.09   9.27 7.766667
2      3     2    6   1          2 6.22  2.10   6.06 8.016667
3      4     2    6   1          3 6.19  2.09   7.88 7.785000
4      5     2    6   1          4 6.23  2.16   8.28 8.216111
5      6     2    6   1          5 6.22  2.27   9.03 8.265556
6      7     2    6   1          6 6.22  2.17   5.42 7.897778
7      8     2    6   1          7 6.25  2.18   9.00 7.857778
8     26     7    6   1          1 6.29  6.57  28.20 7.766667
9     27     7    6   1          2 6.20  6.58  30.64 8.016667
...
```

It is easier to see what we did if we sort the data frame by DOM source.

```
> tox.dat<-tox.dat[order(tox.dat$dom.source),]
> tox.dat
    test n.doc n.ph rep dom.source   ph   doc   lc50 doc.mean
1      2     2    6   1          1 6.25  2.09   9.27 7.766667
8     26     7    6   1          1 6.29  6.57  28.20 7.766667
15    50    15    6   1          1 6.20 14.65  56.87 7.766667
...
2      3     2    6   1          2 6.22  2.10   6.06 8.016667
9     27     7    6   1          2 6.20  6.58  30.64 8.016667
16    51    15    6   1          2 6.00 14.63  38.95 8.016667
...
3      4     2    6   1          3 6.19  2.09   7.88 7.785000
10    28     7    6   1          3 6.22  6.53  33.72 7.785000
17    52    15    6   1          3 5.97 14.33  59.94 7.785000
```

```
...
```

Now we can normalize the DOC concentration by this new column of means:

```
> tox.dat$doc.norm<-tox.dat$doc/tox.dat$doc.mean
```

Of course, it isn't necessary to follow these exact steps. For example, we could do this all in one step by writing out own function, which is covered in a later section:

```
> tox.dat$doc.norm.2<-ave(tox.dat$doc,tox.dat$dom.source,
+ FUN=function(x) x/mean(x))
```

## *6.3. Dates and times*

Dates are handled relatively easily in R. R has a data class called `Dates`, which are represented as the number of days since the beginning of 1970 (generally as integer data). Date data are read in as character data. To convert them to `Date` data, you can use the function `as.Date`.

```
>flow.dat<-read.table("River_flow.txt",header=TRUE)

> flow.dat$date[1:20]
 [1] 2006-01-01 2006-01-02 2006-01-03 2006-01-04 2006-01-05 2006-01-06
 [7] 2006-01-07 2006-01-08 2006-01-09 2006-01-10 2006-01-11 2006-01-12
[13] 2006-01-13 2006-01-14 2006-01-15 2006-01-16 2006-01-17 2006-01-18
[19] 2006-01-19 2006-01-20
365 Levels: 2006-01-01 2006-01-02 2006-01-03 2006-01-04 ... 2006-12-31

> as.Date(flow.dat$date[1:20])
 [1] "2006-01-01" "2006-01-02" "2006-01-03" "2006-01-04" "2006-01-05"
 [6] "2006-01-06" "2006-01-07" "2006-01-08" "2006-01-09" "2006-01-10"
[11] "2006-01-11" "2006-01-12" "2006-01-13" "2006-01-14" "2006-01-15"
[16] "2006-01-16" "2006-01-17" "2006-01-18" "2006-01-19" "2006-01-20"

> some.dates<-as.Date(flow.dat$date[1:20])
```

While these values look like character data, they are not. It is possible to carry out mathematical operations on them now.

```
> class(some.dates)
[1] "Date"

> some.dates - 100
 [1] "2005-09-23" "2005-09-24" "2005-09-25" "2005-09-26" "2005-09-27"
 [6] "2005-09-28" "2005-09-29" "2005-09-30" "2005-10-01" "2005-10-02"
[11] "2005-10-03" "2005-10-04" "2005-10-05" "2005-10-06" "2005-10-07"
[16] "2005-10-08" "2005-10-09" "2005-10-10" "2005-10-11" "2005-10-12"

> mean(some.dates)
[1] "2006-01-10"
```

The default format for dates in R in YYYY-MM-DD (e.g., 2009-12-10 for December 10, 2009), which is represented in R as `"%Y-%m-%d"`. If your dates are in a different format, you need to tell R what that format is when you convert them to a date object.

```
> some.dates<-c("May 01 2008","June 12 2009")
> as.Date(some.dates)
Error in fromchar(x) :
  character string is not in a standard unambiguous format

> as.Date(some.dates,"%b %d %Y")
[1] "2008-05-01" "2009-06-12"
```

R is very flexible in the date formats that it will read in. This makes importing from other programs (e.g. Excel) very easy[32].

```
> other.dates<-c("7/1/1974","7/2/1974","7/3/1974","7/4/1974",
+ "7/5/1974")
> as.Date(other.dates,format="%m/%d/%Y")
[1] "1974-07-01" "1974-07-02" "1974-07-03" "1974-07-04" "1974-07-05"
```

You can find information on the options for specifying the format of dates in the help file for the function `strptime` (this function can be used for converting character data from one date format to another).

Another handy function is `weekdays`, which will return the day of the week for any date or combined date-time.

```
> weekdays(as.Date("1776-07-04"))
[1] "Thursday"
```

R can also handle combined dates and times (although this topic is a bit confusing, and users concerned with second accuracy should consult the relevant help files). There are two basic combined date and time classes in R: `POSIXct` and `POSIXlt`. The first form 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 YYYY-MM-DD, HH:MM:SS, and the time zone. You can use the "`$`" notation to return specific columns.

```
> some.times<-c("1990-01-07 11:10:00 EST","1990-01-07 11:15:00
EST","1990-01-08 22:04:17 EST")
> some.times
[1] "1990-01-07 11:10:00 EST" "1990-01-07 11:15:00 EST"
[3] "1990-01-08 22:04:17 EST"

> as.POSIXct(some.times)
```

[32] If you read in dates from a file and assign the data to a data frame, they will be recognized as a factor by R. To convert to dates, you first need to coerce them to character data using `as.character`.

```
[1] "1990-01-07 11:10:00 EST" "1990-01-07 11:15:00 EST"
[3] "1990-01-08 22:04:17 EST"
```

If we want to subtract one hour:

```
> as.POSIXct(some.times) - 3600
[1] "1990-01-07 10:10:00 EST" "1990-01-07 10:15:00 EST"
[3] "1990-01-08 21:04:17 EST"
```

The default display of the other form (`POSIXlt`) looks similar, but it actually contains separate vectors for seconds, minutes, days, etc.

```
> as.POSIXlt(some.times)
[1] "1990-01-07 11:10:00" "1990-01-07 11:15:00" "1990-01-08 22:04:17"
> as.POSIXlt(some.times)$min
[1] 10 15  4
> as.POSIXlt(some.times)$sec
[1]  0  0 17
```

Here are all the vectors:

```
> names(as.POSIXlt(some.times))
[1] "sec"   "min"   "hour"  "mday"  "mon"   "year"  "wday"  "yday"
[9] "isdst"
```

You can find more information by checking out the help file for `POSIXlt`. It is possible to carry out operations with `POSIXct` and `POSIXlt` objects.

```
> other.times<-c("1990-01-07 22:10:04 EST","1990-01-08 11:22:01
EST","1990-01-14 22:04:00 EST")
> as.POSIXlt(some.times) - as.POSIXct(other.times)
Time differences in hours
[1]  -11.00111  -24.11694 -143.99528
attr(,"tzone")
[1] ""
```

For control over the output units use `difftime`.

```
> difftime(as.POSIXlt(some.times),as.POSIXct(other.times),
+ units="secs")
Time differences in secs
[1]  -39604  -86821 -518383
attr(,"tzone")
[1] ""
```

If you are working with date and time data that are divided up among several columns in a data frame, it is easy to convert this to a `POSIXlt` or `POSIXct` object.

```
> separate.times<-data.frame(month=c("April","April","March"),
```

```
+ day=c(8,9,9),yr=c(2007,2007,2008),hr=c(7,7,9),min=c(10,12,01))
> separate.times
  month day   yr hr min
1 April   8 2007  7  10
2 April   9 2007  7  12
3 March   9 2008  9   1

> comb.times<-paste(separate.times$month,separate.times$day,
+ separate.times$yr,paste(separate.times$hr,separate.times$min,
+ sep=":"))
> comb.times
[1] "April 8 2007 7:10" "April 9 2007 7:12" "March 9 2008 9:1"
```

You can now tell R that these are dates.

```
> as.POSIXlt(comb.times,format="%B %d %Y %H:%M")
[1] "2007-04-08 07:10:00" "2007-04-09 07:12:00" "2008-03-09 09:01:00"
```

A few other useful 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] "2009-02-04 19:26:47 EST"

> system.time(
+ for (i in 1:10000) {
+ rnorm(1000)
+ }
+ )
   user  system elapsed
   8.38    0.00    8.37
```

## *6.4. Reshaping data*

Data with mulipe measurements or variables for each experimental unit (e.g. time series data) present some difficulties that aren't present in simplier data sets. These types of data can be organized into one of two forms, called "wide" and "long" (or "stacked" and "unstacked"). This is probably best demonstrated by example. The file Ave_wind_US.dat contains mean wind monthly wind speeds for cities in the U.S.

```
> wind.dat<-read.table('Ave_wind_US.txt',header=T,sep='\t')

> wind.dat
             location no.yr  jan  feb  mar  apr may jun jul aug sep oct nov dec ann
1 13876BIRMINGHAM AP,AL    65  8.1  8.7  9.0  8.2 6.8 6.0 5.7 5.4 6.3 6.2 7.2 7.7 7.1
2   03856HUNTSVILLE, AL    41  9.0  9.4  9.7  9.2 7.9 6.8 5.9 5.8 6.7 7.2 8.0 8.9 7.9
3       13894MOBILE, AL    60 10.1 10.3 10.5 10.1 8.7 7.5 6.9 6.7 7.7 8.0 8.9 9.6 8.8
4   13895MONTGOMERY, AL    64  7.7  8.2  8.3  7.3 6.1 5.8 5.7 5.2 5.9 5.7 6.5 7.1 6.6
5    26451ANCHORAGE, AK    55  6.4  6.8  7.1  7.3 8.5 8.4 7.3 6.9 6.7 6.7 6.4 6.3 7.1
. . .
```

These data are in the "wide" format, with mean wind speed given in a different column for each month. The `reshape` function can be used to convert a data from wide to long or long to wide.

```
> wind.l.dat<-reshape(wind.dat,direction="long",varying=3:15,
+ v.names='wind',timevar='month',times=c(names(wind.dat)[3:15]))
> wind.l.dat
                    location no.yr month wind id
1.jan 13876BIRMINGHAM AP,AL    65   jan  8.1  1
2.jan   03856HUNTSVILLE, AL    41   jan  9.0  2
3.jan       13894MOBILE, AL    60   jan 10.1  3
4.jan   13895MONTGOMERY, AL    64   jan  7.7  4
5.jan    26451ANCHORAGE, AK    55   jan  6.4  5
. . .
```

## *Exercises*

1. Read in the data on hydrogen production in the file Biohydrogen.txt. The file Reactors.txt contain information on the reactor setup (solution and headspace composition) of the individual reactors. Read in the data in both of these files, and add information on the reactor setup to the data frame on hydrogen production. Your want a new data frame that looks something like this:

```
    reactor solution headspace      date  time   vol conc.h2
1      G171   N-free        N2 9/18/2006 11:12  0.00      NA
2      G171   N-free        N2 9/18/2006 14:00  0.00   31.20
3      G171   N-free        N2 9/19/2006  9:26 11.35   35.22
...
```

2. Still working with the hydrogen data, combine the dates and times and convert them to date/time format. The, calculate the elapsed time for each reactor, i.e. the current date/time minus the initial date time. (Because each reactor has the same starting date/time, you can use the same value for each. If this were not the case, you would have to use something like `ave`. If you are bored, you can try to use `ave` anyway, but note that `ave` will not work with POSIX data.)

3. Read in the data in Eagles.txt, which contains data on the concentration of the chemical alpha-Chlordane (in ng/g) in bald eagle nestling blood. Calculate the mean, sd, and n of alpha-Chlodane concentration by site. Try to organize the results as a data frame, and then write out the data frame to a new file.

4. Calculate your age (as of now) in hours. If you don't know what time you were born, assume it was at noon.

# 7. Exploratory data analysis

*Dalgaard 2008: Chapter 4, Faraway 2005: Chapter 1*

## *7.1. Summary statistics*

Let's demonstrate calculation of summary statistics with one of the soils data sets:

```
> cu.tox.dat<-read.table("Thakali_Cu_EC50s.txt",header=TRUE)
```

Just to see what is in it, first:

```
> names(cu.tox.dat)
[1] "soil"    "ec50.cu" "ph.soil" "oc"      "ph.sol"  "c.cu"    "c.na"
[8] "c.mg"    "c.k"     "c.ca"
```

Here are some useful functions:

```
> mean(cu.tox.dat$ec50.cu)
[1] 192.4727

> median(cu.tox.dat$ec50.cu)
[1] 150.5

> sd(cu.tox.dat$ec50.cu)
[1] 159.3130

> var(cu.tox.dat$ec50.cu)
[1] 25380.62
```

R has a built-in 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[33]. Let's apply `summary` to the first four columns in the `cu.tox.dat` data frame:

```
> summary(cu.tox.dat[,1:4])
          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
```

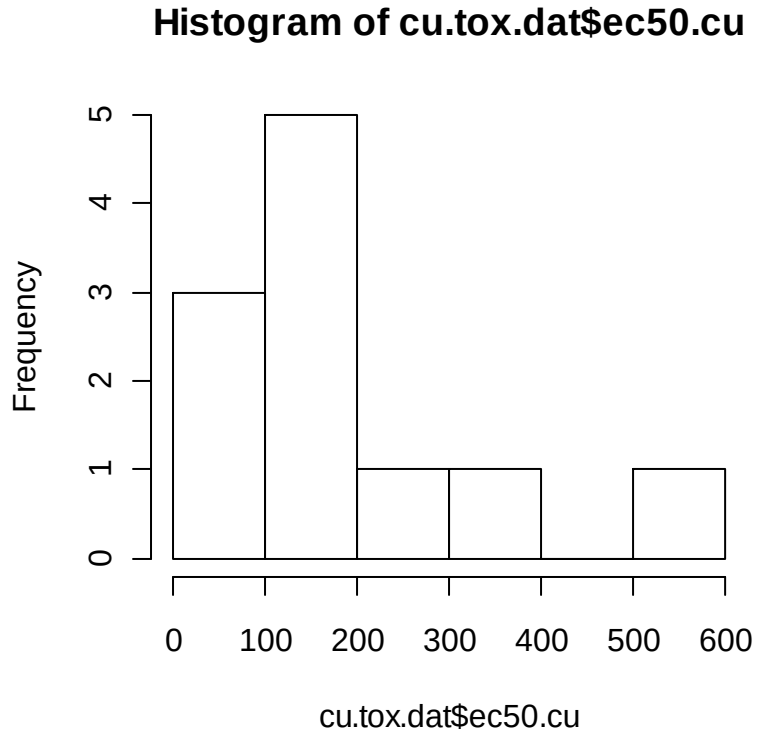
[33] Another use of the `summary` function is in summarizing the results of statistical models, which is discussed in a later section.

Notice the difference between numerical and categorical variables. The `summary` function should probably be your first stop after organizing your data, but before analyzing it—it provides an easy way to check for wildly erroneous values.

## *7.2. Histograms and box plots*

Boxplots and histograms are simple but useful ways of summarizing data. You can generate a histogram in R using the function `hist`.

```
> cu.tox.dat<-read.table("Thakali_Cu_EC50s.txt",header=TRUE)
> hist(cu.tox.dat$ec50.cu)
```




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.

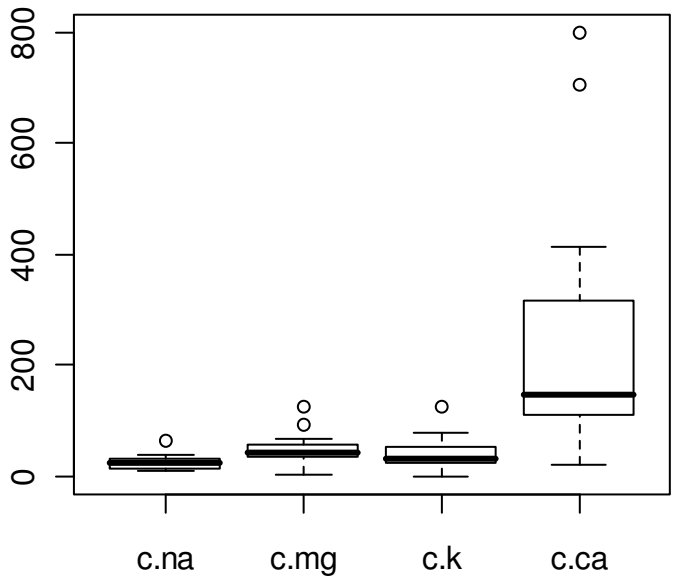
Boxplots are also a good way to summarize data. In the following example, a boxplot is used to compare concentrations of several cations in the soils.

First, let's figure out the order of variables here:

```
> names(cu.tox.dat)
[1] "soil"    "ec50.cu" "ph.soil" "oc"      "ph.sol"
[6] "c.cu"    "c.na"    "c.mg"    "c.k"     "c.ca"
```

And then make the plot.

```
> boxplot(cu.tox.dat[,7:10])
```

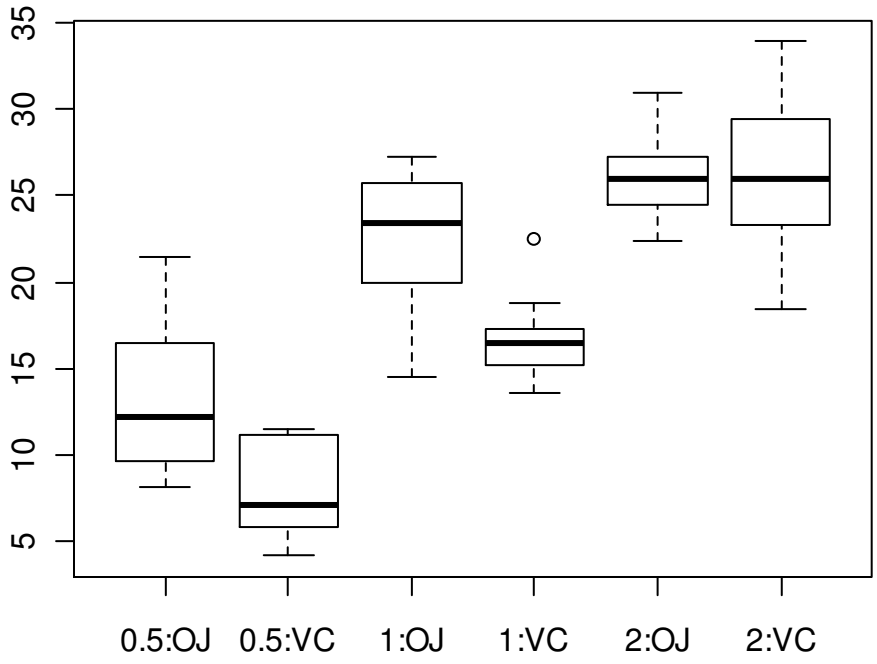



By default, the heavy line shows the median, the box shows the 25th and 75th percentiles, the "whiskers" show the extreme values, and points show any outliers beyond these[34].

Another approach is to plot a single variable by some factor.

```
> tooth.dat<-ToothGrowth
> tooth.dat$trt<-factor(tooth.dat$dose):tooth.dat$supp
> boxplot(len~trt,data=tooth.dat)
> summary(tooth.dat)
      len        supp         dose          trt
 Min.   : 4.20   OJ:30   Min.   :0.500   0.5:OJ:10
 1st Qu.:13.07   VC:30   1st Qu.:0.500   0.5:VC:10
 Median :19.25           Median :1.000   1:OJ  :10
 Mean   :18.81           Mean   :1.167   1:VC  :10
 3rd Qu.:25.27           3rd Qu.:2.000   2:OJ  :10
 Max.   :33.90           Max.   :2.000   2:VC  :10
> boxplot(len~trt,data=tooth.dat)
```

[34] The help file for `boxplot.stats` provides additional information.

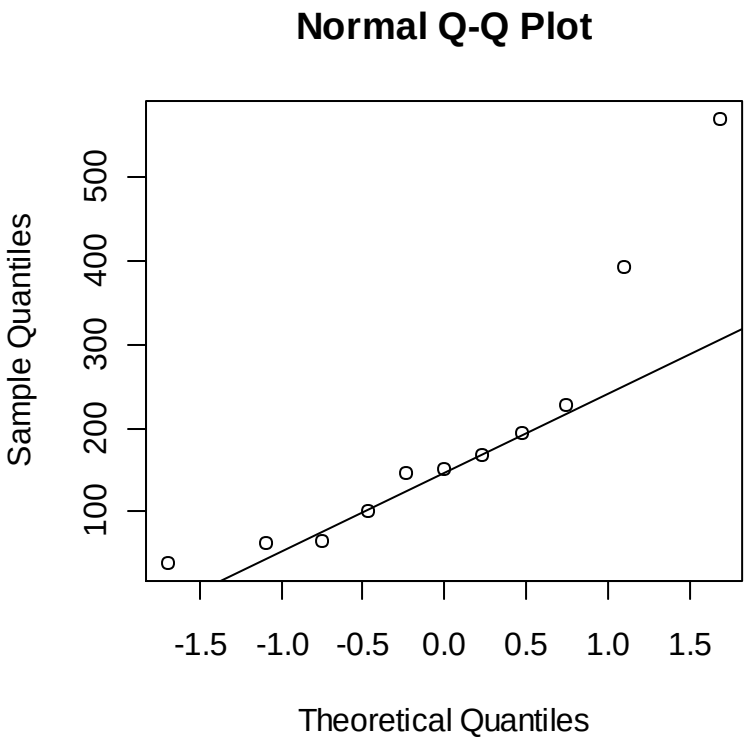



Note the use of the tilde symbol "~" in the above command. The code `len~trt` is analogous to a model formula in this case, and simply indicates that `len` is described by `trt` and should be split up based on the value of this variable. We will see more of this character with the specification of statistical models.

## *7.3. Normal quantile and cumulative probability plots*

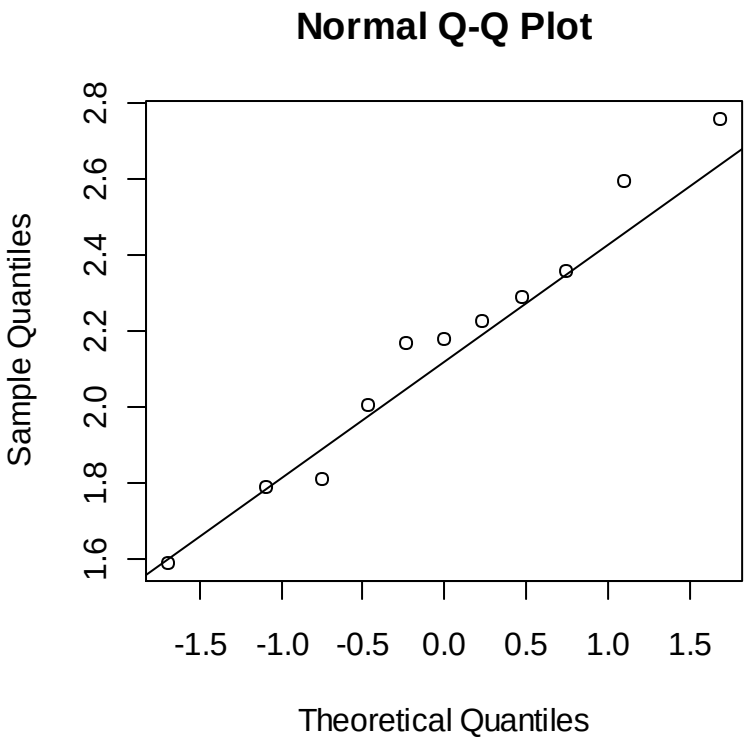
One way to assess the normality of the distribution of a given variable is with a quantile-quantile plot. This plot shows data values vs. quantiles based on a normal distribution (i.e. a $z$ distribution).

```
> qqnorm(cu.tox.dat$ec50.cu)
> qqline(cu.tox.dat$ec50.cu)
```



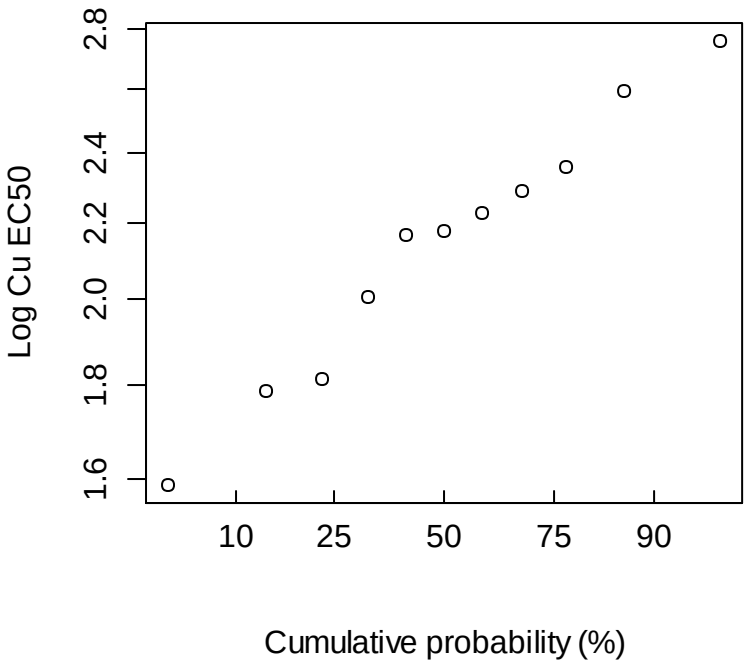

There definitely seems to be some deviation from normality here. A common distribution for toxicity data is log-normal—let's see if this distribution works any better.

```
> cu.tox.dat$l.ec50.cu<-log10(cu.tox.dat$ec50.cu)
> qqnorm(cu.tox.dat$l.ec50.cu)
> qqline(cu.tox.dat$l.ec50.cu)
```




While R does not have a cumulative probability function in its base packages, it 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.

```
> 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.tox.dat$l.ec50)
      0%      25%      50%      75%     100%
1.589950 1.908806 2.177536 2.322487 2.756256

> quantile(cu.tox.dat$l.ec50, 0.05)
      5%
1.688351
```

## *7.4. Dealing with detection limits*

Environmental data may contain observations where a given analyte was not detected, often referred to as nondetect. These data should not be ignored since this would bias your results. Simple solutions such as setting the nondetects to 1/2 of the detection limit is not a satisfactory solution either, since 1/2 of the detection limit may be higher or lower than the true values, and certainly does not include the variability of the true values.

One robust approach to dealing with nondetects is called regression on order statistics, or ROS (Lee & Helsel 2005). The basic approach is to quantify the data distribution of the detected values, and then extrapolate the missing nondetects based on the same distribution. This allows one to calculate unbiased means and standard deviations. In R, this procedure can be done with the function `ros`, which is in the package `NADA`.

The arguments for `ros` are shown below, as given in its help file.

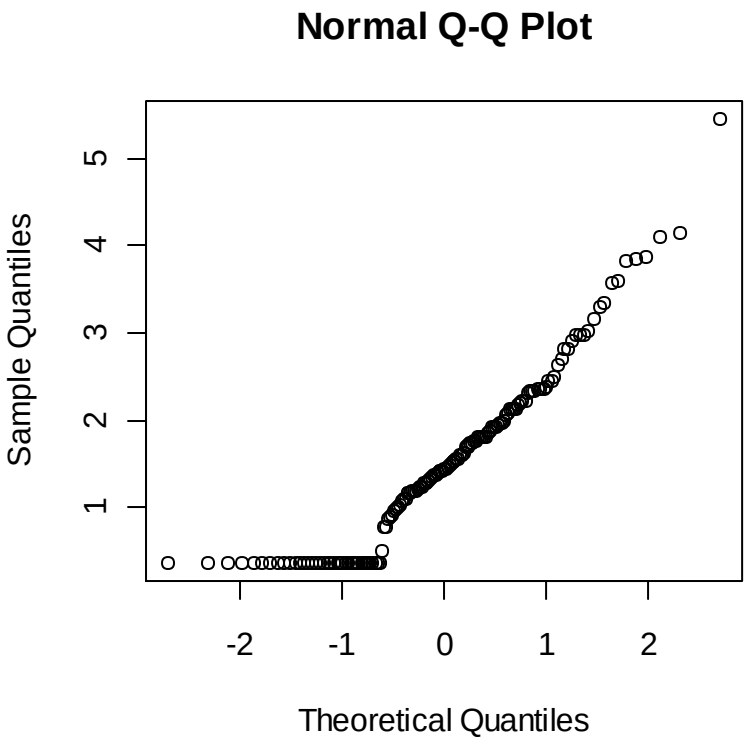
```
ros(obs, censored, forwardT="log", reverseT="exp")
```

The function requires two vectors: the data (with nondetects replaced with the detection limit) (`obs` argument) and a logical vector indicating which values are nondetects (`censored` argument). To demonstrate this procedure, let's use some data on the concentration of the chemical alpha-Chlordane in bald eagle nestling blood.

```
> eagles.dat<-read.table("Eagles.txt",header=T)
> summary(eagles.dat)
          site         achlor
 Mea0.00er  :  9   Min.  :0.375
 CR 306     :  8   1st Qu.:0.375
 Killdeer   :  8   Median :1.440
 Rockwell   :  8   Mean   :1.540
 Ft Seneca  :  7   3rd Qu.:2.130
 Magee Marsh:  7   Max.  :5.450
 (Other)    :100
```

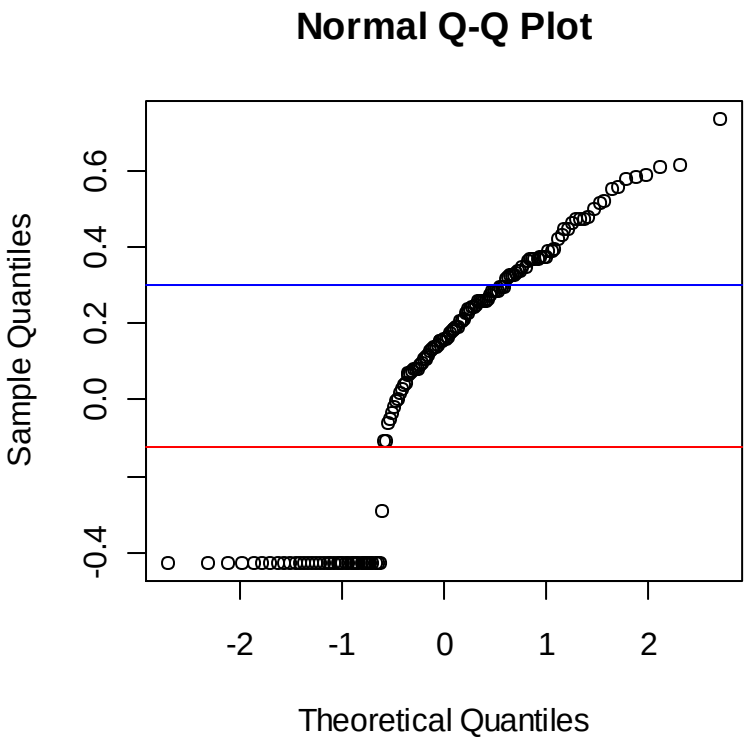
The detection limit in this study was 0.75 ng/g, but the limit of quantification was 2.0 ng/g. In this data set, nondetects are set to ½ of the detection limit. Let's see what these data look like to start out.

```
> qqnorm(eagles.dat$achlor)
```




These data look surprisingly close to normally-distributed. Let's log transform, and add lines at the detection and quantification limits (we will cover the `abline` function in a later section).

```
> qqnorm(log10(eagles.dat$achlor))
> qqnorm(log10(eagles.dat$achlor))
> abline(h=log10(0.75),col="red")
> abline(h=log10(2),col="blue")
```




The `ros` function will make estimates for the values of the observations below the detection limit, and returns the median, mean, and standard deviation of the new, estimated, distribution. To use it, you need to have a logical vector that indicates which values are nondetects. The concentrations for these values should be set to the detection limit. Let's start out using the actual detection limit (as opposed to the quantification limit).

```
> achlor.dat<-data.frame(conc=eagles.dat$achlor,nondetect=F)
> achlor.dat$nondetect[achlor.dat$conc<2]<-T
> achlor.dat$conc[achlor.dat$conc<2]<-2
> achlor.dat
    conc nondetect
1   2.00      TRUE
2   2.00      TRUE
3   2.00      TRUE
4   2.00      TRUE
5   2.00      TRUE
6   2.33     FALSE
7   2.36     FALSE
...
147 4.14     FALSE

> ros(achlor.dat$conc,achlor.dat$nondetect)
         n      n.cen     median       mean         sd
147.000000 106.000000   1.503688   1.698922   0.870380
```

Note that this function by default log transforms data before and after estimating values of the nondetects. This is appropriate for a log-normal distribution, and is relatively robust for distributions that are not log-normal.

We can see the modeled values by using the `data.frame` function.

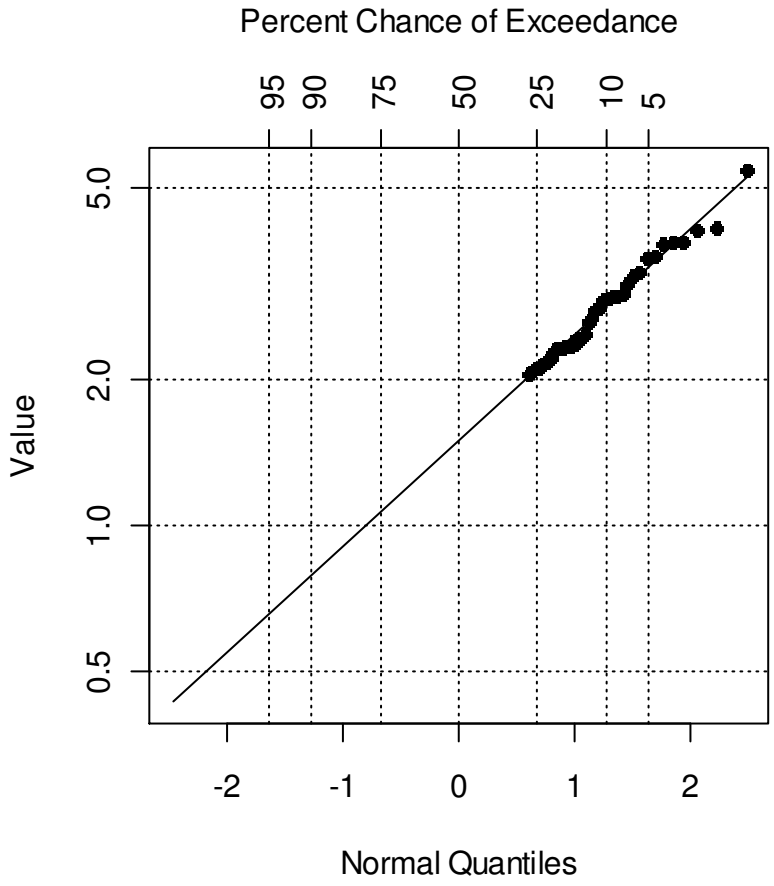
```
> achlor.ros<-ros(achlor.dat$conc,achlor.dat$nondetect)
> achlor.est<-data.frame(achlor.ros)
> achlor.est
   obs censored          pp   modeled
1    2     TRUE 0.006739144 0.4342780
2    2     TRUE 0.013478289 0.4946784
3    2     TRUE 0.020217433 0.5369376
4    2     TRUE 0.026956577 0.5708681
5    2     TRUE 0.033695721 0.5998770
6    2     TRUE 0.040434866 0.6255915
7    2     TRUE 0.047174010 0.6489285
8    2     TRUE 0.053913154 0.6704605
9    2     TRUE 0.060652298 0.6905714
10   2     TRUE 0.067391443 0.7095325
...
```

Of course, the specific estimates are really meaningless; `ros` is only capable of estimating the distribution—the individual modeled values are an intermediate step. But, any parameters estimated from the modeled data are generally unbiased.

```
> 10^mean(log10(achlor.est$modeled))
[1] 1.506938
```

By applying the plot function to `ros` output, you can see the estimated distribution and get a visual representation of the `ros` process.

```
> plot(achlor.ros)
```




## *Exercises*

1. Install and load the `ISwR` package. Check out the help file for the `InsectSprays` data frame. Take a look at the data. Use the `summary` function to summarize the `InsectSprays` data frame. Use `tapply` to apply the `summary` function to each type of spray separately. Generate a boxplot that shows the insect counts as a function of spray type.

2. The file StreamCu.txt contains (generated) data on Cu concentrations in stream water. Values that were below the detection limit already have the detection limit filled in. Use the ros function to estimate the sample median, mean, and standard deviation. Keep in mind the expected distribution of the data. Plot your results.

3. Take a look at the `IgM` data set that is loaded with the `ISwR` package. Make a normal quantile plot for the single variable in it. Are the data closer to a normal or log-normal distribution?

4. Make a cumulative probability plot with the same data used in 3.

# 8. One- and two-sample tests (and the R approach to statistical output)

*Crawley 2007: Chapter 8, R-Intro: Section 8.3, Dalgaard 2008: Chapter 5*

## *8.1.* t *tests*

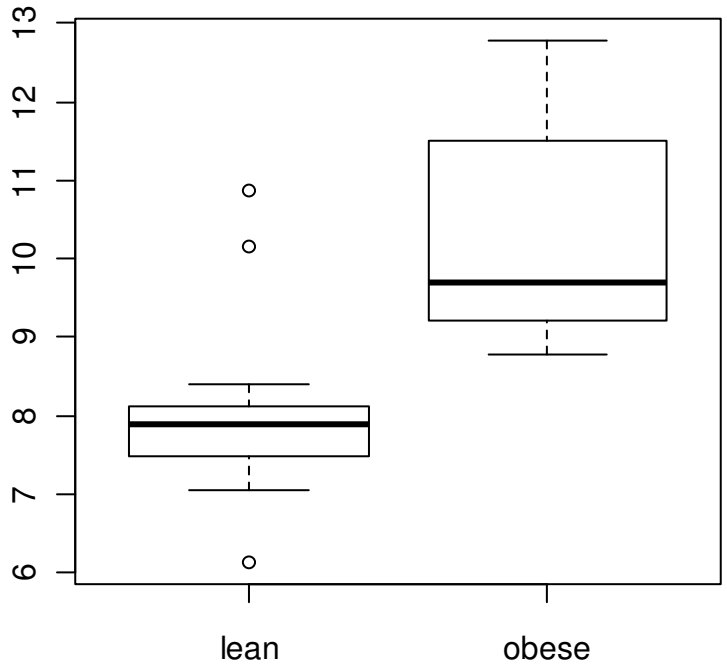
R can be used for one-sample, two-sample, and paired *t* tests. To demonstrate one sample *t* tests, we will use some data from Wilcock et al. (1981) on the measurement of dissolved oxygen (DO) in reference waters using a chemical method called the Winkler method. The objective here is to see if there is a bias in the laboratories' determinations.

```
> DO.dat<-read.table("DO_methods_1.txt",header=TRUE)

> summary(DO.dat)
      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

> t.test(DO.dat$result, mu = 1.2)

        One Sample t-test

data:  DO.dat$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.

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.dat<-energy
> names(energy.dat)
[1] "expend"  "stature"

> boxplot(expend ~ stature, data=energy.dat)
```




```
> t.test(expend ~ stature, data=energy.dat)

        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. We can use the classical approach as well.

```
> t.test(expend ~ stature, data=energy.dat, var.equal=T)

        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.

```
> DO.2.dat<-read.table("DO_methods_2.txt",header=T)
> summary(DO.2.dat)
      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(DO.2.dat$wink, DO.2.dat$elect, paired=T)

        Paired t-test

data:  DO.2.dat$wink and DO.2.dat$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
```

## *8.2. The R approach to statistical output*

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,

```
> t.test.1<-t.test(DO.dat$result,mu=1.2)
> class(t.test.1)
[1] "htest"
```

Objects of class `htest` are lists 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`:

| | |
|---|---|
| `statistic` | the value of the t-statistic. |
| `parameter` | the degrees of freedom for the t-statistic. |
| `p.value` | the p-value for the test. |
| `conf.int` | a confidence interval for the mean appropriate to the specified alternative hypothesis. |

| | |
|---|---|
| `estimate` | the estimated mean or difference in means depending on whether it was a one-sample test or a two-sample test. |
| `null.value` | the specified hypothesized value of the mean or mean difference depending on whether it was a one-sample test or a two-sample test. |
| `alternative` | a character string describing the alternative hypothesis. |
| `method` | a character string indicating what type of t-test was performed. |
| `data.name` | a character string giving the name(s) of the data. |

To demonstrate, say we want *P*-value and the confidence interval:

```
> t.test.1$conf.int
[1] 1.327248 1.618307
attr(,"conf.level")
[1] 0.95

> t.test1$p.value
[1] 0.0005463255
```

To find out what elements are present in a statistical object, you can use the `attributes` function, for example:

```
> attributes(t.test1)
$names
[1] "statistic"   "parameter"   "p.value"     "conf.int"    "estimate"
[6] "null.value"  "alternative" "method"      "data.name"

$class
[1] "htest"
```

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.

## *Exercises*

1. Take a look at the help file for the data frame called `sleep`, which is included in the `datasets` package. Perform a *t* test on these data to determine if the two drugs had different effects on sleep.

2. Extract the *t* statistic, *P*-value, and the confidence interval from the above *t* test, and put them in a new data frame. Note that the confidence interval has two values—see if you can put each one in its own column. Write this data frame out to a text file.

# 9. Classical linear models

*Crawley 2007: Chapater 10; Dalgaard 2008: Chapters 6, 7,& 12; R-Intro 2008: Chapter 11, Faraway 2005*

## *9.1. 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 help file for `lm` lists the following.

```
lm(formula, data, subset, weights, na.action,
   method = "qr", model = TRUE, x = FALSE, y = FALSE, qr = TRUE,
   singular.ok = TRUE, contrasts = NULL, offset, ...)
```

The first argument in the `lm` function call (`formula`) is where you specify the structure of the statistical model. This approach is used in other R functions as well, such as `glm`, `gam`, and others. Venables et al. (2006: 50-51) has a useful list of example formulae—some examples are repeated below. In these examples, the variables `x`, `y`, and `z` are continuous, and `A`, `B`, and `C` are factors.

| | |
|---|---|
| `y ~ x` | Simple linear regression of y on x |
| `y ~ x + z` | Multiple regression of y on x and z |
| `y ~ poly(x,2)` | Second order polynomial regression, using orthogonal polynomials |
| `y ~ x + I(x^2)` | Second order polynomial regression, explicit powers |
| | |
| `y ~ A` | Single factor ANOVA |
| `y ~ A + B` | Two-factor ANOVA |
| `y ~ A + B + A:B` | Two-factor ANOVA with interaction |
| `y ~ A*B` | Two-factor ANOVA with interaction |
| `y ~ A + B + C + A:B + A:C + B:C` | Three-factor ANOVA with secondary interaction term |
| `y ~ (A + B + C)^2` | Three-factor ANOVA with secondary interaction term |
| `y ~ (A + B + C)^2 - B:C` | As above but without B:C interaction |
| `y ~ A + x` | 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`. 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.

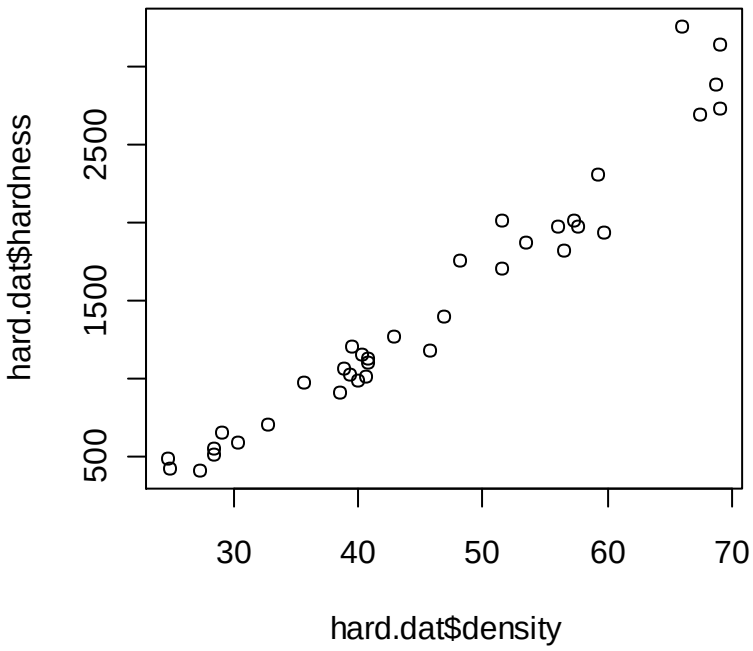
## *9.2. Linear regression*

To demonstrate simple linear regression in R, let's read in a data set on the hardness of some Australian hardwoods.

```
> hard.dat<-read.table("Janka.txt",header=T)
> summary(hard.dat)
    density         hardness
 Min.  :24.70   Min.  : 413.0
 1st Qu.:37.77   1st Qu.: 962.8
 Median :41.80   Median :1195.0
 Mean   :45.73   Mean   :1469.5
 3rd Qu.:56.70   3rd Qu.:1980.0
 Max.  :69.10   Max.  :3260.0
```

Our (alternative) hypothesis here is that density is a good predictor of hardness. Let's start out by seeing what the data look like[35].

```
> plot(hard.dat$density, hard.dat$hardness)
```




[35] We could also have used: `plot(hardness~density,data=hard.dat)`.

This looks like a pretty clear relationship. To fit a linear model, we can use the `lm` function.

```
> mod.1<-lm(hardness ~ density, data=hard.dat)
> mod.1

Call:
lm(formula = hardness ~ density, data=hard.dat)

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.

```
> summary(mod.1)

Call:
lm(formula = hardness ~ density, data = hard.dat)

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
```

Not surprisingly, there is a highly significant relationship here. 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 this list in the help file for `lm`):

| | |
|---|---|
| `coefficients` | a named vector of coefficients |
| `residuals` | the residuals, that is response minus fitted values. |
| `fitted.values` | the fitted mean values. |
| `rank` | the numeric rank of the fitted linear model. |
| `weights` | (only for weighted fits) the specified weights. |
| `df.residual` | the residual degrees of freedom. |
| `call` | the matched call. |
| `terms` | the terms object used. |
| `contrasts` | (only where relevant) the contrasts used. |

| | |
|---|---|
| `xlevels` | (only where relevant) a record of the levels of the factors used in fitting. |
| `offset` | the offset used (missing if none were used). |
| `y` | if requested, the response used. |
| `x` | if requested, the model matrix used. |
| `model` | if requested (the default), the model frame used. |
| `na.action` | (where relevant) information returned by model.frame on the handling of NAs. |

```
> class(mod.1)
[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[36].

```
> mod.1$coeff
(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. (2008: 53-54) 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`.

```
> coef(mod.1)
(Intercept)     density
-1160.49970    57.50667

> resid(mod.1)
          1           2           3           4           5
 224.0848370 161.3341695   3.5674826  44.3101404  76.3101404
...
```

As mentioned above, the `summary` function is a generic function—what it does and what it returns is dependent on the class of its first argument. Here is a list of what's available from the `summary` function for this model[37]:

```
> names(summary(mod.1))
 [1] "call"          "terms"         "residuals"     "coefficients"
 [5] "aliased"       "sigma"         "df"            "r.squared"
 [9] "adj.r.squared" "fstatistic"    "cov.unscaled"
```

---

[36] Remember, you don't need to specify the entire name of the element you are looking for when you use the `$` notation.

[37] Another way to see what various objects contain and how these parts are organized is with the function `str`, which returns the structure of objects.

```
> summary(mod.1)[[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 many model statistics, there are three ways to get your data: an extractor function (such as `coef`), indexing the `lm` object, and indexing the `summary` function. The best approach is to use an extractor function whenever you can[38]. In some cases, the `summary` function will return results that you cannot get by indexing or using extractor functions.

Another function worth mentioning 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. Although we cannot see a any difference here because we are dealing with one predictor, unlike the *t* tests shown when `summary` is applied to an `lm` object, the ANOVA table returned with `anova` show the results of successive deletion tests, starting at the bottom and moving upward, i.e. tests are based on Type I sum of squares (SS)[39].

```
> anova(mod.1)
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
```

Once we have fit a model in R, we can generate predicted values using the `predict` function.

```
> predict(mod.1)
        1         2         3         4         5         6         7
 259.9152  265.6658  409.4325  472.6899  472.6899  507.1939  581.9525
 ...
```

This function 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 back in the original data frame. The `predict` function can also give confidence and prediction intervals.

```
> predict(mod.1,int="conf")
```

[38] These should remain the same in future versions of R, while it is possible that names will change, so indexing may not always work the same.

[39] This is different from the default approach used in other statistical software. More discussion on this topic follows in the section on ANOVA below.

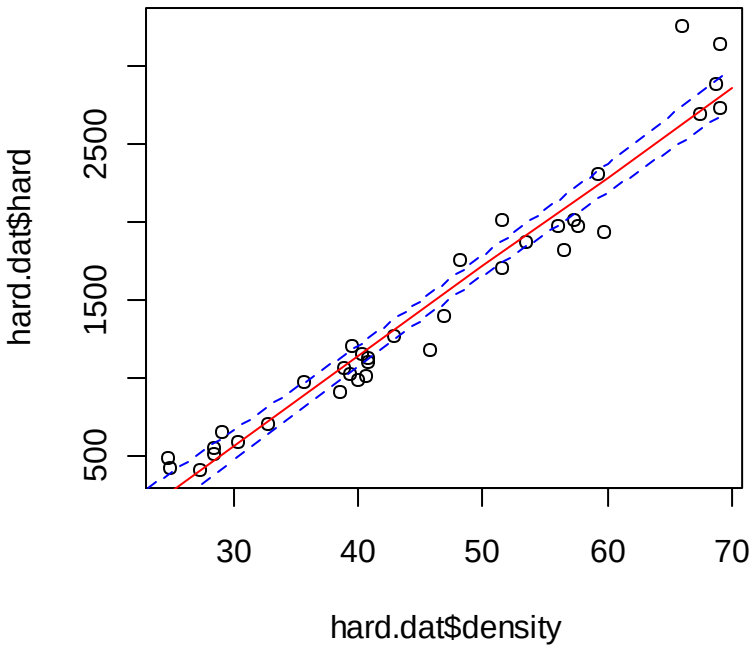
```
         fit       lwr       upr
1   259.9152  144.4580  375.3724
2   265.6658  150.5990  380.7327
3   409.4325  303.9330  514.9320
4   472.6899  371.2673  574.1125
...
```

One problem with plotting these predictions directly 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, let's set up a new data frame just for predictions.

```
> hard.pred.dat<-data.frame(density=density<-seq(10,70,10))
> conf.int<-predict(mod.1,newdata = hard.pred.dat,int="c")
> conf.int
         fit       lwr       upr
1 -585.43296 -762.1332 -408.7327
2  -10.36621 -144.6918  123.9594
3  564.70054  469.0338  660.3673
4 1139.76729 1072.3190 1207.2155
...
```

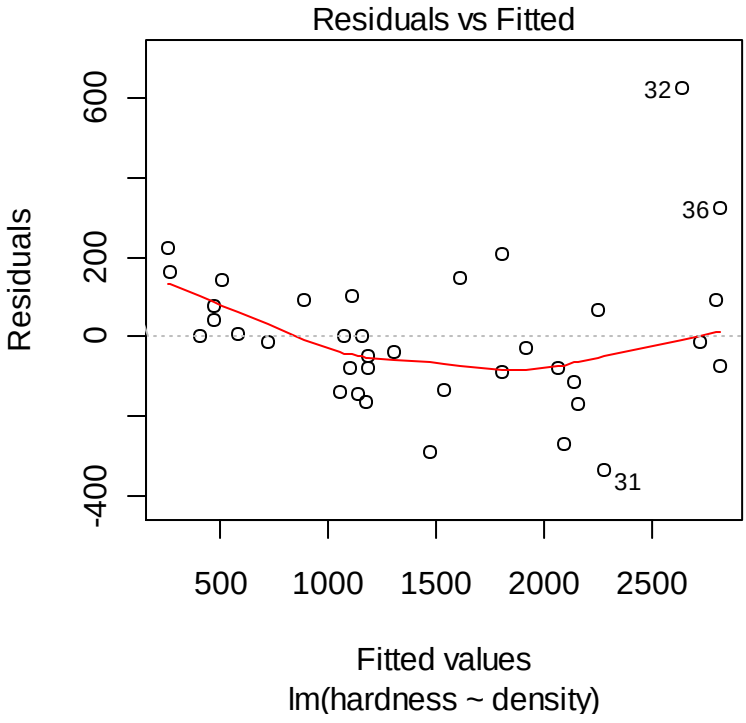
OK, so what is the best way to add these predictions? The function `matlines` will plot the columns in a matrix as a function of a single vector. This works well for plotting confidence intervals and predictions with a single line of code. (Note that the below code does not use the data frame `hard.pred.dat`, but it is used to generate the predictions.)

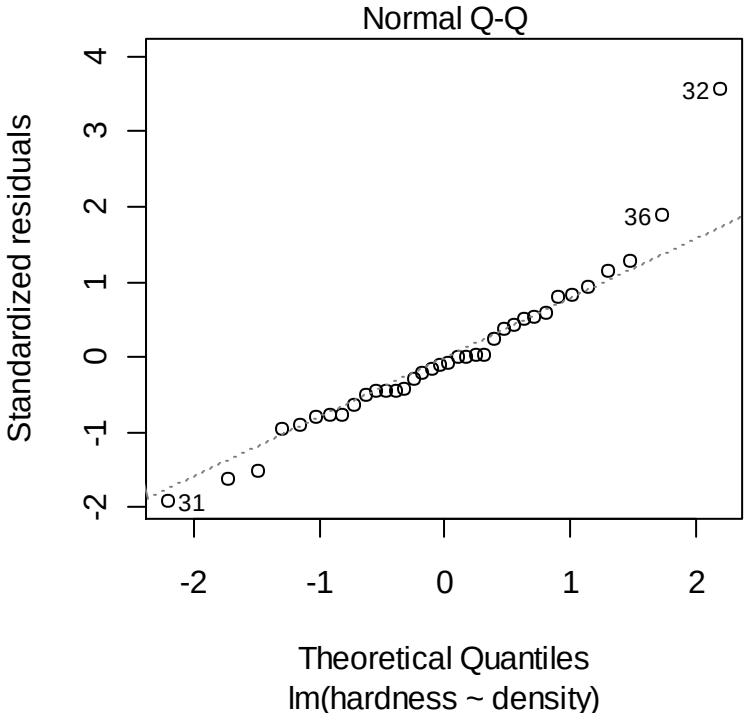
```
> plot(hard.dat$density,hard.dat$hard)
> matlines(density,conf.int,lty=c(1,2,2), col=c("red","blue","blue"))
```

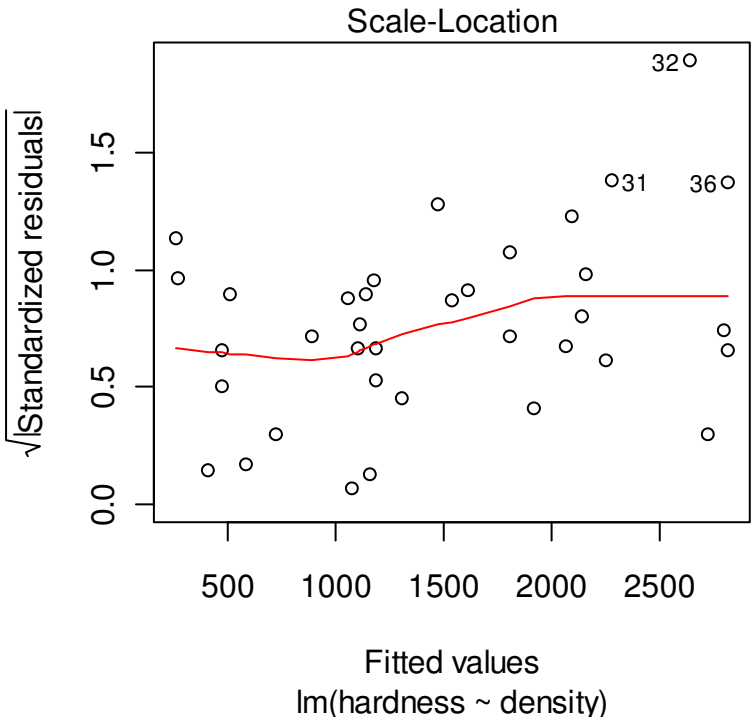


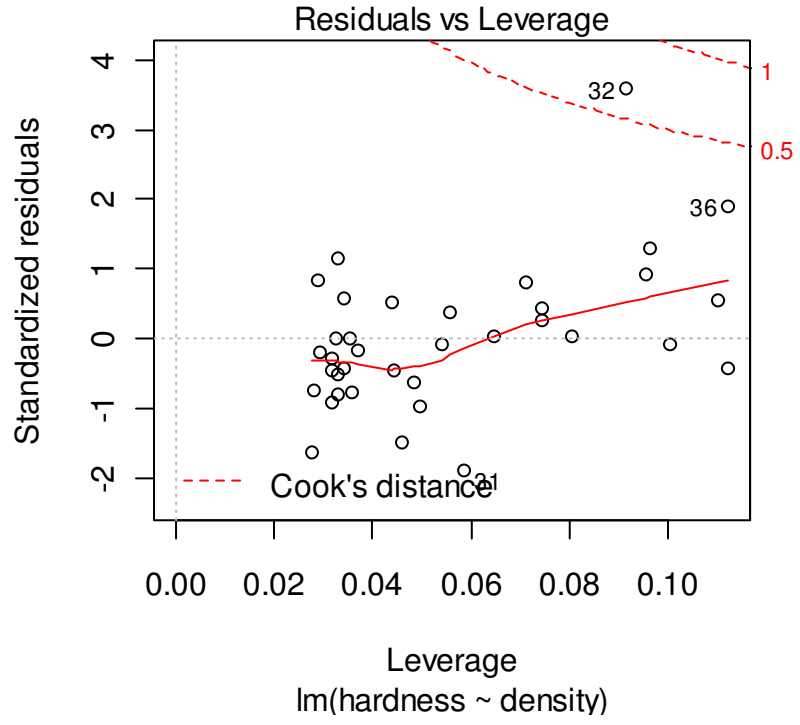

If we wanted to store the predictions in a data frame, we could have done that in one step.

As was mentioned earlier, the plot function does different things for different classes of objects. If we supply to it an `lm` object, we get some useful diagnostic plots.


Residuals vs Fitted
Residuals
600
200
0
-400
32
36
31
500
1000
1500
2000
2500
Fitted values
lm(hardness ~ density)


Normal Q-Q
Standardized residuals
4
3
2
1
0
-1
-2
32
36
31
-2
-1
0
1
2
Theoretical Quantiles
lm(hardness ~ density)


Scale-Location
√|Standardized residuals|
0.0
0.5
1.0
1.5
32
31
36
500
1000
1500
2000
2500
Fitted values
lm(hardness ~ density)


Residuals vs Leverage
Standardized residuals
-2
-1
0
1
2
3
4
1
0.5
32
36
31
Cook's distance
0.00
0.02
0.04
0.06
0.08
0.10
Leverage
lm(hardness ~ density)

To demonstrate multiple linear regression in R, let's use a data set on ozone formation.

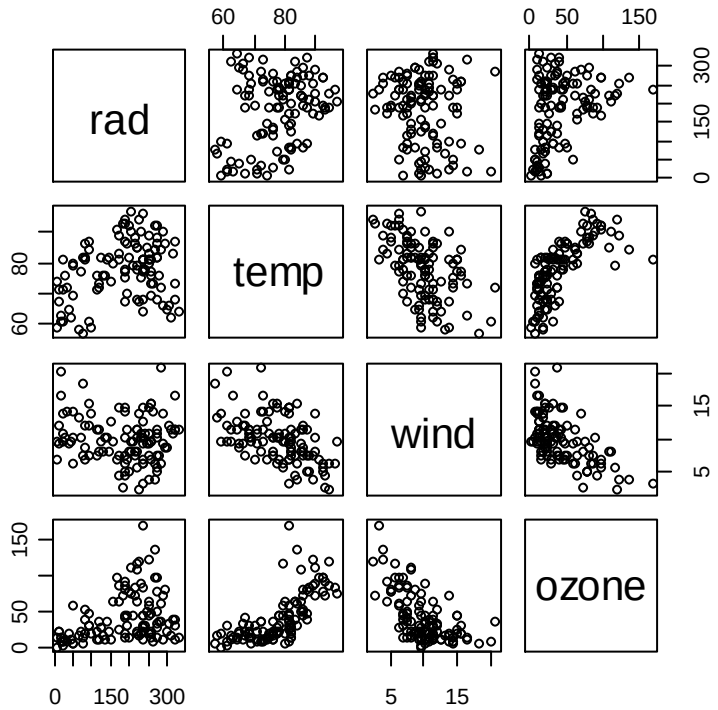
```
> ozone.dat<-read.table("Ozone.txt",header=T)
> summary(ozone.dat)
      rad              temp            wind             ozone       
 Min.   :  7.0   Min.   :57.0   Min.   : 2.300   Min.   :  1.0  
 1st Qu.:113.5   1st Qu.:71.0   1st Qu.: 7.400   1st Qu.: 18.0  
 Median :207.0   Median :79.0   Median : 9.700   Median : 31.0  
 Mean   :184.8   Mean   :77.8   Mean   : 9.939   Mean   : 42.1  
 3rd Qu.:255.5   3rd Qu.:84.5   3rd Qu.:11.500   3rd Qu.: 62.0  
 Max.   :334.0   Max.   :97.0   Max.   :20.700   Max.   :168.0  
```

A quick way to look for relationships between variables in a data frame is with the `cor` function.

```
> cor(ozone.dat)
              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.dat)
```



There certainly seem to be some interesting relationships. Let's fit a model.

```
> mod.1<-lm(ozone~rad + temp + wind, data = ozone.dat)
> summary(mod.1)

Call:
lm(formula = ozone ~ rad + temp + wind, data = ozone.dat)

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).

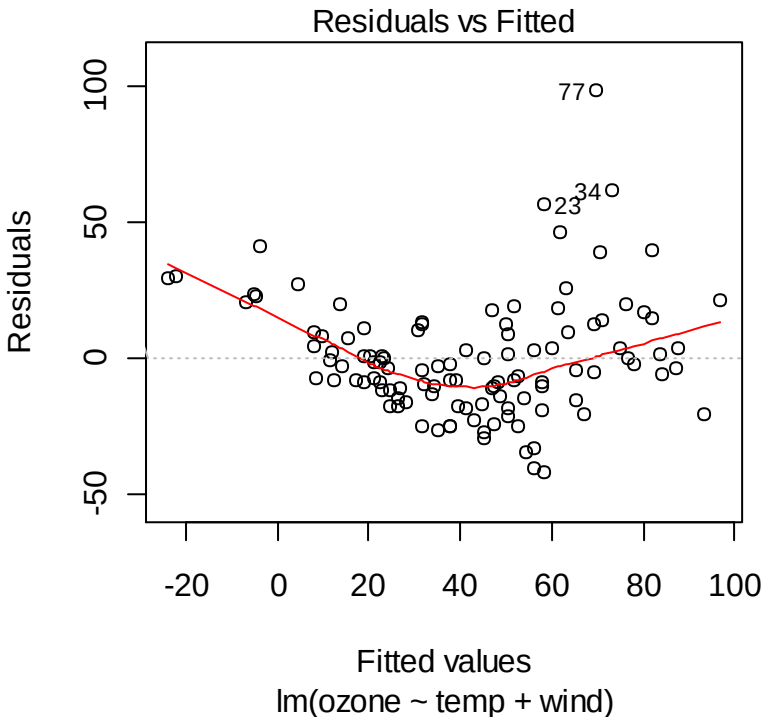
```
> mod.2<-lm(ozone~temp + wind, data = ozone.dat)
> summary(mod.2)

Call:
lm(formula = ozone ~ temp + wind, data = ozone.dat)

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
```

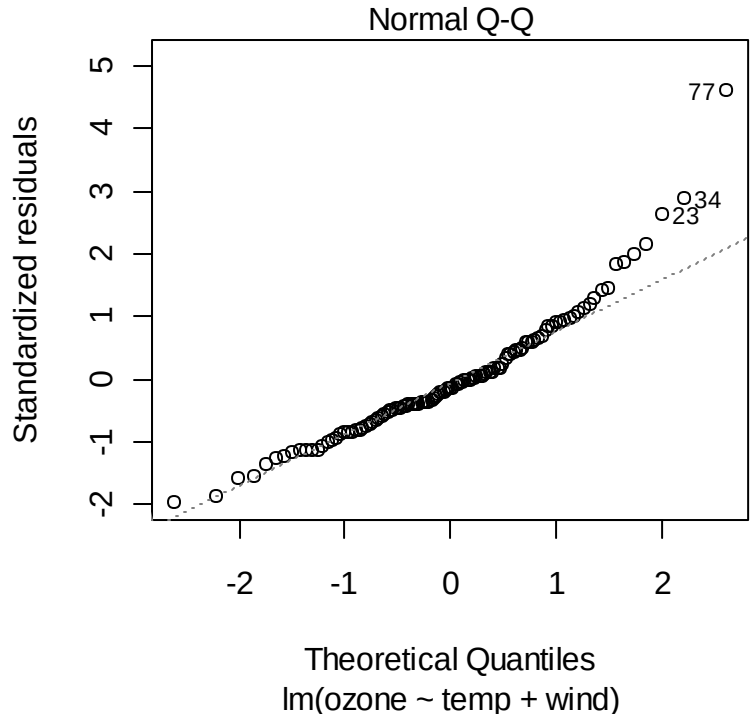
We could also use the `update` function for model 2—this is especially handy for dealing with large model formulas.

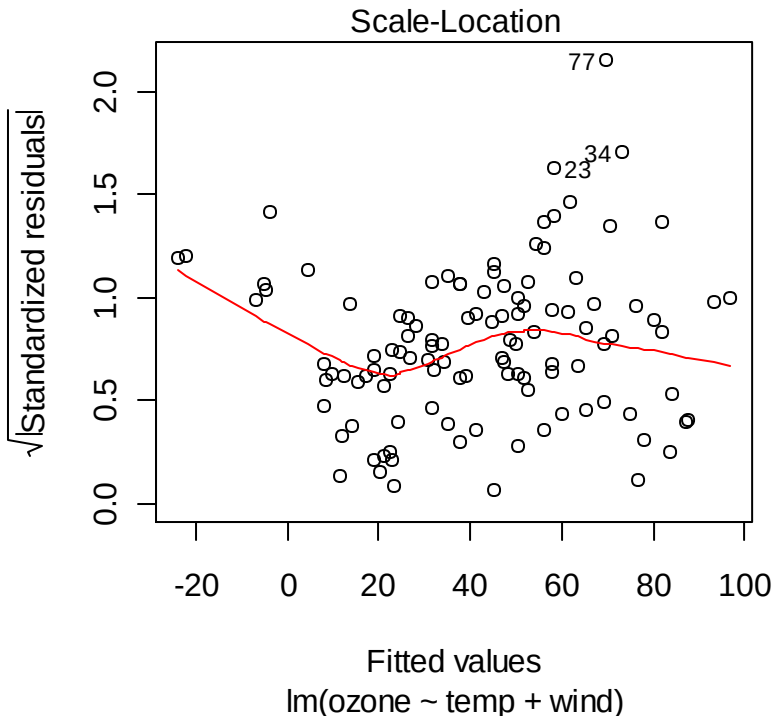
```
> mod.2<-update(mod.1, ~. -rad)
> summary(mod.2)

Call:
lm(formula = ozone ~ temp + wind, data = ozone.dat)

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
```

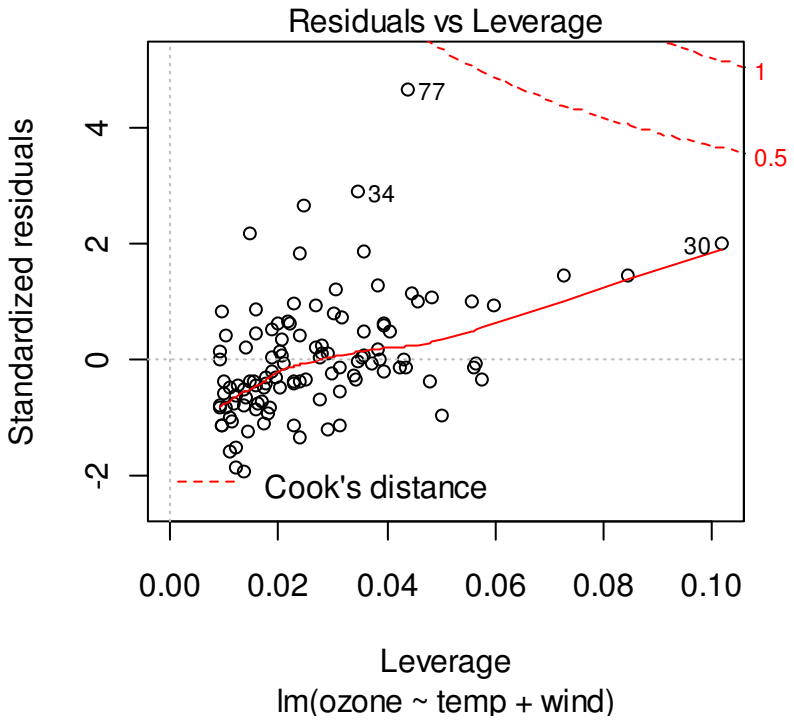
Let's take a look at the results.

```
> plot(mod.2)
```




Normal Q-Q
Standardized residuals
77
34
23
-2
-1
0
1
2
3
4
5
-2
-1
0
1
2
Theoretical Quantiles
lm(ozone ~ temp + wind)


Scale-Location
√|Standardized residuals|
77
34
23
0.0
0.5
1.0
1.5
2.0
-20
0
20
40
60
80
100
Fitted values
lm(ozone ~ temp + wind)




It looks like the residuals are not normally distributed, and there seems to be a relationship to the fiitted values. Since we are attempting to model concentration data, we might consider log-transforming the values[40].

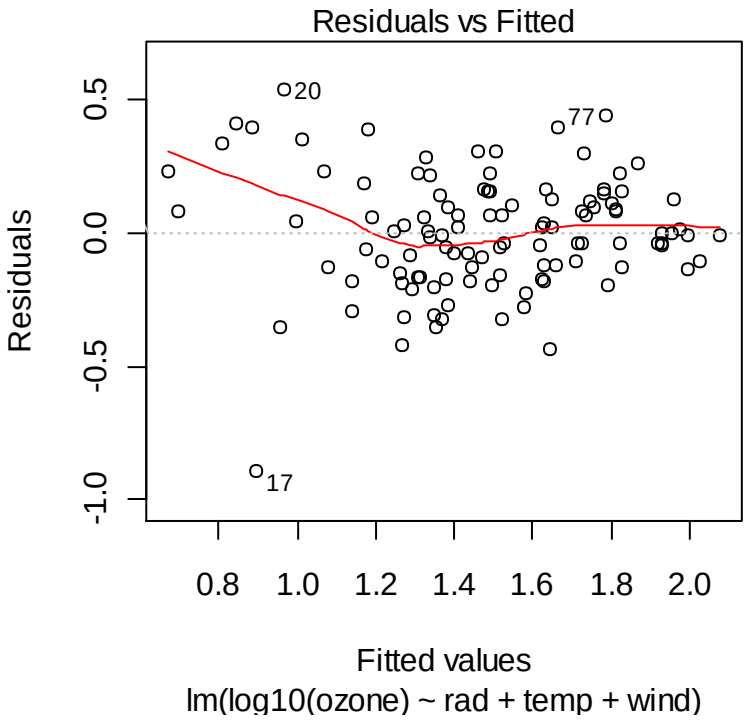
```
> mod.3<-lm(log10(ozone)~rad + temp + wind, data = ozone.dat)
> summary(mod.3)

Call:
lm(formula = log10(ozone) ~ rad + temp + wind, data = ozone.dat)

Residuals:
       Min          1Q      Median          3Q         Max
-0.8955655 -0.1301492 -0.0009698  0.1336213  0.5366669

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 ***
---
```

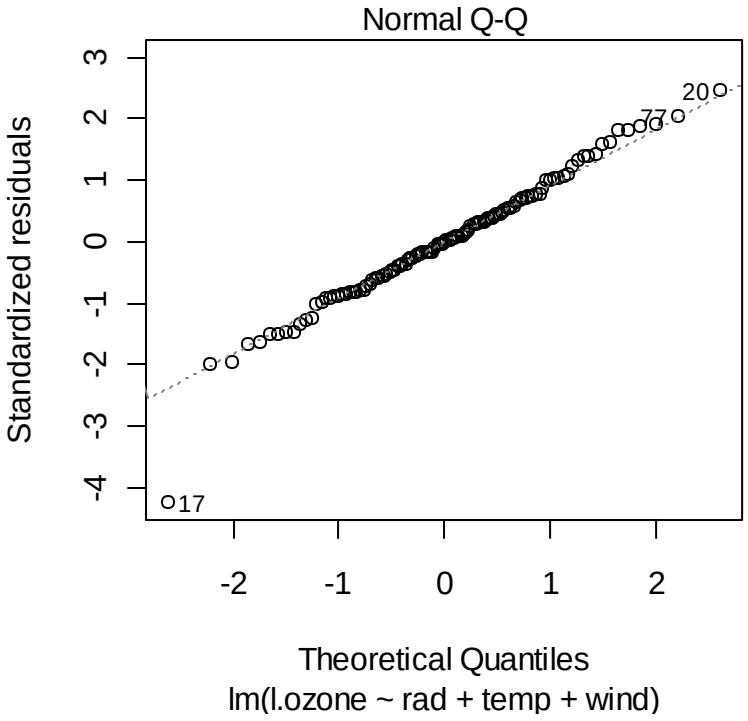
[40] R also has a `transform` function, for transforming variables and maintaining the same variable name.

```
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
```

The first obvious difference is that the *t* value has increased for radiation.

```
> plot(mod.3)
```

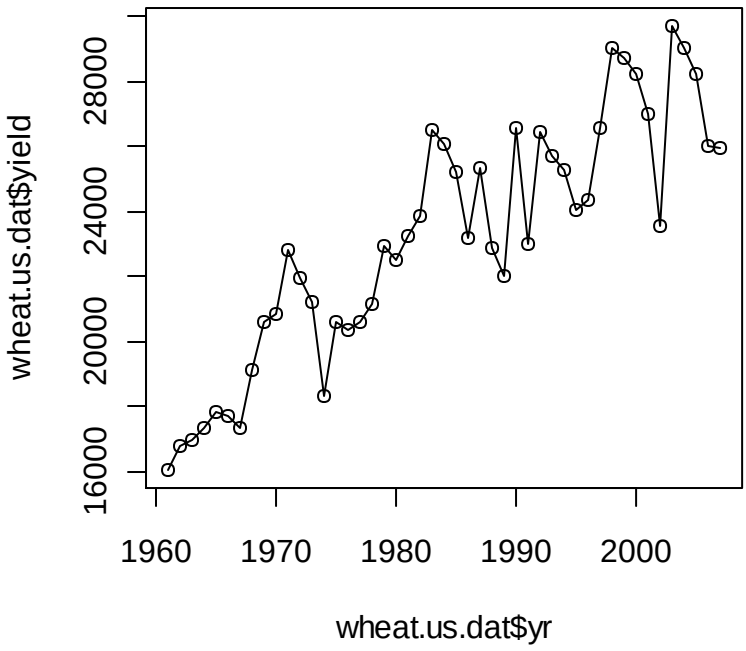





Our residuals look better now as well.

Polynomial regression can be carried out in R using the `lm` function. Let's demonstrate using wheat yield in the United States.

```
> wheat.dat<-read.table("Wheat.txt",header=T)
> summary(wheat.dat)
       yr       country     yield
 Min.  :1961   mx:47   Min.  :16070
 1st Qu.:1972   us:47   1st Qu.:22849
 Median :1984           Median :26748
 Mean   :1984           Mean   :30411
 3rd Qu.:1996           3rd Qu.:39432
 Max.  :2007           Max.  :52274

> wheat.dat$yr.idx<-wheat.dat$yr - 1961
> wheat.us.dat<-subset(wheat.dat,country=="us")

> plot(wheat.us.dat$yr,wheat.us.dat$yield,type="o")
```

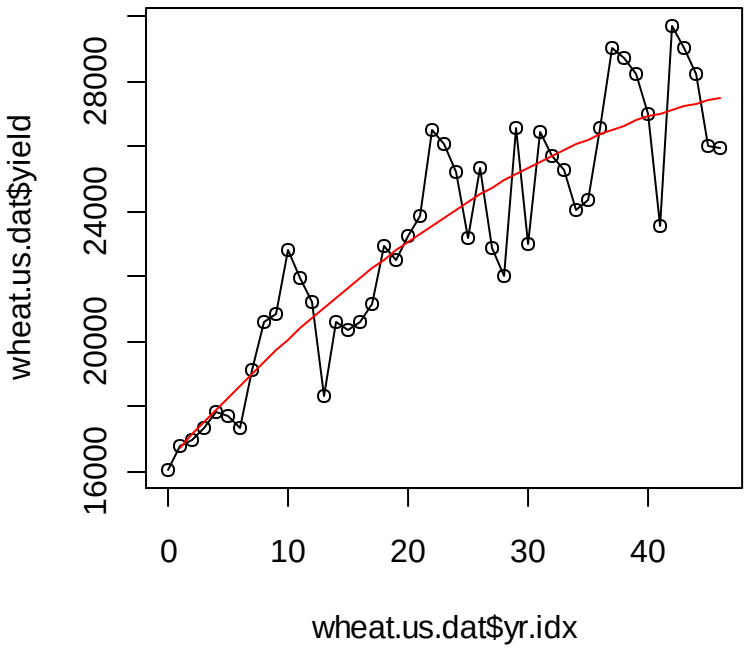



```
> mod.1<-lm(yield~yr.idx + I(yr.idx^2), data=wheat.us.dat)
> summary(mod.1)

Call:
lm(formula = yield ~ yr.idx + I(yr.idx^2), data = wheat.us.dat)

Residuals:
     Min       1Q   Median       3Q      Max
-3474.86 -1215.17    63.42  1128.11  2918.19

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 16339.195    671.988  24.315  < 2e-16 ***
yr.idx        408.997     67.568   6.053 2.82e-07 ***
I(yr.idx^2)    -3.608      1.420  -2.540   0.0147 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 1601 on 44 degrees of freedom
Multiple R-squared: 0.8237,     Adjusted R-squared: 0.8157
F-statistic: 102.8 on 2 and 44 DF,  p-value: < 2.2e-16
```

To use orthogonal polynomial regression, use the `poly` function.

```
> mod.2<-lm(yield~poly(yr.idx,2), data=wheat.us.dat)
```

```
> summary(mod.2)

Call:
lm(formula = yield ~ poly(yr.idx, 2), data = wheat.us.dat)

Residuals:
     Min       1Q   Median       3Q      Max
-3474.86 -1215.17    63.42  1128.11  2918.19

Coefficients:
                 Estimate Std. Error t value Pr(>|t|)
(Intercept)       23173.6      233.6   99.20   <2e-16 ***
poly(yr.idx, 2)1  22600.0     1601.4   14.11   <2e-16 ***
poly(yr.idx, 2)2  -4068.1     1601.4   -2.54   0.0147 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 1601 on 44 degrees of freedom
Multiple R-squared: 0.8237,     Adjusted R-squared: 0.8157
F-statistic: 102.8 on 2 and 44 DF,  p-value: < 2.2e-16

> plot(wheat.us.dat$yr.idx,wheat.us.dat$yield,type="o")
> wheat.pred.dat<-data.frame(yr.idx=0:46)
> wheat.pred.dat$yield.pred<-predict(mod.2,newdata=wheat.pred.dat)
> lines(wheat.pred.dat$yr.idx,wheat.pred.dat$yield.pred,
+ col="red")
```




## *9.3. ANOVA and pairwise comparisons*

To demonstrate ANOVA in R, let's start with a simple data set distributed with the base packages called `InsectSprays`. This dataset shows the effectiveness of six different insecticides.

```
> insects.dat<-InsectSprays
> summary(insects.dat)
     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
```

There are two options for specifying an ANOVA: `lm` and `aov`. Really, `aov` is just a "wrapper" for calling up the `lm` function. 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.

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 (Zar 1999)[41].

```
> insects.dat$sr.count<-sqrt(insects.dat$count + 3/8)
> mod.1<-aov(sr.count ~ spray, data = insects.dat)
> mod.1
Call:
   aov(formula = sr.count ~ spray, data = insects.dat)

Terms:
                    spray Residuals
Sum of Squares  80.52844  22.80262
Deg. of Freedom        5        66

Residual standard error: 0.5877876
Estimated effects may be unbalanced
```

To get more detailed output, we need to use the summary function.

```
> summary(mod.1)
            Df Sum Sq Mean Sq F value    Pr(>F)
spray        5 80.528  16.106  46.616 < 2.2e-16 ***
Residuals   66 22.803   0.345
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

We can specify this same model using the `lm` function.

```
> mod.2<-lm(sr.count ~ spray, data = insects.dat)
> summary(mod.2)

Call:
lm(formula = sr.count ~ spray, data = insects.dat)

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 ***
```

[41] Another option is to forget about `lm` and just use a generalized linear model. This is covered in a later section.

```
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
```

The above output gives you some insight into how R carries out ANOVA—it uses linear regression with dummy variables. You can get more information on the variable coding with the `model.matrix` function, which returns the X-matrix for the regression. From the output, it is clear that spray=A is the reference level.

```
   (Intercept) sprayB sprayC sprayD sprayE sprayF
1            1      0      0      0      0      0
2            1      0      0      0      0      0
...
71           1      0      0      0      0      1
72           1      0      0      0      0      1
attr(,"assign")
[1] 0 1 1 1 1 1
attr(,"contrasts")
attr(,"contrasts")$spray
[1] "contr.treatment"
```

To get the same output that `aov` returns, you can use `anova` on the `lm` object:

```
> anova(mod.2)
Analysis of Variance Table

Response: sr.count
          Df Sum Sq Mean Sq F value    Pr(>F)
spray      5 80.528  16.106  46.616 < 2.2e-16 ***
Residuals 66 22.803   0.345
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

R has many multiple range tests available, including Tukey’s HSD test in the base package, and many others in the `multcomp` package. The Tukey test is applied using the `TukeyHSD` function. Note that this function requires `aov` output (`lm` will not work).

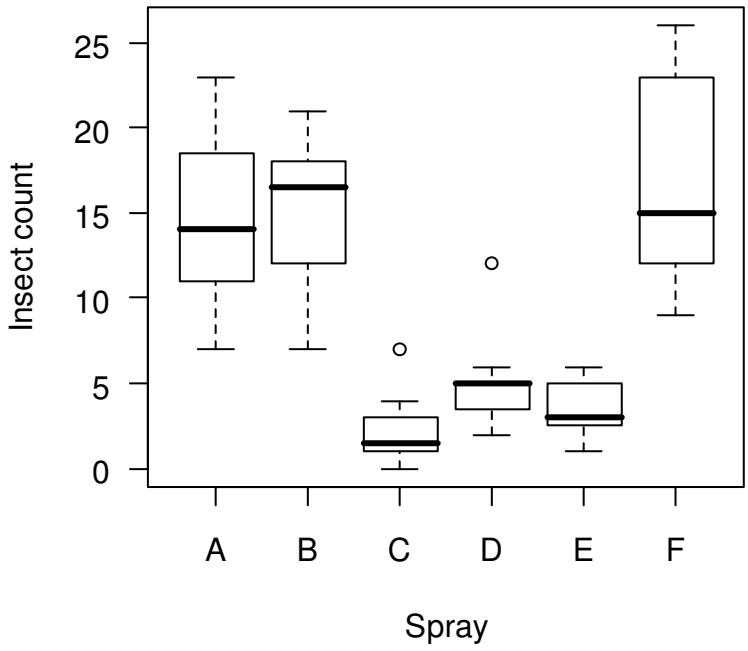
```
> TukeyHSD(mod.1)
  Tukey multiple comparisons of means
    95% family-wise confidence level

Fit: aov(formula = sr.count ~ spray, data = insects.dat)

$spray
          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
```

There are many more options for pairwise comparisons in the `multcomp` package.

To look at ANOVA data, `boxplot` and `barplot` can both be useful.

```
> boxplot(count ~ spray, data=insects.dat,xlab="Spray",ylab="Insect
count",las=1)
```




A two-factor or multi-factor ANOVA is carried out in a similar way. Let's use some data from Zar (1999) on the respiration rate of 3 species of crabs in response to temperature to demonstrate.

```
> crabs.dat<-read.table("Crabs.txt",header=T)
> summary(crabs.dat)
```

```
       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

> crabs.dat$sp<-factor(crabs.dat$sp)
> summary(crabs.dat)
 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. We can check for balance by using the `replications` function.

```
> replications(resp~(sp+temp+sex)^3,data=crabs.dat)
         sp        temp         sex     sp:temp
         24          24          36           8
     sp:sex    temp:sex sp:temp:sex
         12          12           4
```

If the replications function returns a vector (it did), you have a balanced design[42].

Now for the ANOVA.

```
> mod.1<-aov(resp~sp+temp+sex,data=crabs.dat)
> summary(mod.1)
            Df  Sum Sq Mean Sq  F value    Pr(>F)
sp           2  1.8175  0.9088  15.4869 3.057e-06 ***
temp         2 24.6558 12.3279 210.0927 < 2.2e-16 ***
sex          1  0.0089  0.0089   0.1515    0.6984
Residuals   66  3.8728  0.0587
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Let’s include interactions. In R interactions are specified using a colon.

```
> mod.2<-aov(resp~sp+temp+sex+sp:temp+sp:sex+temp:sex,data=crabs.dat)
> summary(mod.2)
```

[42] A test for balance in R is therefore `!is.list(replications(resp~(sp+temp+sex)^3, data=crabs.dat))`—a bit clunky, but it works! See the help file for `replications` for more information.

```
            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
```

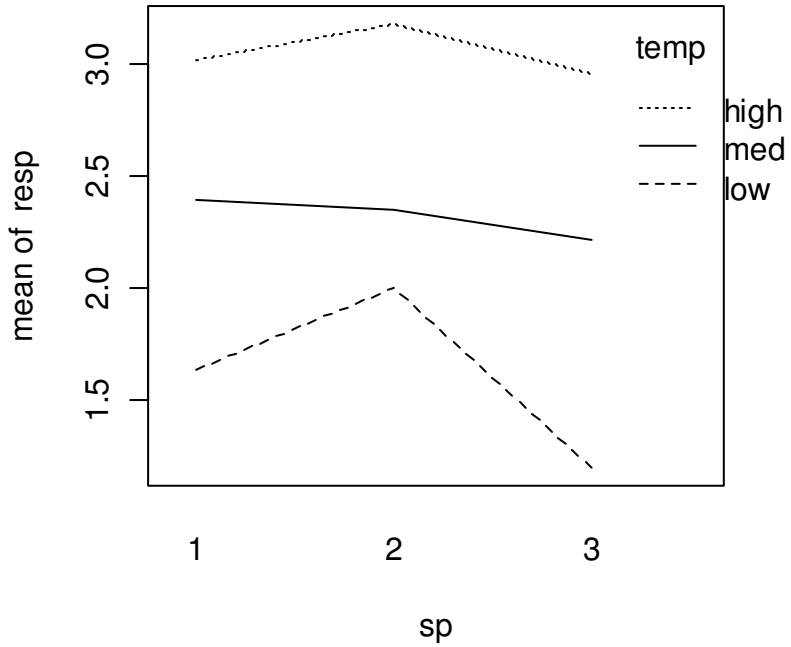
However, R has a trick for specifying all interactions of a certain order.

```
> mod.3<-aov(resp~(sp+temp+sex)^2,data=crabs.dat)
> summary(mod.3)
            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

> mod.4<-aov(resp~(sp+temp+sex)^3,data=crabs.dat)
> summary(mod.4)
            Df  Sum Sq Mean Sq  F value    Pr(>F)
sp           2  1.8175  0.9088  24.4751 2.715e-08 ***
temp         2 24.6558 12.3279 332.0237 < 2.2e-16 ***
sex          1  0.0089  0.0089   0.2394    0.6266
sp:temp      4  1.1017  0.2754   7.4177 7.752e-05 ***
sp:sex       2  0.3703  0.1851   4.9863    0.0103 *
temp:sex     2  0.1753  0.0876   2.3603    0.1041
sp:temp:sex  4  0.2206  0.0551   1.4850    0.2196
Residuals   54  2.0050  0.0371
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

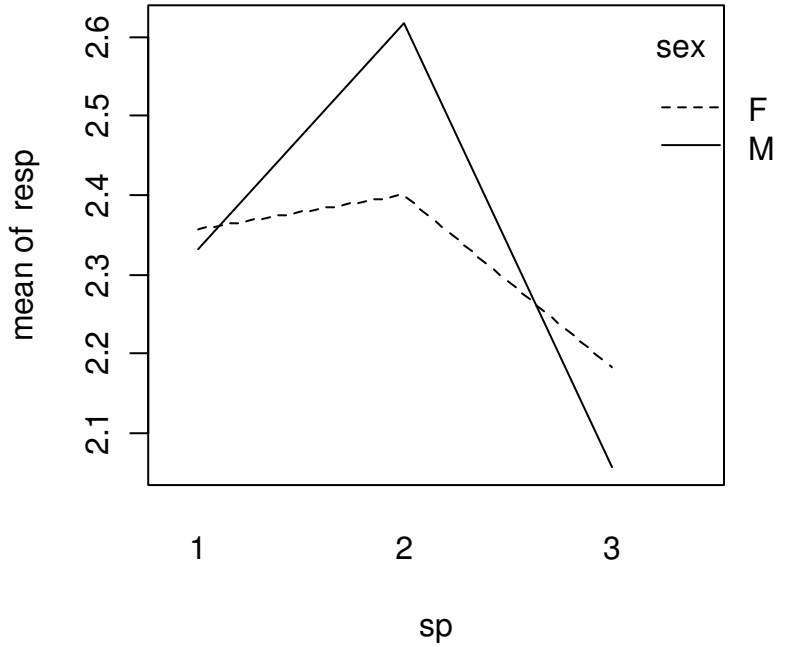
So, our analysis shows that respiration differs with species and temperature, and that the effect of temperature is dependent on the species. Additionally, the difference in respiration rates between sexes is depdent on species (the main effect is actually not significant). What do the results actually look like? We can use the function `interaction.plot` to quickly generate some illuminating plots[43].

[43] A shorter option is `with(crabs.dat,interaction.plot(sp,temp,resp))`.

```
> interaction.plot(crabs.dat$sp,crabs.dat$temp,crabs.dat$resp)
```




```
> interaction.plot(crabs.dat$sp,crabs.dat$sex,crabs.dat$resp)
```



This second interaction plot shows why we don't see a significant main effect for sex (although this test is essentially irrelevant, considering that the interaction is significant).

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"—you can extract or manipulate data in it as with any other list.

```
> model.tables(mod.3,type='means')
Tables of means
Grand mean

2.325

 sp
sp
     1      2      3
2.3458 2.5083 2.1208

 temp
temp
  high    low    med
3.0458 1.6125 2.3167

 sex
sex
     F      M
2.3139 2.3361

 sp:temp
   temp
sp  high  low   med
  1 3.013 1.638 2.388
  2 3.175 2.000 2.350
  3 2.950 1.200 2.213

 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
  high 2.9750 3.1167
  low  1.6000 1.6250
  med  2.3667 2.2667
```

As with regression models, we can also apply the `plot` function to the model output to get diagnostic plots.

```
> plot(mod.4)
...
```

Let's back up a bit and try to get a better understaning of analysis of variance in R. Remember that aov is just a wrapper for lm. So, let's try lm itself.

```
> mod.5<-lm(resp~(sp+temp+sex)^2,data=crabs.dat)
> summary(mod.5)

Call:
lm(formula = resp ~ (sp + temp + sex)^2, data = crabs.dat)

Residuals:
      Min        1Q    Median        3Q       Max
-0.313889 -0.144097 -0.004861  0.135764  0.409722

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)   2.96528    0.08638  34.329  < 2e-16 ***
sp2           0.04167    0.11310   0.368   0.7139
sp3          -0.01250    0.11310  -0.111   0.9124
templow      -1.31667    0.11310 -11.642  < 2e-16 ***
tempmed      -0.50417    0.11310  -4.458 3.85e-05 ***
sexM          0.09444    0.10324   0.915   0.3641
sp2:templow   0.20000    0.13851   1.444   0.1541
sp3:templow  -0.37500    0.13851  -2.707   0.0089 **
sp2:tempmed  -0.20000    0.13851  -1.444   0.1541
sp3:tempmed  -0.11250    0.13851  -0.812   0.4200
sp2:sexM      0.24167    0.11310   2.137   0.0368 *
sp3:sexM     -0.10000    0.11310  -0.884   0.3802
templow:sexM -0.11667    0.11310  -1.032   0.3066
tempmed:sexM -0.24167    0.11310  -2.137   0.0368 *
---
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 a ANOVA table, we can use the `anova` function.

```
> anova(mod.5)
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(...))` 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:

```
> mod.6<-update(mod.5, ~. - temp:sex)
```

Now, we can compare the two models with `anova`.

```
> anova(mod.5,mod.6)
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  2.22556
2     60  2.40083 -2  -0.17528 2.2839 0.1110
```

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.[44]

```
> drop1(mod.5,test="F")
Single term deletions

Model:
resp ~ (sp + temp + sex)^2
         Df Sum of Sq      RSS      AIC F value     Pr(F)
```

[44] And, many R users might say, a shortcoming in 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..

```
<none>                    2.226 -222.319
sp:temp   4     1.102    3.327 -201.366  7.1776 9.136e-05 ***
sp:sex    2     0.370    2.596 -215.239  4.8249   0.01153 *
temp:sex  2     0.175    2.401 -220.861  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.

## *9.4. 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). The experimental design is a 7 × 3 × 3 factorial design, with 7 dissolved organic matter (DOM) sources, 3 DOC concentrations, and 3 pH levels. Copper toxicity (expressed as LC50) was measured for each combination of levels.

```
> lc50.dat<-read.table("Ogeechee_tox_summary.txt",header=TRUE)
```

```
> summary(lc50.dat)
      test            n.doc          n.ph        rep
 Min.   :  2.00   Min.   : 2   Min.   :6   Min.   :1.0
 1st Qu.: 37.25   1st Qu.: 2   1st Qu.:6   1st Qu.:1.0
 Median : 73.00   Median : 7   Median :7   Median :1.5
 Mean   : 73.00   Mean   : 8   Mean   :7   Mean   :1.5
 3rd Qu.:108.75   3rd Qu.:15   3rd Qu.:8   3rd Qu.:2.0
 Max.   :144.00   Max.   :15   Max.   :8   Max.   :2.0
   dom.source       ph             doc
 Min.   :1    Min.   :5.860   Min.   : 2.000
 1st Qu.:2    1st Qu.:6.220   1st Qu.: 2.250
 Median :4    Median :7.240   Median : 6.725
 Mean   :4    Mean   :7.155   Mean   : 7.972
 3rd Qu.:6    3rd Qu.:7.957   3rd Qu.:14.330
 Max.   :7    Max.   :8.400   Max.   :16.130
      lc50
 Min.   :  5.42
 1st Qu.: 23.09
 Median : 64.20
 Mean   :126.37
 3rd Qu.:175.36
 Max.   :574.82
```

We need to make `dom.source` a factor.

```
> lc50.dat$dom.source<-factor(lc50.dat$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 will log transform LC50 and DOC.

```
> lc50.dat$l.doc<-log10(lc50.dat$doc)
> lc50.dat$l.lc50<-log10(lc50.dat$lc50)
```

Let's start with

```
> mod.1<-lm(l.lc50 ~ (dom.source + l.doc + ph)^2, data = lc50.dat)
> summary(mod.1)

Call:
lm(formula = l.lc50 ~ (dom.source + l.doc + ph)^2, data = lc50.dat)

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
l.doc              1.88073    0.22770   8.260 4.97e-13
ph                 0.69521    0.03737  18.603  < 2e-16
dom.source2:l.doc  0.15330    0.08720   1.758 0.081675
dom.source3:l.doc  0.10787    0.08751   1.233 0.220445
dom.source4:l.doc  0.09537    0.08749   1.090 0.278206
dom.source5:l.doc  0.10160    0.08756   1.160 0.248566
dom.source6:l.doc  0.15615    0.08853   1.764 0.080687
dom.source7:l.doc  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
l.doc:ph          -0.13452    0.03060  -4.396 2.66e-05

(Intercept)       ***
dom.source2       .
dom.source3       ***
dom.source4       *
dom.source5       *
dom.source6
dom.source7
```

```
l.doc             ***
ph                ***
dom.source2:l.doc .
dom.source3:l.doc
dom.source4:l.doc
dom.source5:l.doc
dom.source6:l.doc .
dom.source7:l.doc
dom.source2:ph    **
dom.source3:ph    ***
dom.source4:ph    **
dom.source5:ph    **
dom.source6:ph
dom.source7:ph
l.doc:ph          ***
---
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.1)
Analysis of Variance Table

Response: l.lc50
                  Df  Sum Sq Mean Sq   F value    Pr(>F)
dom.source         6  0.1962  0.0327    4.0550 0.0010804 **
l.doc              1 16.6982 16.6982 2070.9118 < 2.2e-16 ***
ph                 1 19.3704 19.3704 2402.3121 < 2.2e-16 ***
dom.source:l.doc   6  0.0306  0.0051    0.6330 0.7035138
dom.source:ph      6  0.2069  0.0345    4.2759 0.0006842 ***
l.doc: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 (i.e. backward elimination):

```
> mod.2<-lm(l.lc50 ~ (dom.source + l.doc + ph)^2 - dom.source:l.doc,
data = lc50.dat)

> anova(mod.2)
Analysis of Variance Table

Response: l.lc50
               Df  Sum Sq Mean Sq   F value    Pr(>F)
dom.source      6  0.1962  0.0327    4.1184 0.0009083 ***
l.doc           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 ***
```

```
l.doc: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
```

Our final analysis says 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(mod.2)

Call:
lm(formula = l.lc50 ~ (dom.source + l.doc + ph)^2 - dom.source:l.doc,
    data = lc50.dat)

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
l.doc           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
l.doc: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(mod.2)
   (Intercept)    dom.source2    dom.source3    dom.source4
   -3.87739361     0.59755833     1.07256669     0.73278355
   dom.source5    dom.source6    dom.source7          l.doc
```

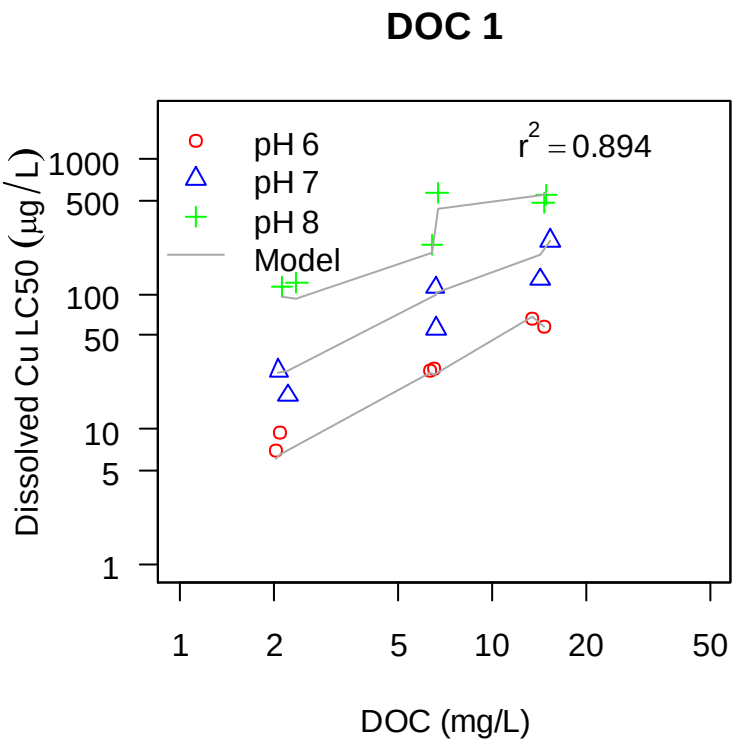
```
    0.69672694     0.18510862     0.38225760     1.99340957
            ph dom.source2:ph dom.source3:ph dom.source4:ph
    0.69204731    -0.10018229    -0.16208067    -0.11256778
dom.source5:ph dom.source6:ph dom.source7:ph       l.doc:ph
   -0.11074601    -0.03244520    -0.05999614    -0.13543281
```

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.

```
> par(mfrow=c(2,2))
> plot(mod.2)

...
```

Here is some more advanced code for plotting these results. One of the resulting seven plots are shown below the code.

```
lc50.dat$l.lc50.pred<-predict(mod.2)
lc50.dat$lc50.pred<-10^lc50.dat$l.lc50.pred
lc50.dat$n.ph<-factor(lc50.dat$n.ph)
n.colors<-c("red","blue","green")
par(ask=TRUE)
for(i in levels(lc50.dat$dom.source)) {
   sub.1<-subset(lc50.dat,dom.source==i)
   plot(1,1,type="n",log='xy',xlim=c(1,50),ylim=c(1,2000),xlab="DOC (mg/L)",
   ylab=expression("Dissolved Cu LC50"~~(mu*g/L)),main=paste("DOC",i),las=1)
   box()
   k<-0
   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(20,1500,substitute(r^2==x,list(x=signif(cor(sub.2$lc50.pred,sub.2$lc50)^2,
        3))))
}
par(ask=FALSE)
```




## *Exercises*

1. If you haven't done so already, install and load the package `faraway`. You should now have the data frame `mammalsleep` in your workspace. Use multiple regression to determine if any ecological or biological predictors appear to be related to the time mammals spend sleeping (`sleep`). Keep a few things in mind: `nondream` and `dream` are alternate response variables, so you can ignore them in your analysis; `danger` is an overall danger index and so would be expected to be correlated with `predation` and `exposure`. You should probably start out by calculating a `summary` and applying the `pairs` or `cor` functions to look for correlations. Produce diagnostic plots after fitting your model. Finally, plot predicted versus observed time spent sleeping.

2. Install and load the MASS package. You should now have a data frame called `cabbages` in your workspace. These data seem to be from a designed experiment to look at the effect of planting date on head mass and ascorbic acid concentration for two different cultivars. Carry out an ANOVA to determine if the cultivar (`Cult`) and planting date (`Date`) had an effect on the height weight of the cabbages. Calculate the means for each group, and plot the data to assess any interaction between the two predictors.

3. Check out the data in the data frame `fruitfly`, which is part of the `faraway` package. Carry out an ANCOVA to determine whether sexual activity has an effect on male fruitfly longevity. As explained in the help file for the data set, thorax length is known to affect longevity, and thus should be included as a covariate. You might compare a model with the covariate to one without

to see what effect inclusion of thorax length has on your results. Plot the data and model predictions (a typical plot for a data set with this structure would have longevity versus thorax length, with different plotting symbols for each of the groups). Check out the diagnostic plots.

4. Read in the data in the file Cactus_width.txt. It contains data on cactus height and width in areas with and without tortoises. Understroy cover is also included as a possible covariate. Carry out an ANCOVA to determine if the height:width ratio of these cacti are related to the presence of tortoises.

# 10. Nonparametric alternatives to *t* tests and ANOVA

*Dalgaard 2008: Chapters 5 & 7*

## *10.1. Wilcoxon signed-rank test*

A nonparametric equivalent to the one-sample *t* test is the Wilcoxon test. Let's apply it to the same data that we used for the *t* test examples.

One-sample test:

```
> wilcox.test(DO.dat$result,mu=1.2)

        Wilcoxon signed rank test with continuity correction

data:  DO.dat$result
V = 1000, p-value = 5.155e-07
alternative hypothesis: true location is not equal to 1.2

Warning messages:
1: In wilcox.test.default(DO.dat$result, mu = 1.2) :
  cannot compute exact p-value with ties
2: In wilcox.test.default(DO.dat$result, mu = 1.2) :
  cannot compute exact p-value with zeroes
```

Two-sample test:

```
> wilcox.test(expend ~ stature, data=energy.dat)

        Wilcoxon rank sum test with continuity correction

data:  expend by stature
W = 12, p-value = 0.002122
alternative hypothesis: true location shift is not equal to 0

Warning message:
In wilcox.test.default(x = c(7.53, 7.48, 8.08, 8.09, 10.15, 8.4,  :
  cannot compute exact p-value with ties
```

Paired test:

```
> wilcox.test (DO.2.dat$wink, DO.2.dat$elect, paired=T)

        Wilcoxon signed rank test with continuity correction

data:  DO.2.dat$wink and DO.2.dat$elect
V = 0, p-value = 0.00903
alternative hypothesis: true location shift is not equal to 0

Warning messages:
1: In wilcox.test.default(DO.2.dat$wink, DO.2.dat$elect, paired = T) :
```

```
  cannot compute exact p-value with ties
2: In wilcox.test.default(DO.2.dat$wink, DO.2.dat$elect, paired = T) :
  cannot compute exact p-value with zeroes
```

## *10.2. Kruskal-Wallis test*

The Kruskal-Wallis test is a nonparametric alternative to one-way ANOVA. Here it is applied to the same data that we use for ANOVA above.

```
> insects.dat<-InsectSprays
> mod.1<-kruskal.test(count ~ spray, data=insects.dat)
> mod.1

        Kruskal-Wallis rank sum test

data:  s.count by spray
Kruskal-Wallis chi-squared = 54.6913, df = 5, p-value = 1.511e-10
```

We can also make nonparametric pairwise comparisons with these data. In R, you can use the pairwise Wilcoxon signed-rank test.

```
> with(insects.dat,pairwise.wilcox.test(count,spray))

        Pairwise comparisons using Wilcoxon rank sum test

data:  count and spray

  A       B       C       D       E
B 1.00000 -       -       -       -
C 0.00051 0.00051 -       -       -
D 0.00062 0.00062 0.01591 -       -
E 0.00051 0.00051 0.26287 0.69778 -
F 1.00000 1.00000 0.00051 0.00062 0.00051

P value adjustment method: holm
There were 15 warnings (use warnings() to see them)
```

## *Excercise*

1. Take a look at the help file for the data frame called `sleep`, which is included in the `datasets` package. Perform a Wilcoxon paired-sample test on these data to determine if the two drugs had different effects on sleep.

# 11. Loops, grouping, and conditional execution

*Dalgaard 2008: Sections 2.3 & 10.2, R-Intro: Chapter 9, R-Lang: Section 3.2*

With R, it is possible to write script files that can be run later, and also to write functions that can 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.

## *11.1. 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.

```
> some.vector<-c(1,3.45,pi,8.1,0,-100)
> for (j in some.vector) print(j)
[1] 1
[1] 3.45
[1] 3.141593
[1] 8.1
[1] 0
[1] -100

> orgs<-c("Tree","Grass","Snail")
> for (i in orgs) print(i)
[1] "Tree"
[1] "Grass"
[1] "Snail"
```

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. (Loops are not the only place where grouping is useful.)

```
> for (i in 1:5) {
+ x<-i^2
+ z<-i^3
+ print(c(i,x,z))
+ }
[1] 1 1 1
[1] 2 4 8
[1]  3  9 27
[1]  4 16 64
[1]   5  25 125
```

Of course, loops are generally used for more complicated (and useful) operations. One common use of loops is to repeat a set of commands for subsets of data. A simple case is to select a different individual row or column within a data frame with each pass:

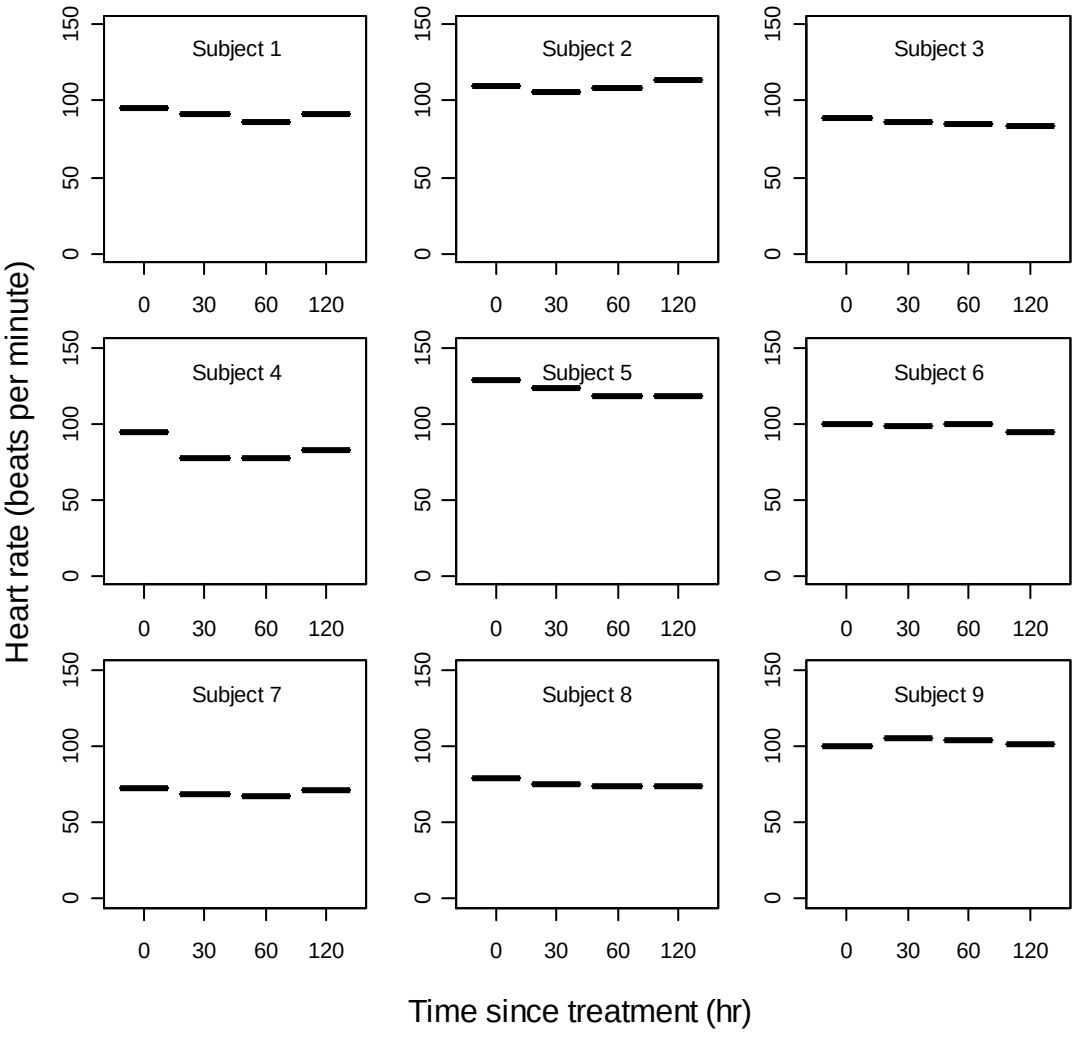
```
> dat<-data.frame(a=letters[1:5],x=rnorm(5))
> dat
  a           x
1 a  0.08667766
2 b -1.15645634
3 c  1.31002675
4 d  0.94639152
5 e  0.40411418

> for(i in 1:5) {
+ print(dat[i,])
+ }
  a          x
1 a 0.08667766
  a         x
2 b -1.156456
  a        x
3 c 1.310027
  a         x
4 d 0.9463915
  a         x
5 e 0.4041142
```

The code in this loop does nothing more than print the selected row to the screen. However, any operation could be carried out on the selected data. Of course, anything that can be done with this approach could be done more efficiently with apply. One place where loops are generally better than other approaches is when you need to carry out a multi-step or otherwise complicated

operation on subsets of data. Producing multi-panel plots is a good example. This type of loop can often make use of the `levels` function, which will return the levels of a factor, or `unique`, which will return the set of unique elements within a vector.

Let's demonstrate this with the data set called `heart.rate` from the ISwR package. In this case, since we want multiple commands to be executed with each pass of the loop, we need to group the commands with braces.

```
> install.packages("ISwR")
> library(ISwR)

> hr.dat<-heart.rate
> names(hr.dat)
[1] "hr"   "subj" "time"

> hr.dat$subj<-factor(hr.dat$subj)

par(mfrow=c(3,3),mar=c(1.5,2,1.5,1.5),oma=c(4,3,2,1))

for (i in levels(hr.dat$subj)) {
   sub.dat<-subset(hr.dat, subj==i)
   plot(hr~time,data=sub.dat,xlab="",ylab="",ylim=c(0,150))
   text(2.5,135,paste("Subject",i))
   box()
}
mtext("Heart rate (beats per minute)",2,1,outer=T)
mtext("Time since treatment (hr)",1,2,outer=T)
```




Another way to do this is with the `split` function.

```
hr.lst<-split(hr.dat,hr.dat$subj)
for (i in levels(hr.dat$subj)) {
   sub.dat<-hr.lst[[i]]
   plot(hr~time,data=sub.dat,xlab="",ylab="",ylim=c(0,150))
   text(2.5,135,paste("Subject",i))
   box()
}
mtext("Heart rate (beats per minute)",2,1,outer=T)
mtext("Time since treatment (hr)",1,2,outer=T)
```

Selecting a different element, row, or column or even a subset within a data frame with each pass of a `for` loop is pretty straightforward.But what if you (for some reason) need to select a different variable with each pass? Let's take a look at wind speed data for several weather stations in the US:

```
> wind.dat<-read.table('Ave_wind_US.txt',header=T,sep='\t')
> names(wind.dat)
 [1] "location" "no.yr"    "jan"      "feb"      "mar"
 [6] "apr"      "may"      "jun"      "jul"      "aug"
[11] "sep"      "oct"      "nov"      "dec"      "ann"
> wind.dat[1,]
               location no.yr jan feb mar apr may jun jul aug
1 13876BIRMINGHAM AP,AL    65 8.1 8.7   9 8.2 6.8   6 5.7 5.4
  sep oct nov dec ann
1 6.3 6.2 7.2 7.7 7.1
```

What if we wanted to select a different month with each pass of a loop, starting with January? One option is to use indexing with numbers:

```
> for(i in 3:15) {
+ print(wind.dat[1,i])
+ }
[1] 8.1
[1] 8.7
[1] 9
[1] 8.2
[1] 6.8
[1] 6
[1] 5.7
[1] 5.4
[1] 6.3
[1] 6.2
[1] 7.2
[1] 7.7
[1] 7.1
```

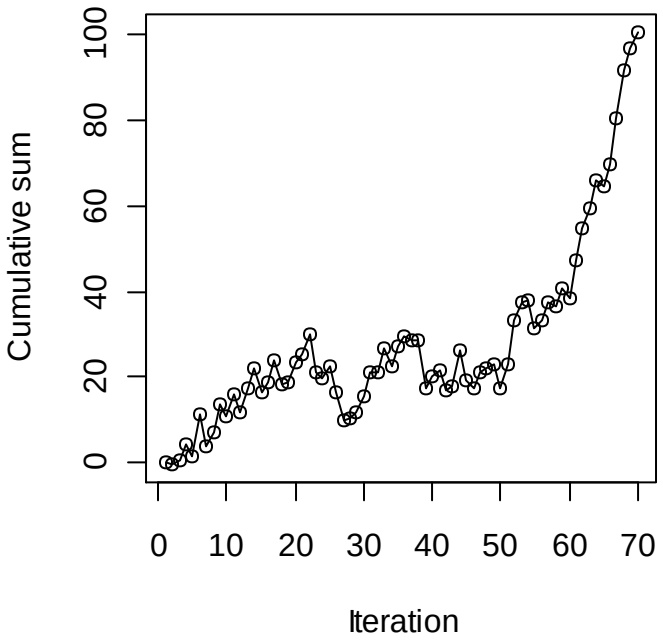
Or, we can work with variable names:

```
> for(i in
c("jan","feb","mar","apr","may","jun","jul","aug","sep","oct","nov",
+ "dec")){
+ print(wind.dat[1,i])
+ }
[1] 8.1
[1] 8.7
[1] 9
[1] 8.2
[1] 6.8
[1] 6
[1] 5.7
[1] 5.4
[1] 6.3
[1] 6.2
[1] 7.2
[1] 7.7
```

There are many other uses of `for` loops—if you have an idea that seems reasonable, there is a good chance that you can get it to work. But, keep in mind that you can often using indexing and functions in place of loops, and end up with cleaner and faster-running code.

The `while` loop is used to repeat a set of commands until some condition is met. While the simple example below doesn't do much, it should give you an idea of how `while` loops can be used.

```
> par(mfrow=c(1,1))

> r.sum<-0
> i<-1

> while (max(r.sum) < 100) {
+ i<-i + 1
+ r.sum[i]<-r.sum[i-1] + rnorm(n=1,mean=1,sd=5)
+ }

> plot(r.sum,xlab="Iteration",ylab="Cumulative sum",type="o")
```




Here is a more complicated (and useful) example that uses both `for` and `while` loops. This code calculates the first 10 roots of the equation

$$b \tan b = L$$

which are required for some analytical mass transfer models.

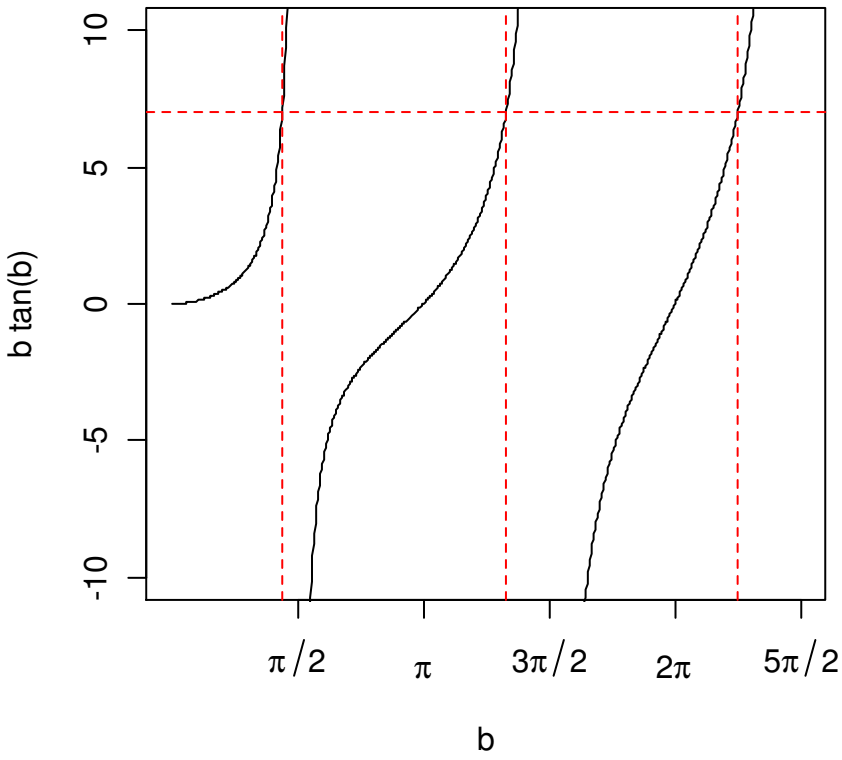
```
> L<-7
> r<-NULL
> for (i in 1:10) {
+ b<-pi/4
+ ct<-0
+ while (abs(log10((b + (i-1)*pi)*tan(b)/L))>1E-5){
+ ct<-ct+1
+ b<-ifelse(ct>1,(atan(L/(b + (i-1)*pi)) + b)/2,atan(L/(b + (i-
+ 1)*pi)))
+ }
+ r[i]<-b + (i-1)*pi
+ }
>
> r
 [1]  1.376618  4.174646  7.064024 10.033909 13.058435 16.117683
 [7] 19.199171 22.295364 25.401635 28.515054
```

It is pretty difficult to keep track of this code if you were to enter it in the GUI directly. We recommend that you keep your R commands for anything more than the most basic analyses in script files. In script files, you should follow good programming practice and use indentation to show the structure of your code:

```
L<-7
r<-NULL
for (i in 1:20) {
   b<-pi/4
   ct<-0
   while (abs(log10((b+(i-1)*pi)*tan(b)/L))>1E-5){
      ct<-ct+1
      b<-ifelse(ct>1,(atan(L/(b+(i-1)*pi))+b)/2,atan(L/(b+(i-1)*pi)))
   }
   r[i]<-b + (i-1)*pi
}
```

Here is a graphical representation of this problem (the plot is only large enough to see the first three roots):

```
> b<-seq(0,2.5*pi,0.01)
> y<-b*tan(b)
> y[y>20|y< -20]<-NA
> plot(b,y,ylim=c(-10,10),type="l",xaxt='n',ylab='b tan(b)')
> axis(1,at=1:5/2*pi,labels=c(expression(pi/2),expression(pi),
+ expression(3*pi/2),expression(2*pi),expression(5*pi/2)))
> abline(v=r,col='red',lty=2)
> abline(h=7,col='red',lty=2)
```




## *11.2. Conditional statements*

Conditional statements are another ubiquitous feature of programming languages. In R, there are two conditional statements, `if...else` and `ifelse`, each designed for a different application.

The `if` construct will execute a command if a specified condition is met.

```
> if (10>3) x<-101
> x
[1] 101

> a<-10
> if (!is.na(a)) print(a)
[1] 10
```

If you would like several commands to be dependent on a single condition, then simply group the commands together using braces.

```
> a<-100
> if (a>3) {
+ print(a)
+ a<-0
```

```
+ }
[1] 100

> a
[1] 0
```

The `if` construct can include an `else`. The command that follows an `else` will be executed if the condition is not met.

```
> grade<-82
> if (grade > 65) result<-"pass" else result<-"fail"
> result
[1] "pass"
```

If you need to group commands using braces, just put the else after (but on the same line as) the closing brace[45].

```
> if (your.grade > 65) {
+ result<-"pass"
+ message<-"Nice job"
+ } else {
+ result<-"fail"
+ message<-"See you next semester"
}

> result
[1] "pass"
```

Of course, you could group multiple commands within nested curly braces.

```
> your.grade<-82
> if (your.grade > 65) {
+ result<-"pass"
+ print("Way to go")
+ } else {
+ result<-"fail"
+ print("See you next semester")
+ }
[1] "Way to go"
```

This is important: `if...else` works only with length-one vectors (i.e. scalars). 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)
> results<-ifelse(grades > 65,"pass","fail")
```

---

[45] Another way to do this is by grouping the entire `if…else` construct.

```
> results
[1] "pass" "fail" "pass" "fail" "pass" "pass" "pass" "pass"
[9] "pass" "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.dat<-read.table("River_flow.txt",header=TRUE)

> names(flow.dat)
[1] "agency"         "site"           "date"           "discharge"
[5] "flag.discharge"

> flow.dat$site<-factor(flow.dat$site)
> levels(flow.dat$site)
[1] "1509000" "4232730"

> flow.dat$name<-ifelse(flow.dat$site==1509000,"Tioughnioga","Seneca")
> flow.dat[1:3,]
  agency    site       date discharge flag.discharge   name
1   USGS 4232730 2006-01-01        75              P Seneca
2   USGS 4232730 2006-01-02       493              P Seneca
3   USGS 4232730 2006-01-03      1380              P Seneca

> flow.dat[500:502,]
    agency    site       date discharge flag.discharge        name
500   USGS 1509000 2006-05-15       302              A Tioughnioga
501   USGS 1509000 2006-05-16       317              A Tioughnioga
502   USGS 1509000 2006-05-17       286              A Tioughnioga
```

Much or all of what can be done using `ifelse` can also be done using indexing. One or the other approach may make more sense or use less code in some cases.

## *Exercises*

1. Vectorized operations make R code efficient to write and execute. To get an idea of the effect of this on execution time, calculate the square root of all the integers from one to 10000 two different ways: as a simple vectorized operation, and within a loop. Time the two methods using `system.time`.

2. Read in the data in the file Eagles.txt. Using a loop, create a separate file for each invidual site that contains the output for the summary function applied to data for the specific site only.

3. Generate a 100 element vector contains random numbers—specify whatever mean you like (or use the default). Now use `ifelse` or `if...else` (whichever is appropriate) to generate a new vector that contains H where the random number is greater than your specifed mean, and L where it is lower than your specified mean.

# 12. Graphics II

*Murrell 2005, Crawley 2007, Dalgaard 2008*

## *12.1. Arranging multiple plots per page*

There are two common ways of putting multiple plots on one page. 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[46]. Before we start describing each of these methods, it is useful to view the current graphic parameters. Query the settings with `par()`.

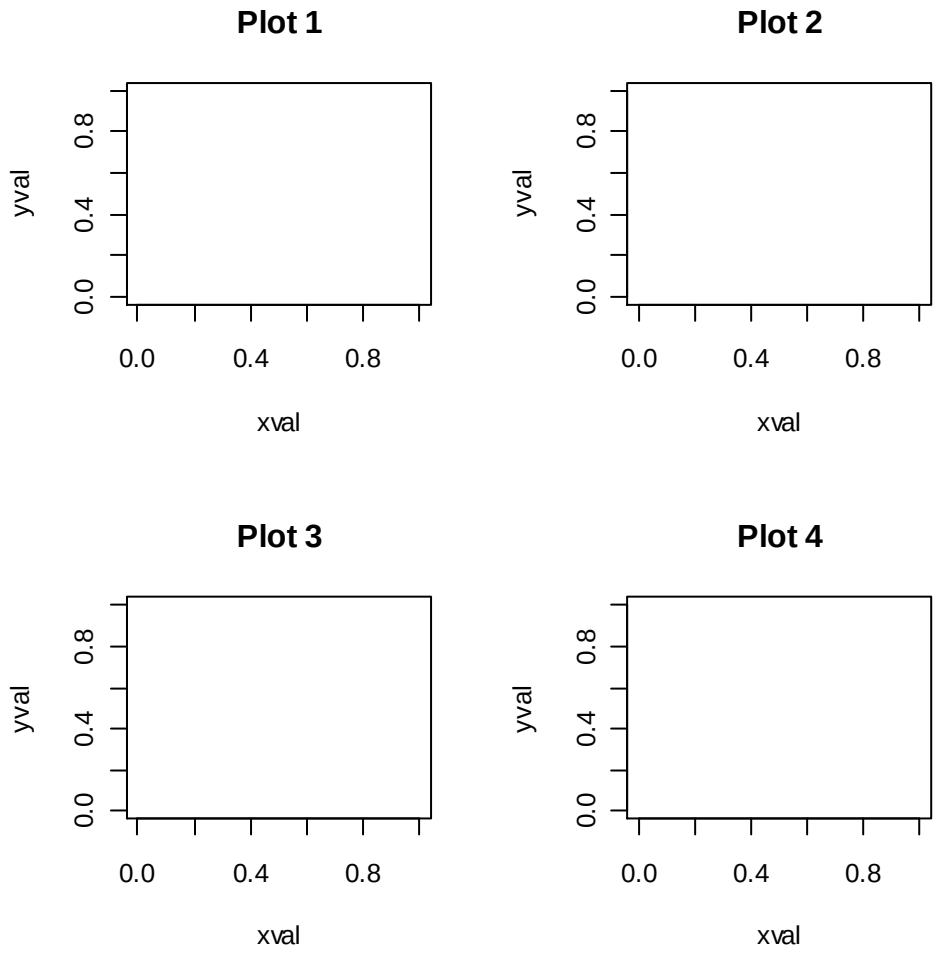
```
> par()
$xlog
[1] FALSE

$ylog
[1] FALSE

...

$mfcol
[1] 1 1

$mfg
[1] 1 1 1 1

$mfrow
[1] 1 1

...

$yaxt
[1] "s"
```

Since we have not yet modified these settings, these are the defaults (some of them—there are a total or 70). Note that the default setting for `mfcol` 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 `mfcol`) would be set to `c(2,2)`. The difference between `mfrow` and `mfcol` is that with `mfrow`, the plots are first organized by row, and with `mfcol`, the plots are first organized by column.

```
> x<-c(0,1)
> y<-c(0,1)
> par(mfrow=c(2,2))
```

[46] You can find more flexibility still in the `lattice` package, which is not covered in this workshop.

```
> 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")
```




Or, with a loop:

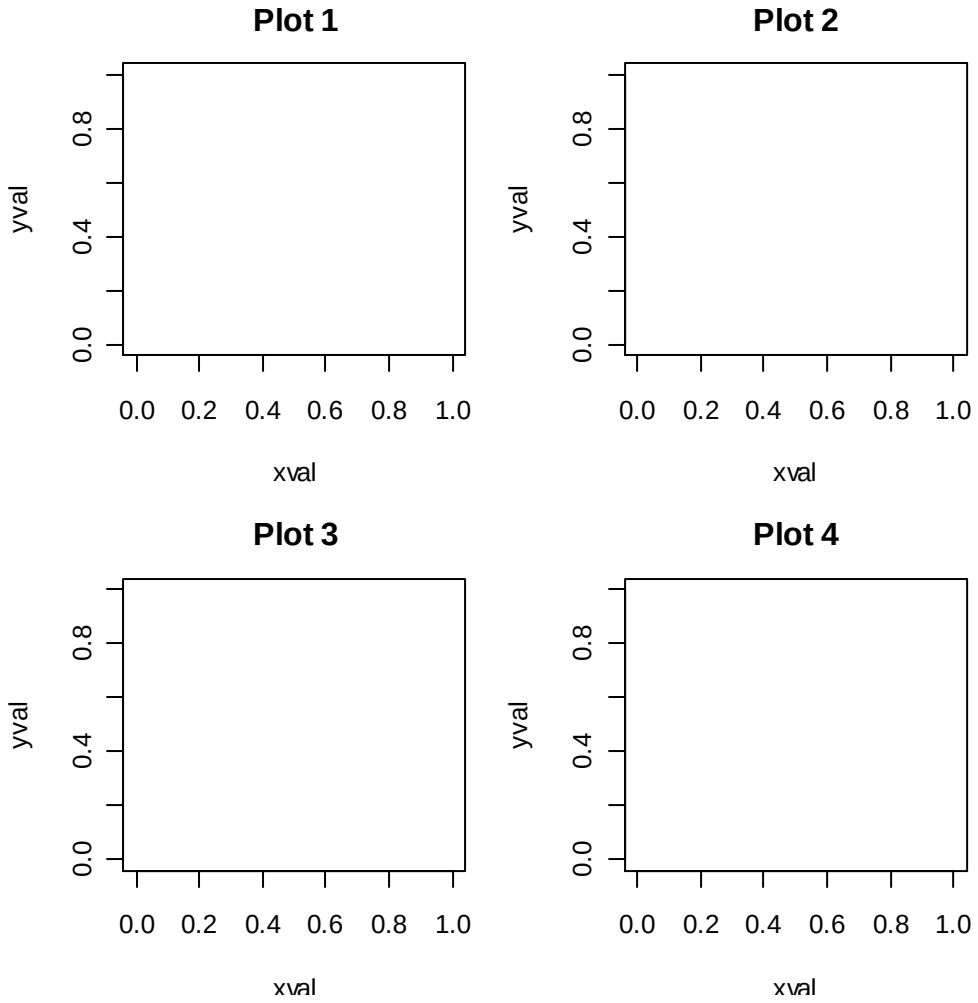
```
> for (i in 1:4) plot(x,y,type="n",main=paste("Plot",i))
```

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. The amount of space around individual plots and aroud 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[47].

```
> par(mfrow=c(2,2),mar=c(4,4,3,1))
> for (i in 1:4) plot(x,y,type="n",main=paste("Plot",i))
```

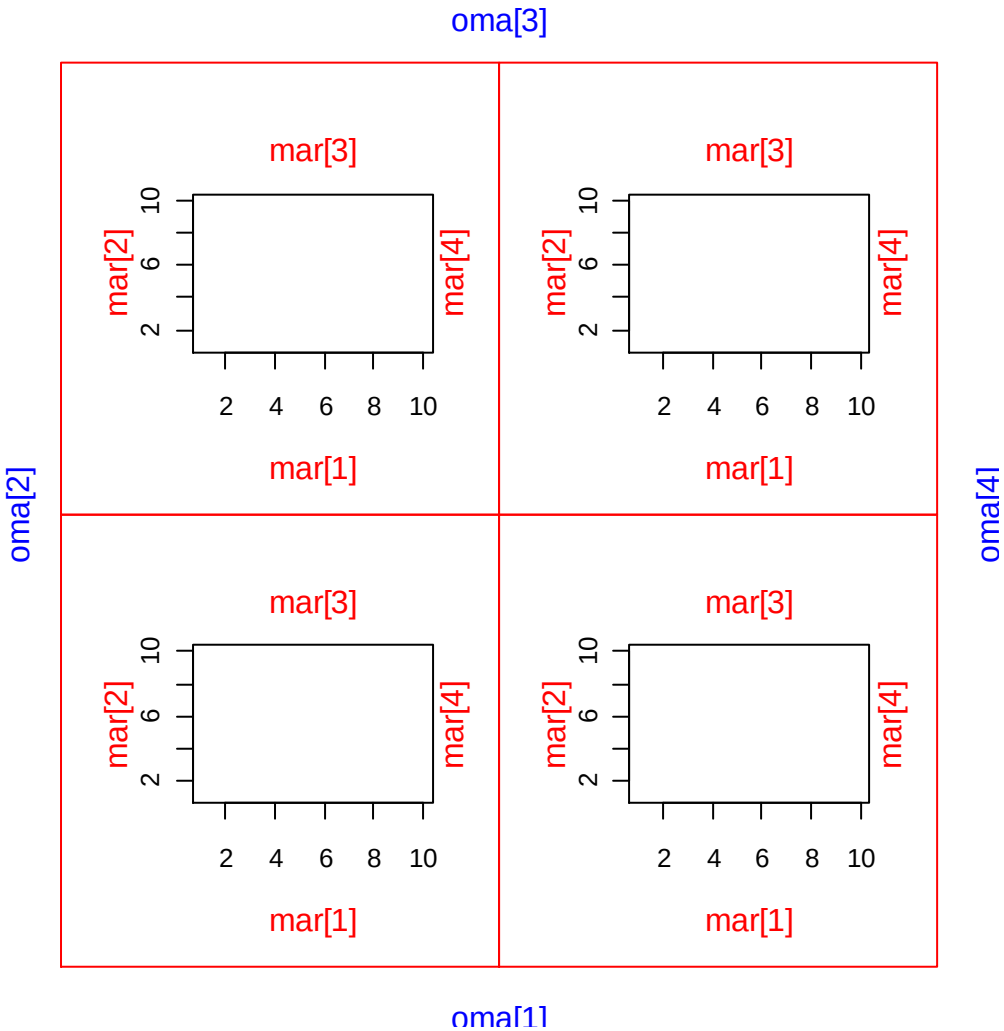



The following code displays the areas controlled by `mar` and `oma`.

```
> 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="")
```

[47] We have also reduced the code for producing four plots to a single line using a loop.

```
+ box("figure",col="red")
+ for(i in 1:4) mtext(paste("mar[",i,"]",sep=""),side=i,line=4-
+ i,col="red")
+ }
> for(i in 1:4) mtext(paste("oma[",i,"]",sep=""),side=i,line=1,
+ col="blue",outer=TRUE)
```




More flexibility is allowed with the `layout` function because plot regions with different sizes can be specified. First, let's reset the `mfrow`, `mar`, and `oma` to the defaults:

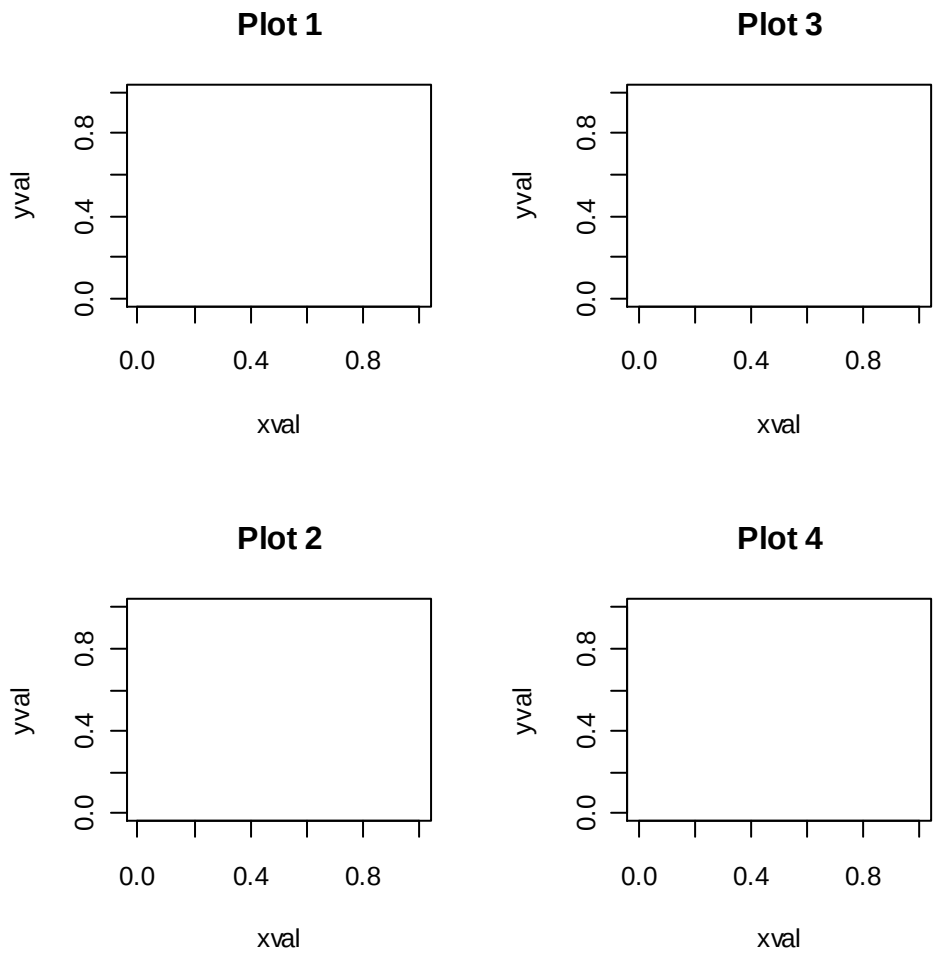
```
> par(mfrow=c(1,1),mar=c(5,4,4,2)+0.1,oma=c(0,0,0,0))
```

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
```

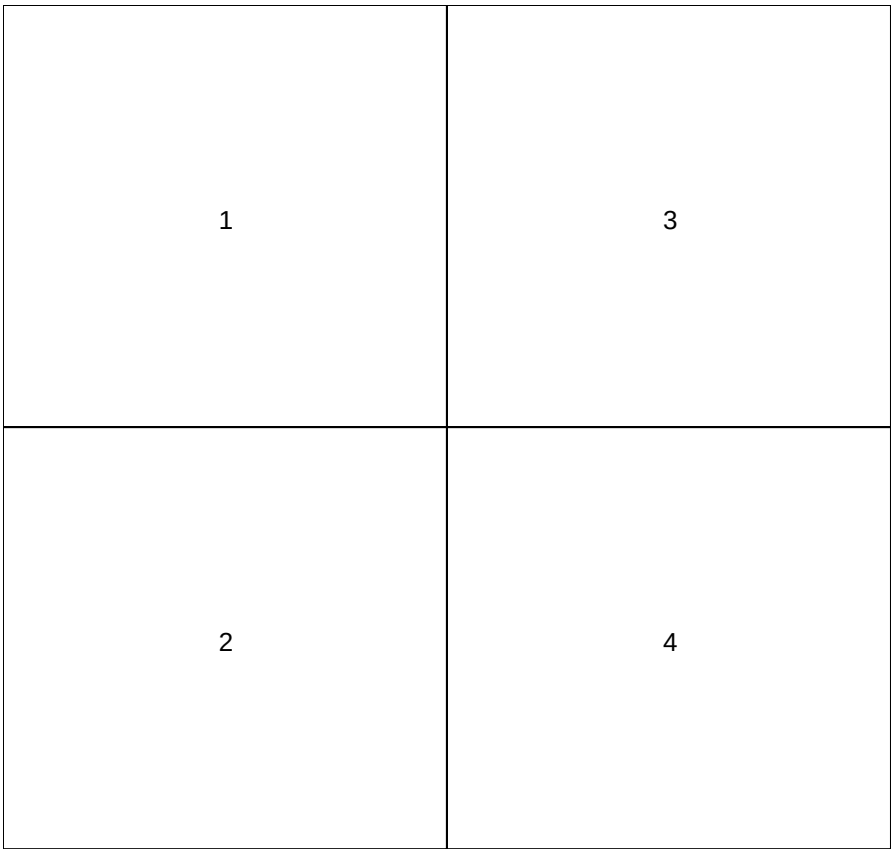
You can probably guess what kind of a layout this will give when you submit it to layout:

```
> layout(matrix(1:4,nrow=2))
> for (i in 1:4) plot(x,y,type="n",main=paste("Plot",i))
```

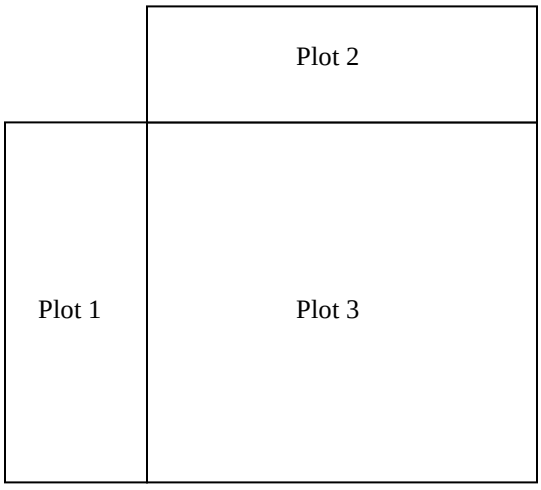



An easier way to quickly show where your plots will show up is to use the `layout.show` function:

```
> layout.show(4)
```




This layout also could have been created by setting `mfcol=c(2,2)` in a `par` call. But, layout can do much more than this. For example, how about something like this?



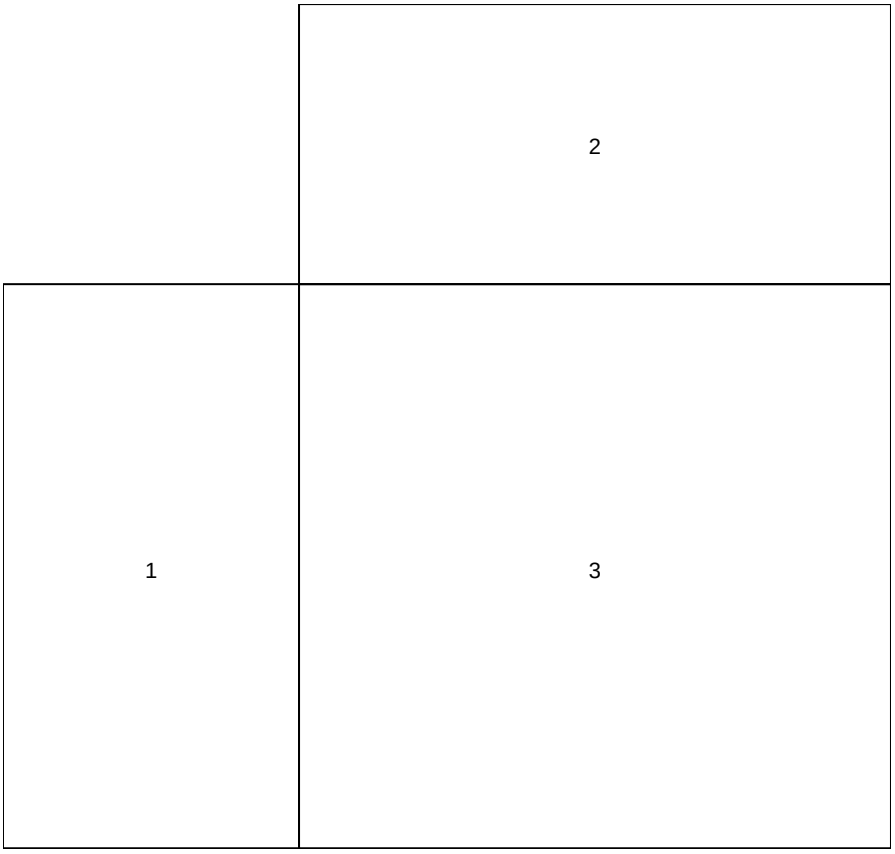

The only challenge in using `layout` is in figuring out a matrix that accurately reflects what you want. In this case, the narrow dimensions of plots 1 & 2 are about half as large as a side of plot 3, so a 3 × 3 matrix should work. How about this:

```
> 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
```

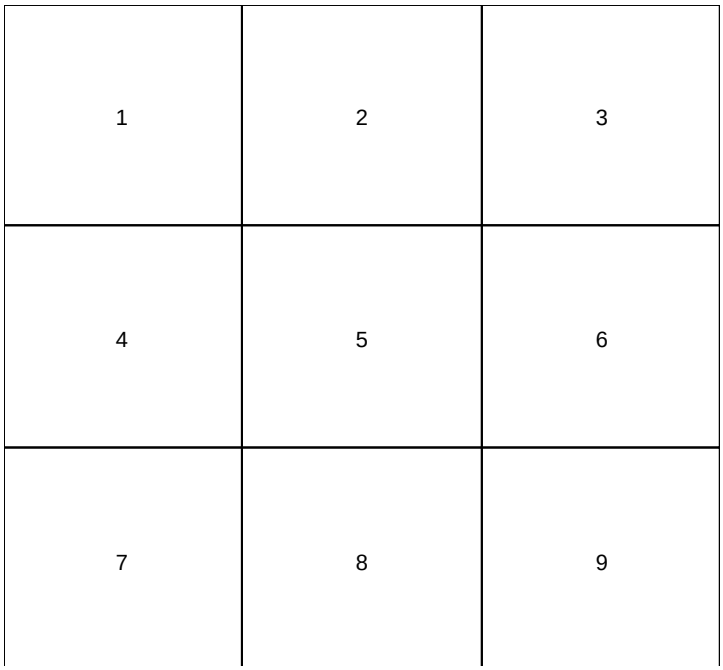
Note that you can just put a 0 where you don't want any plot.

```
> layout.show(3)
```



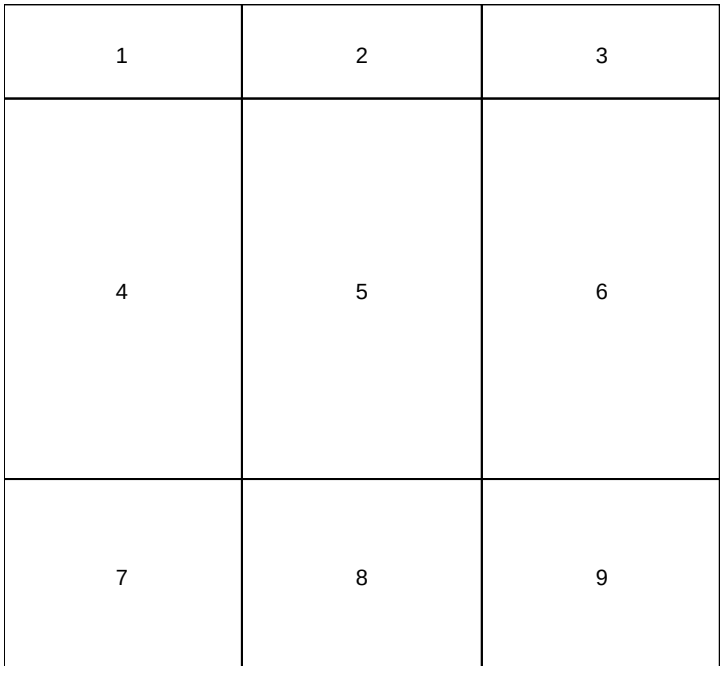

Additional flexibility comes from the `heights` argument in `layout`, which allows you to set the relative height of each row in your matrix.

```
> layout(matrix(1:9,nrow=3,byrow=TRUE))
> layout.show(9)
```




Compare that to this:

```
> layout(matrix(1:9,nrow=3,byrow=TRUE),heights=c(0.5,2,1))
> layout.show(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` is an easier option.

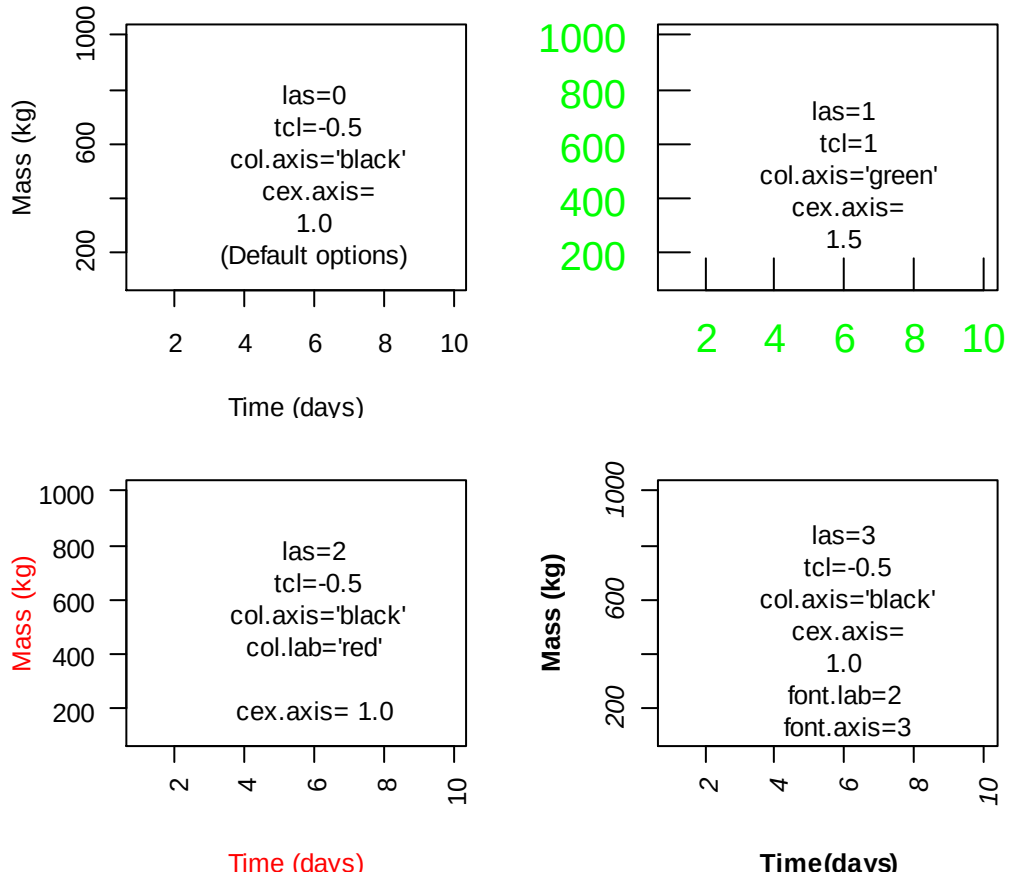
## *12.2. More on the plot function: arguments and values*

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 | Additional information |
|---|---|---|
| `cex` | `0` to `3` (or greater) | Numerical value. Controls size of plotting text and symbols. No upper limit, but typically 0.5-2. Default is 1. Is reset to 1 when the layout is changed. |
| `cex.axis` | `0` to `3` (or greater) | Magnification for axis annotation, relative to cex. |
| `col.axis` | Any color | Color for axis annotation. Default of "black". |
| `col.lab` | Any color | Color for axis label. Default of "black". |
| `font.lab` | `1` through `5` | Font of axis label. 1, plain; 2, bold; 3, italic; 4, bold italic; 5 symbol. See also `family`. |
| `font.axis` | `1`<br>`2`<br>`3`<br>`4`<br>`5` | Font of axis annotation. 1, plain; 2, bold; 3, italic; 4, bold italic; 5 symbol. See also `family`. |
| `las` | `0`<br>`1`<br>`2`<br>`3` | Rotation of axis annotation. 0, parallel to axis; 1, horizontal; 2, perpendicular to axis; 3 vertical. |
| `mgp` | `c(0,0,0)` through `c(5,5,5)` (or greater) | Location of axis title, labels, and line. Default is c(3,1,0). Adjust mgp to move title and numeric labels closer to or farther from plot edge. |
| `tcl` | `0` through `2` (or greater) | Tick mark length. |
| `xaxs` | `"r"`<br>`"i"` | Style of x axis interval. Default is `"r"`, which extends a bit beyond limits. To stick with specified limits, use `"i"`. |
| `yaxs` | `"r"`<br>`"i"` | Style of y axis interval. See xaxs. |

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(4,4,2,2))
> 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 corresponding to the specified settings. If settings are set in a `plot` command, they are only set for that individual plot.

## *12.2 Adding data to plots*

What is the best way to plot multiple data series? The answer probably depends on the details of your data and plot, as well as your preferred approach to writing R code (e.g., as short and efficient as possible versus easy-to-understand). You can plot multiple data series with one `plot` call by using its vectorized capabilities, but this isn't the most straightforward option, and doesn't work well with lines. Functions that add data to an existing plot include `points`, `lines`, and `curve`. For this set of examples, let's revisit some data we've looked at previously on fruitfly longevity.

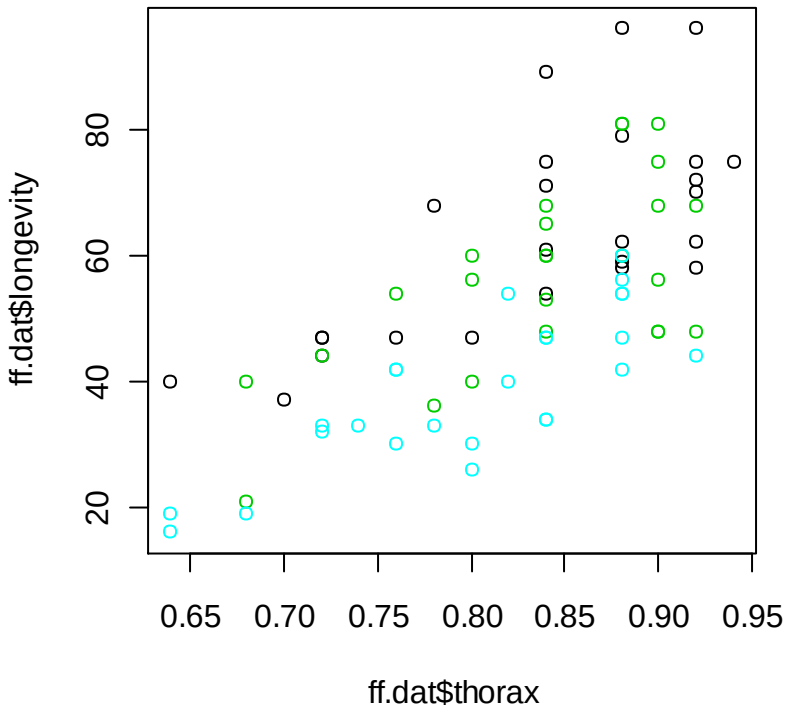
```
> library(faraway)
> summary(fruitfly)

   thorax         longevity         activity
 Min.   :0.6400   Min.   :16.00   isolated:25
 1st Qu.:0.7600   1st Qu.:46.00   one     :25
 Median :0.8400   Median :58.00   low     :25
 Mean   :0.8224   Mean   :57.62   many    :24
 3rd Qu.:0.8800   3rd Qu.:70.00   high    :25
 Max.   :0.9400   Max.   :97.00
```

Let's select just a few activity levels, to keep things simple.

```
> ff.dat<-subset(fruitfly,activity %in% c("isolated","low","high"))
```

To start, let's produce one plot with different series for each activity condition. This example demonstrates the vectorized usage of the `col` argument. In this case, different colors are automatically used for observations with different values for `activity`:

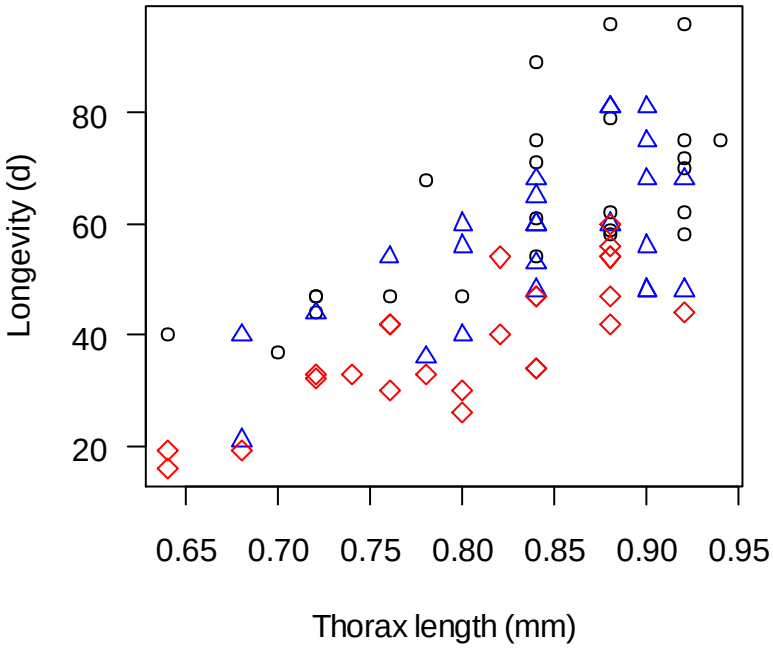
```
> plot(ff.dat$thorax,ff.dat$longevity,col=ff.dat$activity)
```




This simple approach can also be applied with character or factor data—just use `as.numeric` as needed[48]. Moreover, you can use the same type of expression to produce a different plotting symbol for each group. The downside with this approach is that you are stuck with whichever colors or symbols happen to correspond with the numeric data that are in your data set. A more sophisticated and flexible approach is to use indexing or `ifelse` to use a vector of colors or plotting symbols that are based on the value of `activity`. This is most clear if done in a few separate steps

```
> cols<-ifelse(ff.dat$activity=="isolated","black",
+ ifelse(ff.dat$activity=="low",'blue','red'))

> symbs<- ifelse(ff.dat$activity=="isolated",1,
+ ifelse(ff.dat$activity=="low",2,5))

> plot(ff.dat$thorax,ff.dat$longevity,
+ xlab="Thorax length (mm)",ylab="Longevity (d)",
+ pch=symbs,col=cols,las=1)
```

[48] When applied to character data, `as.numeric` will sort the unique values alphabetically, and assign integer values.

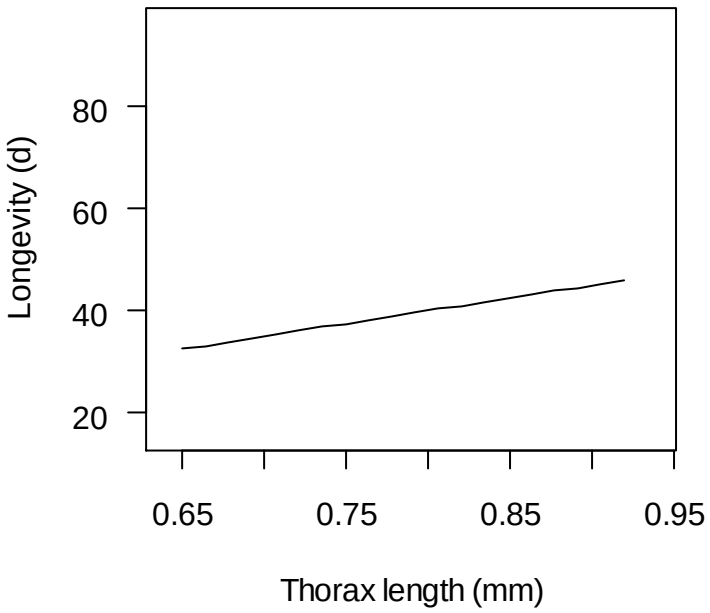



The downside with this approach is that it is a bit complicated, and it doesn't work well with lines. It may be easier to just add points or lines to a plot that already exists. The `points` function can be used to add points or lines to an existing plot. The code below will produce a plot that is identical to the one just above.

```
> plot(ff.dat$thorax,ff.dat$longevity,type="n",
+ xlab="Thorax length (mm)",ylab="Longevity (d)",
+ pch=1,col=ff.dat$activity,las=1)

> points(ff.dat$thorax[ff.dat$activity=="isolated"],
+ ff.dat$longevity[ff.dat$activity=="isolated"],pch=1)

> points(ff.dat$thorax[ff.dat$activity=="low"],
+ ff.dat$longevity[ff.dat$activity=="low"],pch=2,col="blue")

> points(ff.dat$thorax[ff.dat$activity=="high"],
+ ff.dat$longevity[ff.dat$activity=="high"],pch=5,col="red")
```

Lines can also be added to a plot with `points`.

```
> plot(ff.dat$thorax,ff.dat$longevity,type="n",
+ xlab="Thorax length (mm)",ylab="Longevity (d)",las=1)
> x<-seq(0.65,0.92,length=20)
> y<-50*x
> points(x,y,type="l")
```

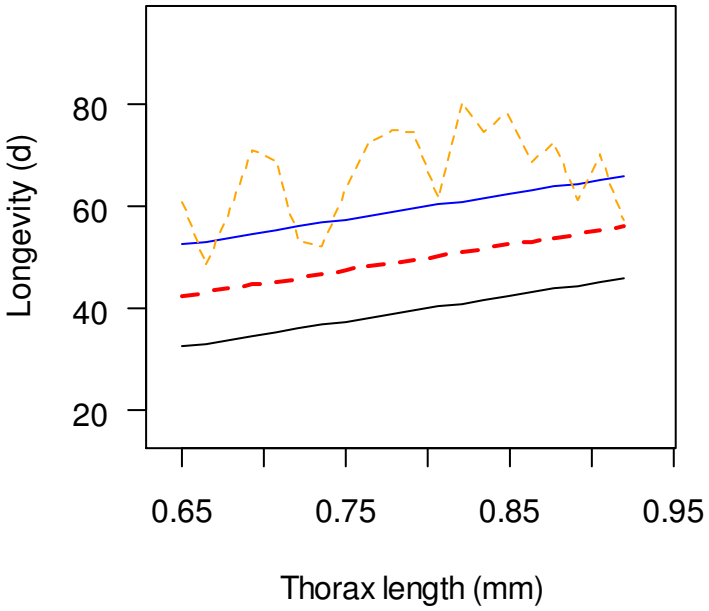



But, the `lines` function requires a little less code.

```
> lines(x,y)
```

Line color, weight, and pattern can be specified in any function that plots lines. Line type, controlled by `lty`, is very flexible and therefore a bit complicated. The easiest approach is to use just use the integers 0 to 6. For (many) more options, see the `lty` entry in the help file for `par`.

```
> lines(x,y+10,lwd=2,col="red",lty=2)
> lines(x,y+20,lwd=1,col="blue",lty="solid")
> lines(x,y+rnorm(20,30,10),lwd=1.5,col="orange",lty=3242)
```



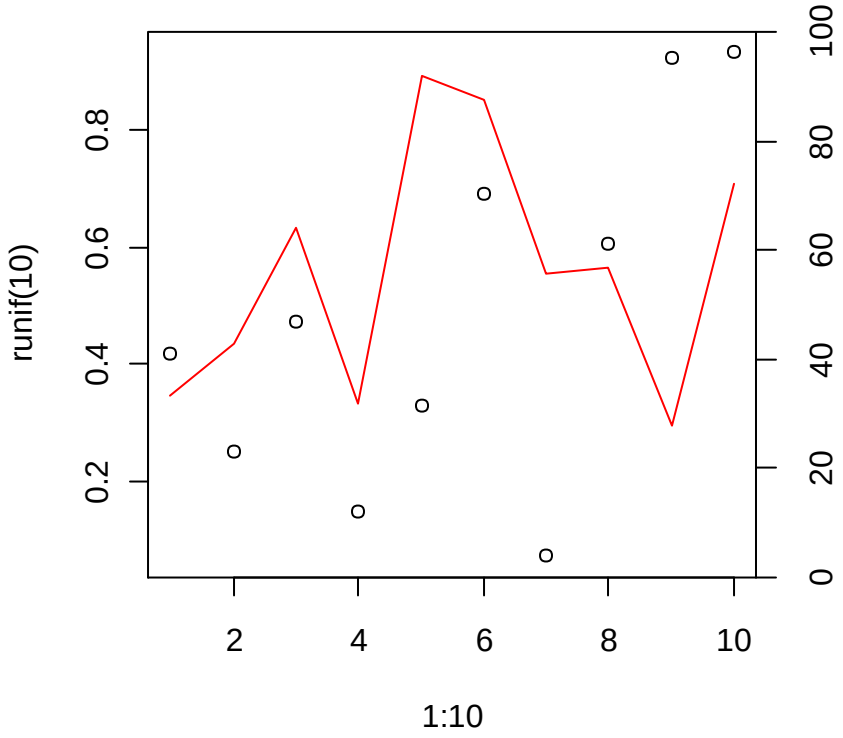

If you can arrange your data as a matrix (this requires that all series have identical x values), three functions for plotting multiple series simultaneously are available. We could add all these lines at once using the `matlines` function. The single command below will do the same thing that three separate commands did above.

```
> matlines(x,matrix(c(y,y+20,y+rnorm(20,30,10)),nrow=20),
+ lwd=c(2,1,1.5),col=c("red","blue","orange"),lty=c(2,1,1))
```

For adding multiple series as points, an analogous function `matpoints` is available. And, to generate a plot and add multiple series of points or lines at once, use `matplot`.

What if you want to plot multiple data series on a single plot, but with different axes? There are at least two ways to do this, and although both are easy, they may not be easy to remember. In the first method, use `par` to set the limits of the y axis.

```
> plot(1:10,runif(10))
> par(usr=c(par('usr')[1:2],0,100))
> axis(4)
> lines(1:10,rnorm(10,50,25),col="red")
```

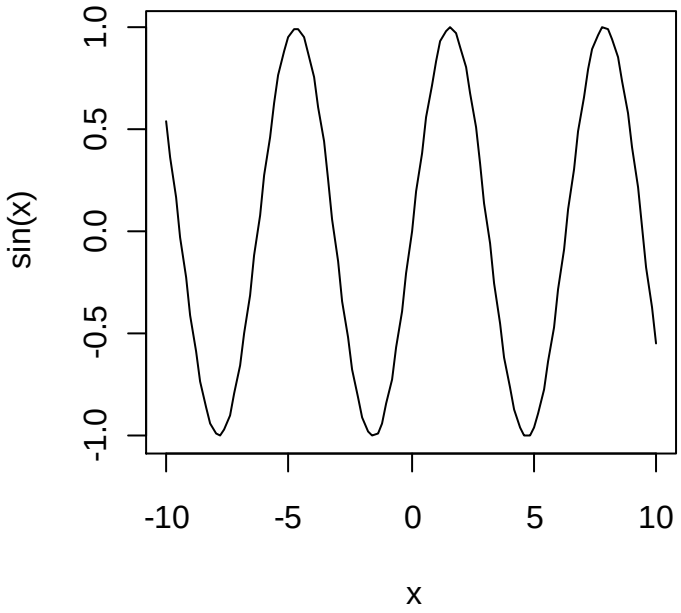



The alternative is to plot a new plot over a previous one, setting `axes` and other arguments to `FALSE` so they don't show up. Then, set `par(new=TRUE)` (this tells R that the plotting device is on a new page already, and that it is therefore OK to go ahead and plot), and finally, submit the command to produce a new plot. To generate the same example shown above, you could use the following code. This approach is very flexible, because there is no limit to what type of plots can be overlaid. But, you need to be sure that your plots align as you think they do.

```
> plot(1:10,rnorm(10,50,25),ylim=c(0,100),axes=FALSE,xlab="",
+ ylab="",col="red",type="l")
> axis(4)
```

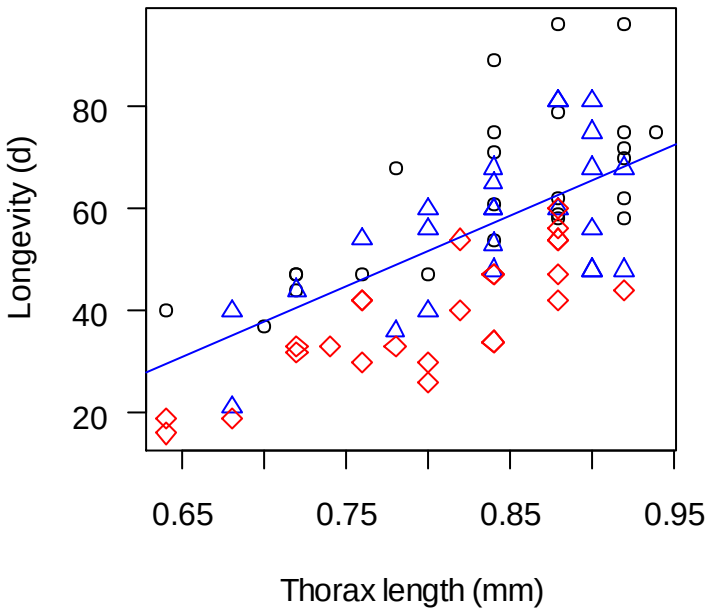
The `curve` function is used for plotting functions. Given alone, it will create a plot, but if submitted after a plot call has already been made with the argument `add=TRUE`, it will add a new line to the plot. This function is the easiest way to see visualize a function.

```
> curve(sin(x),-10,10)
```



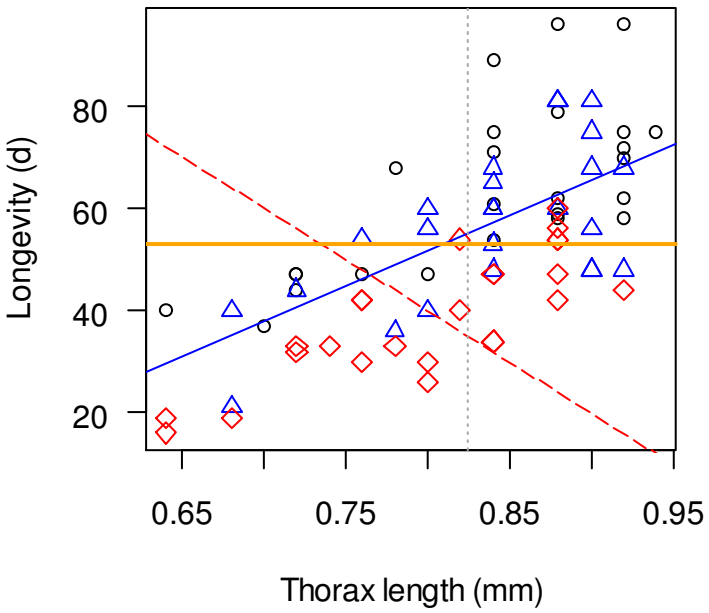

If you want to add model predictions to a plot, we can of course use `lines` or `points` after we have generated predictions. If you want to plot model predictions as a function of the only predictor variable, the flexible `abline` function is easier. Let's demonstrate with the fruit fly data.

```
> plot(ff.dat$thorax,ff.dat$longevity,
+ xlab="Thorax length (mm)",ylab="Longevity (d)",
+ pch=symbs,col=cols,las=1)

> mod<-lm(longevity ~ thorax,data=ff.dat,subset=activity=="low")
> abline(mod,col="blue")
```




The `abline` function can only be used for straight lines (unlike `points`, `lines`, and `curve`). With it, you can specify a slope and intercept manually, or plot horizontal or vertical lines.

```
> abline(a=200,b=-200,col="red",lty=5)
> abline(v=mean(ff.dat$thorax),col="darkgray",lty="dotted")
> abline(h=mean(ff.dat$longevity),col="orange",lwd=2.5)
```



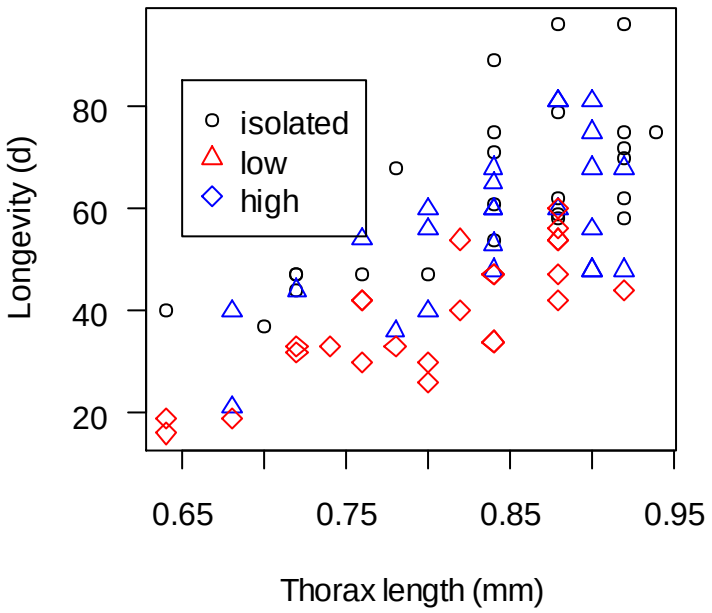

## *12.3. Annotating plots*

Let's start with an important addition to a plot—a legend. The `legend` function in R is perhaps a little less automatic than in other graphics software, but once you get the hang of it, you may appreciate its flexibility. Let's generate a plot using, again, the fruit fly data.

```
> plot(ff.dat$thorax,ff.dat$longevity,
+ xlab="Thorax length (mm)",ylab="Longevity (d)",
+ pch=symbs,col=cols,las=1)
```

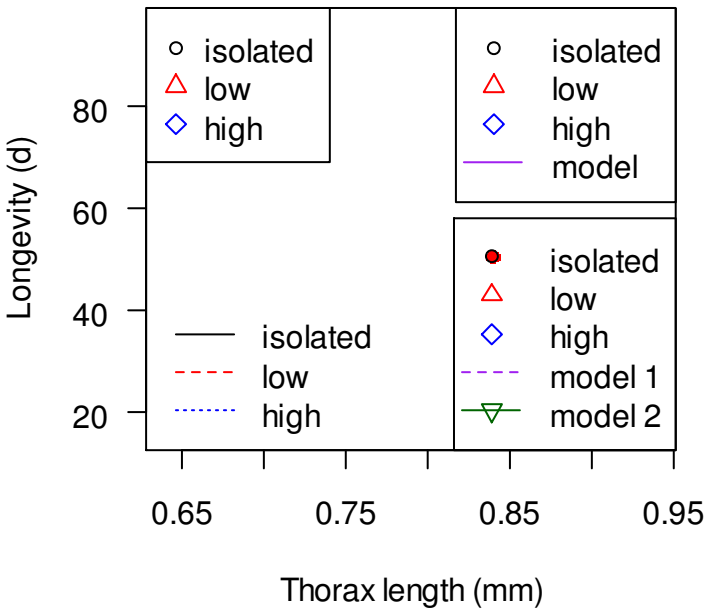
There are many optional arguments for the `legend` function (see the help file for a list), but most of the time, you just need to specify a location, the text labels, and plotting characters or lines. For example, using the plot generated.

```
> legend(0.65,85,c("isolated","low","high"),pch=c(1,2,5),
+ col=c("black","red","blue"))
```




This function has some handy shortcuts for positioning a legend—instead of specifying x and y coordinates, you can just use "top","bottom","left","right","middle", or combinations of these. You can also use lines in place of plotting symbols, or you can combine lines and plotting symbols.

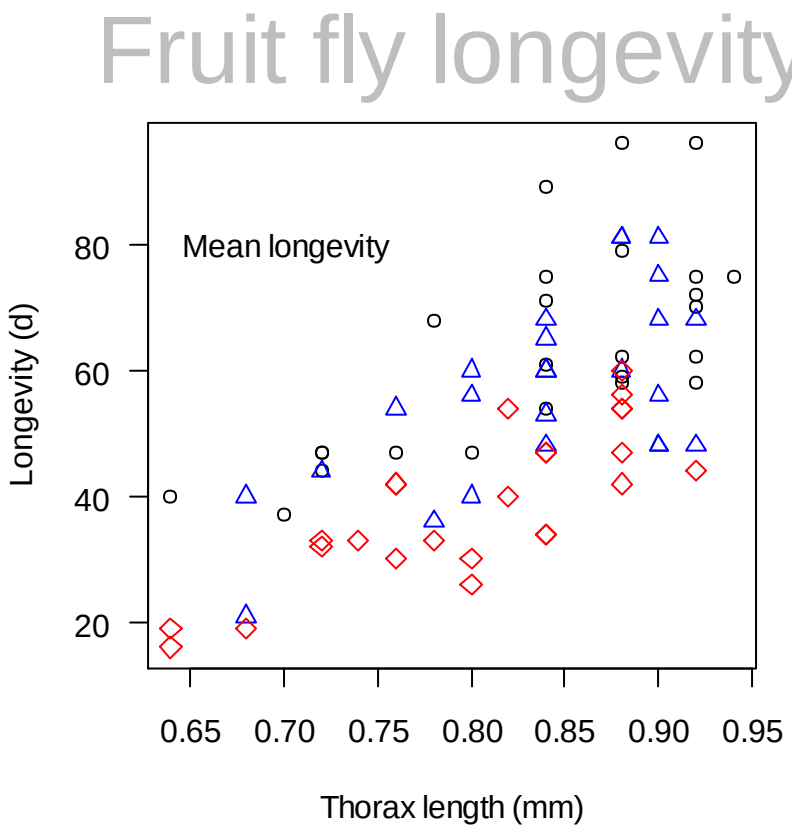
```
> plot(ff.dat$thorax,ff.dat$longevity,type="n",
+ xlab="Thorax length (mm)",ylab="Longevity (d)",las=1)

> legend("topleft",c("isolated","low","high"),pch=c(1,2,5),
+ col=c("black","red","blue"))

> legend("bottomleft",c("isolated","low","high"),lty=1:3,
+ col=c("black","red","blue"),bty="n")

> legend("topright",c("isolated","low","high","model"),
+ pch=c(1,2,5,-1),lty=c(0,0,0,1),col=c("black","red","blue","purple"))

> legend("bottomright",c("isolated","low","high","model 1","model 2"),
+ pch=c(21,2,5,-1,6),lty=c(0,0,0,2,1),pt.bg="red",
+ col=c("black","red","blue","purple","darkgreen"))
```




Notice that it can be a little tricky to get combinations of lines and symbols. The trick is to use `lty=0` for no line and `pch= - 1` for no plotting symbol.

To add text to a plot, you can use the `text` or the `mtext` functions. These two functions are similar, but `mtext` is for adding text in the plot margins.

```
> plot(ff.dat$thorax,ff.dat$longevity,
+ xlab="Thorax length (mm)",ylab="Longevity (d)",
+ pch=symbs,col=cols,las=1)

> text(0.7,80,"Mean longevity")

> mtext("Fruit fly longevity",side=3,line=1,cex=3,col="gray")
```

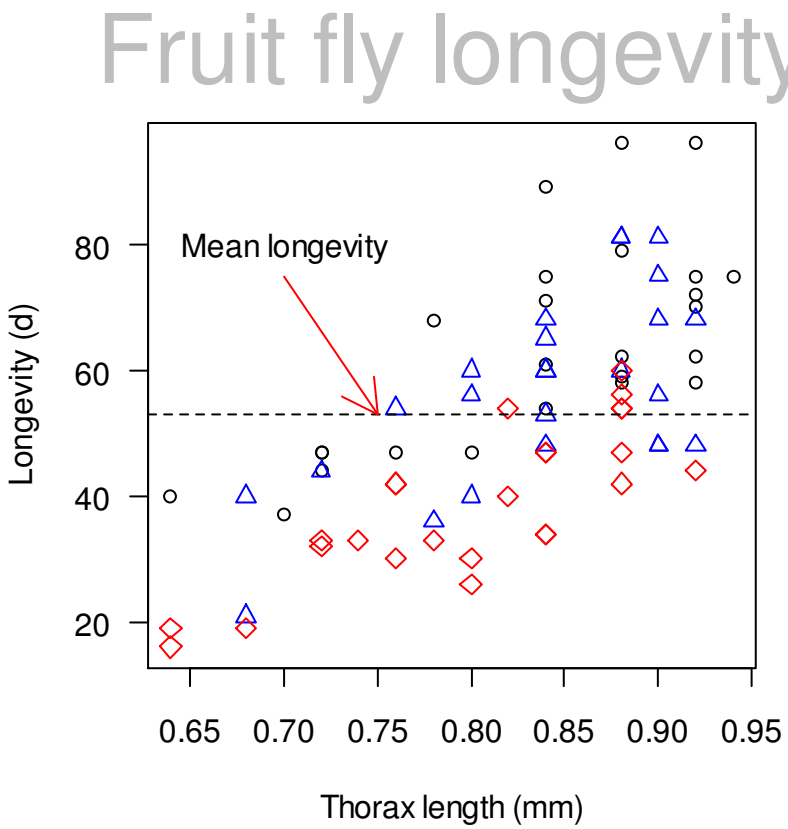



Both `text` and `mtext` have several optional arguments that can be used to adjust font, size, or color (as is demonstrated in the `mtext` command above). See the help file for par for more information.

With the function `arrows`, it is easy to create arrows and specify a color, head length and angle, and line characteristics. Again, see the help file for par for more information. Note that `text` and `arrows` are both vectorized functions. Let's add a line and an arrow to the plot we just made.

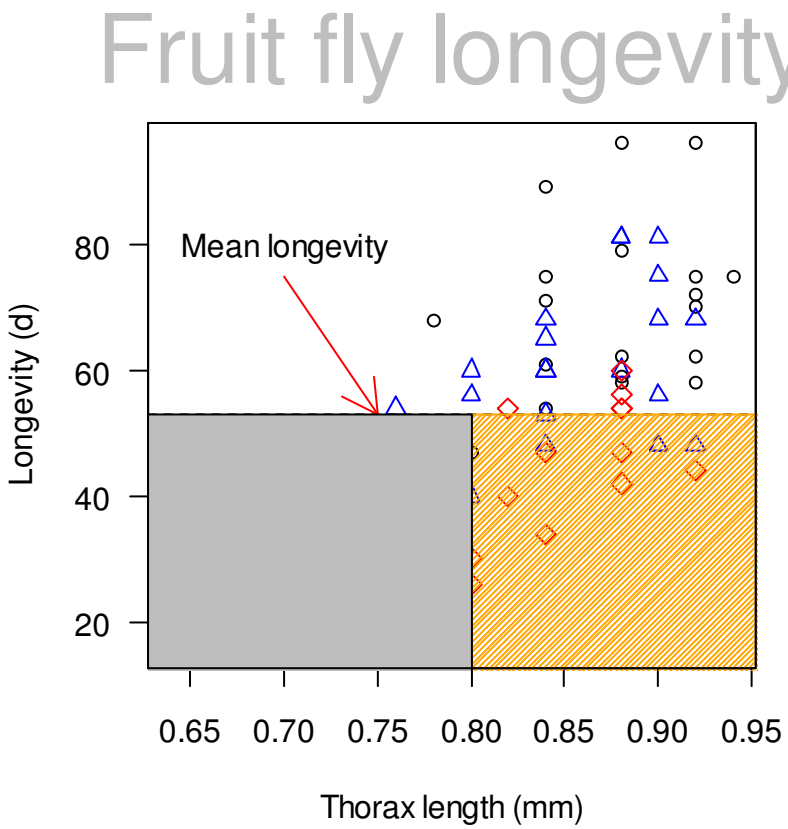
```
> abline(h=mean(ff.dat$longevity),lty=2)
```

```
> arrows(0.7,75,0.75,mean(ff.dat$longevity),length=0.2,col="red")
```




Polygons of any shape can be added to a plot in R.

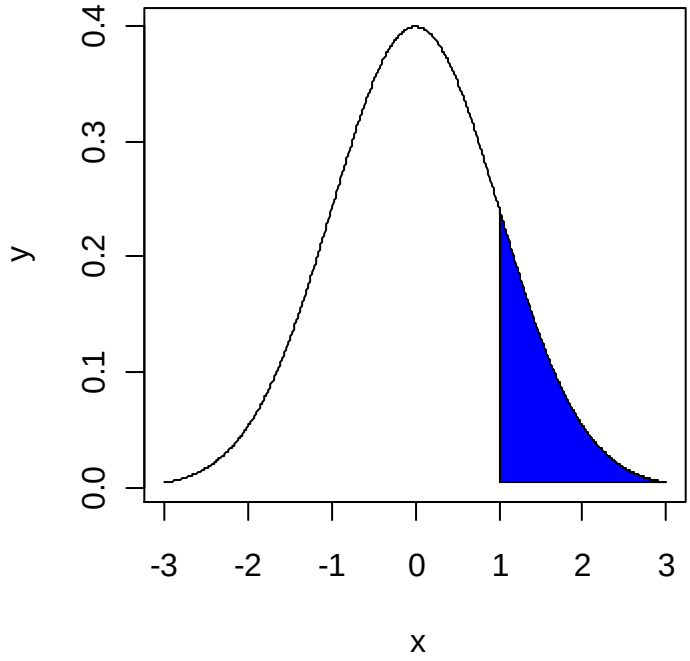
```
> polygon(x=c(0.8,0.8,1,1),y=c(0,mean(ff.dat$longevity),
+ mean(ff.dat$longevity),0),density=50,col="orange")

> polygon(x=c(0.6,0.6,0.8,0.8),y=c(0,mean(ff.dat$longevity),
+ mean(ff.dat$longevity),0),col="gray")
> box()
```




A more interesting example of using `polygon` showing the vectorized nature of the vertices follows:

```
> x<-seq(-3,3,0.01)
> y<-dnorm(x)
> plot(x,y,type="l")
> polygon(c(1,x[x>=1]),c(y[x==3],y[x>=1]),
+ col="blue")
```

We will cover the `dnorm` function later, but as you might guess, it is associated with the normal distribution. This figure shows that you can specify a vector for the vertices, and the resulting polygon appears to have smooth edges.

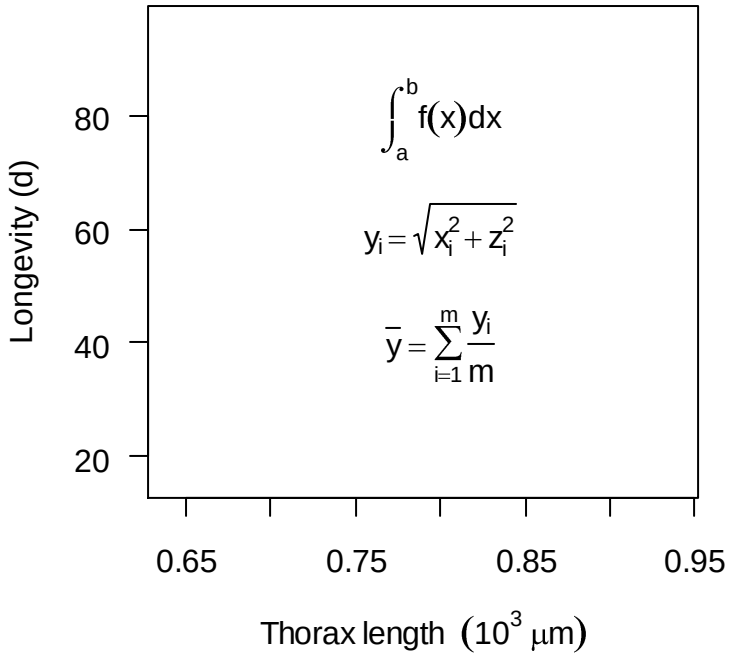



Equations and symbols can be added to figures by using the `expression` function for the value for any argument that accepts text. The syntax for writing equations and symbols with function can be viewed by typing:

```
> demo(plotmath)
```

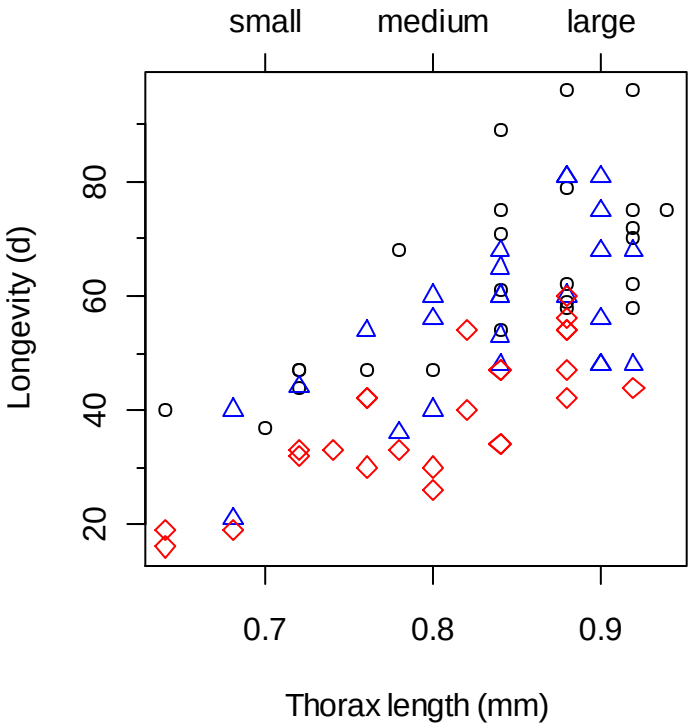
The code below shows just a few uses of `expression`.

```
> plot(ff.dat$thorax,ff.dat$longevity,type="n",
+ xlab=expression("Thorax length"~~(10^3~mu*m)),
+ ylab="Longevity (d)",las=1)

> text(0.8,40,expression(bar(y)==sum(frac(y[i],m),i==1,m)))
> text(0.8,60,expression(y[i]==sqrt(x[i]^2+z[i]^2)))
> text(0.8,80,expression(integral(f(x)*dx,a,b)))
```




If you are not happy with the default axes that R produces, you can produce custom ones using the `axis` function. The only required argument is `side` (1=bottom, 2=left, 3= top, 4=right). Some examples are shown below.

```
> plot(ff.dat$thorax,ff.dat$longevity,
+ xlab="Thorax length (mm)",ylab="Longevity (d)",
+ pch=symbs,col=cols,las=1,axes=F)

> axis(1,at=7:9/10)
> axis(2,at=2:9*10,labels=F,tcl=-0.2)
> axis(2,at=c(20,40,60,80))
> axis(3,at=7:9/10,labels=c("small","medium","large"))
> box()
```

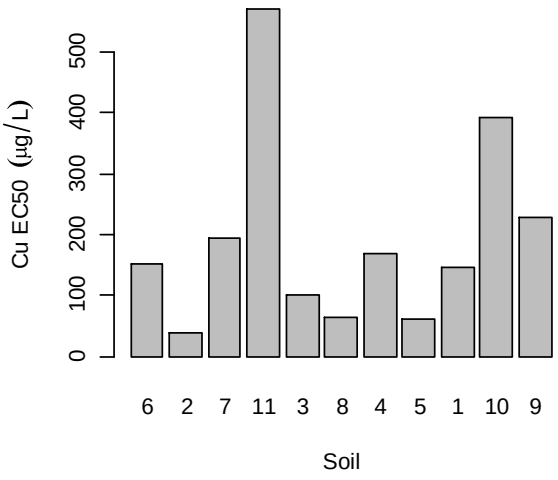


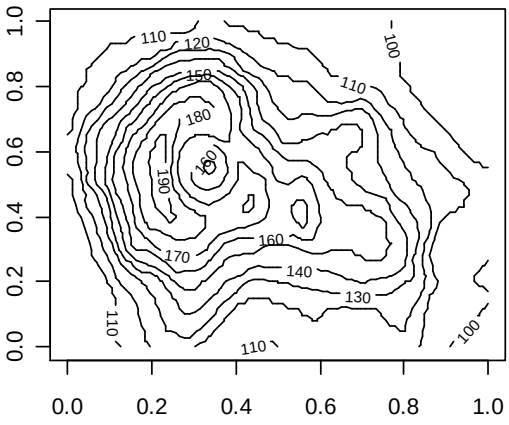

All these possibilities for adding "stuff" to plots may seem a bit overwhelming. We do not recommend that you try to memorize all of them. Instead, use the functions we covered as you need them. As if this wasn't complicated enough, each function accepts numerous optional arguments that are not listed in the corresponding help file. These arguments adjust graphical parameters, and you can find information on them in the help file for `par`. If you think that some option should exist (What if I want to rotate text in `text`? Can I change the line thickness of an arrow? Shouldn't I be able to align a `text` label based on its center?) it probably does—take a look at the `par` help file.

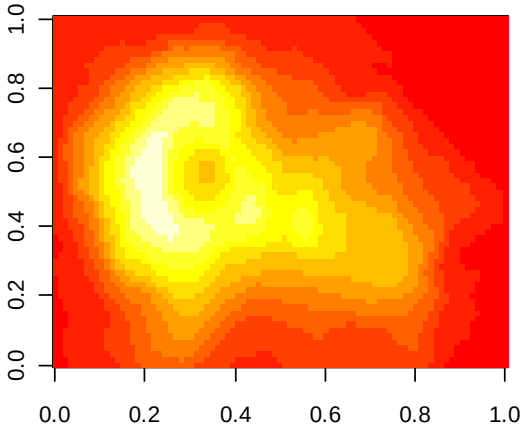
## *12.4. Other high-level plotting functions*

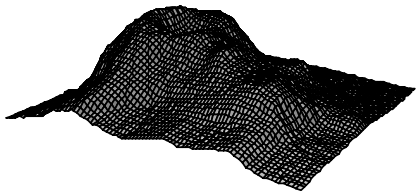
Many other high level plotting functions are available in R. We are not going to cover many, and won't cover any in detail, but we hope the code given below gives you some idea of possibilities. More information can be found online, and in books on R graphics (e.g., Murrell 2005).

```
> par(mfrow=c(2,2))
> cu.tox.dat<-read.table("Thakali_Cu_EC50s.txt",header=TRUE)
> barplot(cu.tox.dat$ec50.cu, names.arg=as.numeric(cu.tox.dat$soil),
+ ylab=expression("Cu EC50 "~(mu*g/L)),xlab="Soil")
> contour(volcano)
> image(volcano)
> persp(volcano,theta=30,phi=15,d=1.5,expand=0.3,box=FALSE,shade=0.3)
```



And, for plotting factor data, check out two options.

```
> mosaicplot(Titanic, main="Survival on the Titanic", color = TRUE)
> squirrel.dat<-read.table("Squirrel_color.txt",header=T)
> squirrel.dat$black<-factor(squirrel.dat$black)
> spineplot(black~dist2ctr,data=squirrel.dat,breaks=100,
+ col=c('gray70','gray25'))
```

## *12.5. Graphics output*

R can be used to produce graphics output in a variety of graphical formats. Output is directed to a particular output "device" that dictates the output format that will be produced. The device must be opened in order to receive graphical output, and then it must be closed to complete the output.

For example, to create a pdf file that contains some of the plots we produced above, we start by opening the device.

```
> pdf("Plots.pdf",height=11,width=8.5)
```

Now we can go ahead and add our plots to the pdf file. In this case, we first change some of the plotting parameters.

```
> par(mfrow=c(2,2),oma=c(1,1,1,1))
> # First plot
> plot(ff.dat$thorax,ff.dat$longevity,
+ xlab="Thorax length (mm)",ylab="Longevity (d)",
+ pch=symbs,col=cols,las=1)

> mod<-lm(longevity ~ thorax,data=ff.dat,subset=activity=="low")
> abline(mod,col="blue")

> # Second plot
> plot(ff.dat$thorax,ff.dat$longevity,
+ xlab="Thorax length (mm)",ylab="Longevity (d)",
+ pch=symbs,col=cols,las=1)

> text(0.7,80,"Mean longevity")

> mtext("Fruit fly longevity",side=3,line=1,cex=3,col="gray")

> abline(h=mean(ff.dat$longevity),lty=2)
> arrows(0.7,75,0.75,mean(ff.dat$longevity),length=0.2,col="red")

> # Third plot
> x<-seq(-3,3,0.01)
> y<-dnorm(x)
> plot(x,y,type="l")
> polygon(c(1,x[x>=1]),c(y[x==3],y[x>=1]),
+ col="blue")

> # Fourth plot
> plot(ff.dat$thorax,ff.dat$longevity,type="n",
+ xlab=expression("Thorax length"~~(10^3~mu*m)),
+ ylab="Longevity (d)",las=1)

> text(0.8,40,expression(bar(y)==sum(frac(y[i],m),i==1,m)))
> text(0.8,60,expression(y[i]==sqrt(x[i]^2+z[i]^2)))
> text(0.8,80,expression(integral(f(x)*dx,a,b)))
```

We have to close the device when we are finished.

```
> dev.off()
```

If you run the above code, you should end up with a new pdf file in your working directory.

In addition to pdf files, R can create bmp, jpeg, png, and tiff files, as well as postscript files. For example:

```
> png(file="Fig.png",width=4,height=4,units="in",res=400)
> filled.contour(volcano)
> dev.off()
```

A png file cannot be more than one page (neither can jpeg, bmp, or tiff), so if multiple pages are desired, they must be in different files. It is very easy to create multiple graphics files in a single command: just insert `%02d` (or `%01d` or `%03d`. . .) in the file name[49].

```
> png(file="Fig%02d.png",width=4,height=4,units="in",res=400)
> image(volcano)
> contour(volcano)
> filled.contour(volcano)
> dev.off()
```

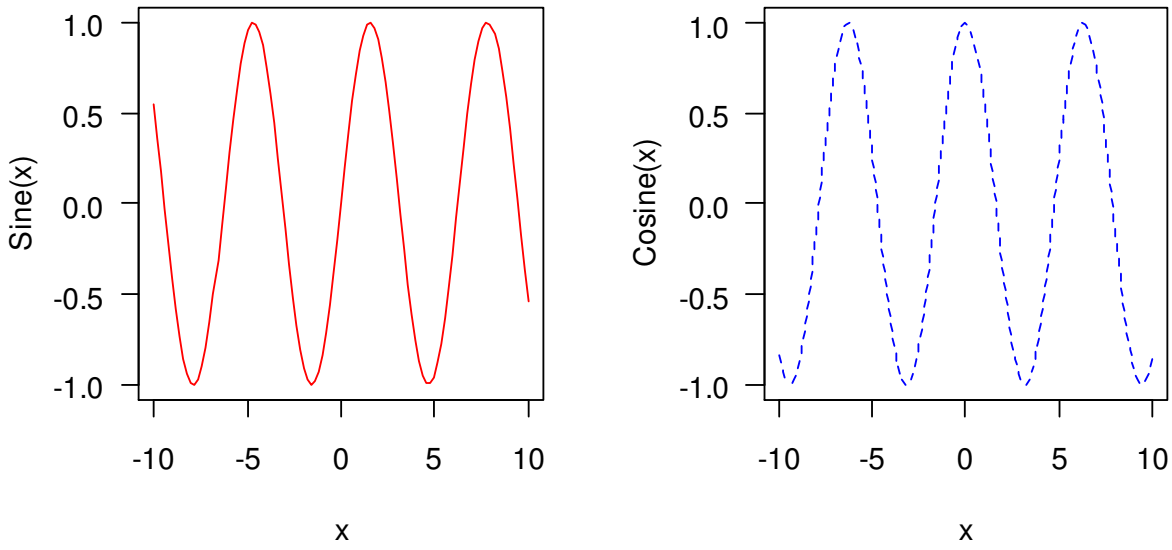
This code produces three files: Fig01.png, Fig02.png, and Fig03.png (but it could produce more). Conversely, pdf and postscript files can contain multiple pages.

Each of the graphics output types has several optional arguments, e.g., for adjusting the size and quality of the resulting files.

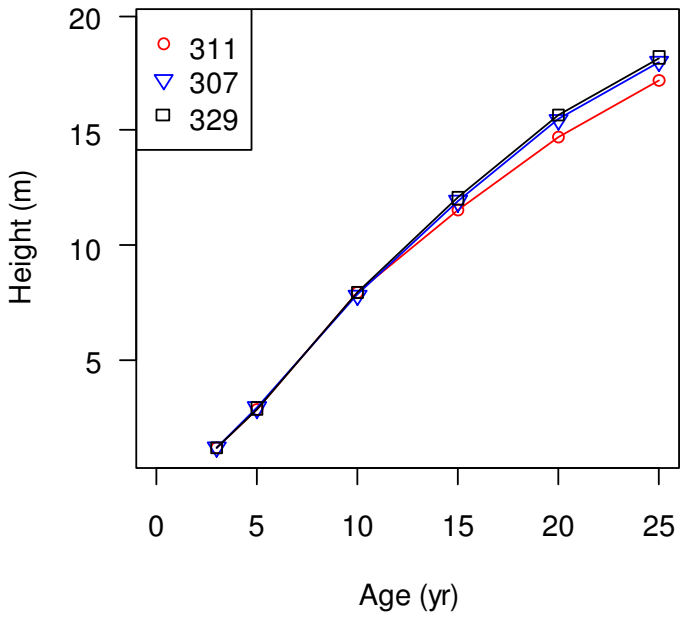
## *Exercises*

1. Produce a two-panel plot like the one shown below. (Hint: you can use the plot fuction to make the plots, but recall that there is an easier way.) Now plot both series on a single plot. See if you can do this using more than one approach.

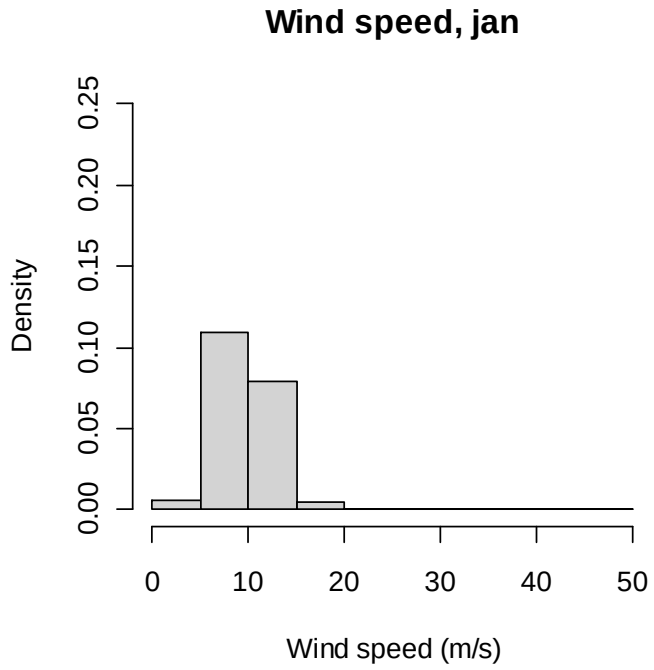
[49] The expression %02d tells R to number the files using a two-digit number. The default file name value actually uses this option.




2. Check out the data in the data frame Loblolly (in the `datasets` package), on the growth of loblolly pine trees from different stocks. Create the following plot, or something similar (note that you will need to convert height from ft to m).

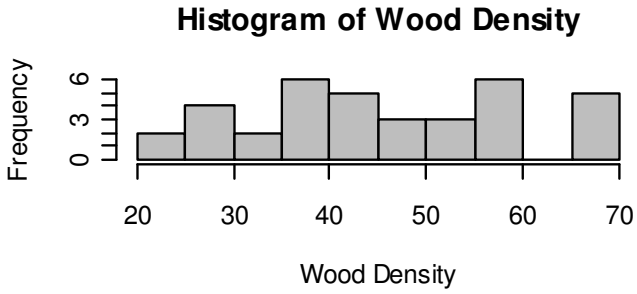


3. Read in the data on wind speed in US cities in the file Ave_wind_US.txt (note that you need to specify the tab separator explicitly, because the column of names contains spaces). Using a loop, produce a histogram like the one below for each month.



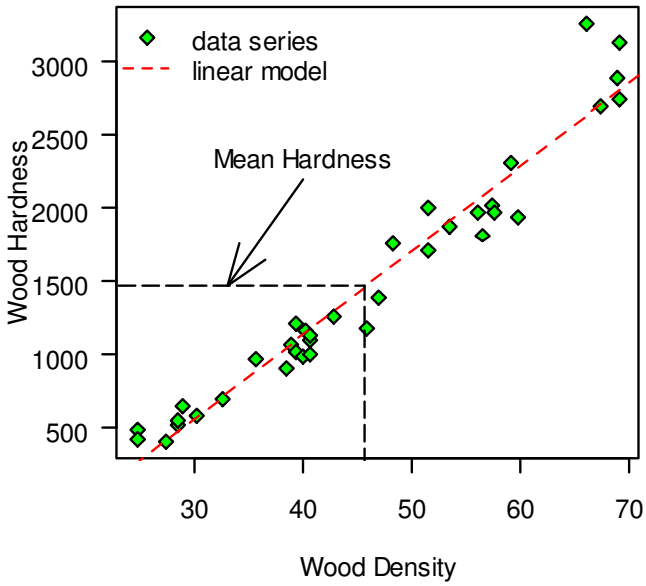

Export each one of the 12 plots produced to a separate jpeg file. Then, try exporting all the plots to a single pdf file, but six per page.

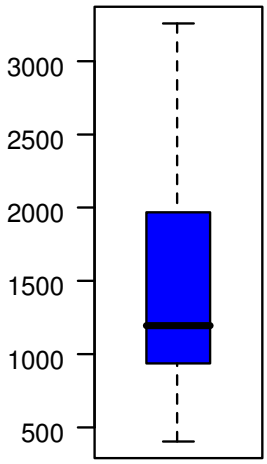
4. Using the Janka.txt dataset that was used earlier for the `lm` example, create the following plots. You will have to use `layout` to get plotting regions of different size. Export this plot to a pdf file.




Mean hardness
= 1470

Mean density
= 45.7



# 13. Functions

*Crawley 2007: 47, R-Intro: Chapter 10, R-Lang: Chapter 4*

## *13.1. Writing functions*

User-definied functions are a very useful feature of R. Once defined, these functions are stored internally, and effectively do not differ from the functions that come with any R packages. Functions can be entered directly into the R GUI or saved in a script file that can be later loaded. It is also possible to store your function calls (or a source call to a script) in a special file Rprofile.site so that they are loaded every time you start R.

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 simple 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 your expression does not need to be assigned to anything—it is automatically returned when you call up the expression. Let's test out this function.

```
> x<-c(100,1000,10000)

> geomean(x)
[1] 1000
```

Let's look at a more complicated example. Let's say we want a function that will calculate the fraction of sample variance explained by a model (while statistical model output in R generally give this or similar information, in some cases it is necessary to calculate this separately, for example when the model calculations are made by a separate piece of software).

```
r2<-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 an example of a simple linear regression from above.

```
> hard.dat<-read.table("Janka.txt",header=T)
> mod.1<-lm(hardness ~ density, data = hard.dat)
> hard.dat$hard.pred<-predict(mod.1)
> hard.dat
```

```
   density hardness hard.pred
1     24.7      484  259.9152
2     24.8      427  265.6658
3     27.3      413  409.4325
...
36    69.1     3140 2813.2115

> r2(meas=hard.dat$hardness,mod=hard.dat$hard.pred)
[1] 0.9493278
```

Note that the variables defined within the function are not available outside the function—they are locally stored (i.e. they are not global) and are lost when R exits the function.

```
> tss
Error: object "tss" not found
```

As with all the functions we covered, you can use both named and positional arguments.

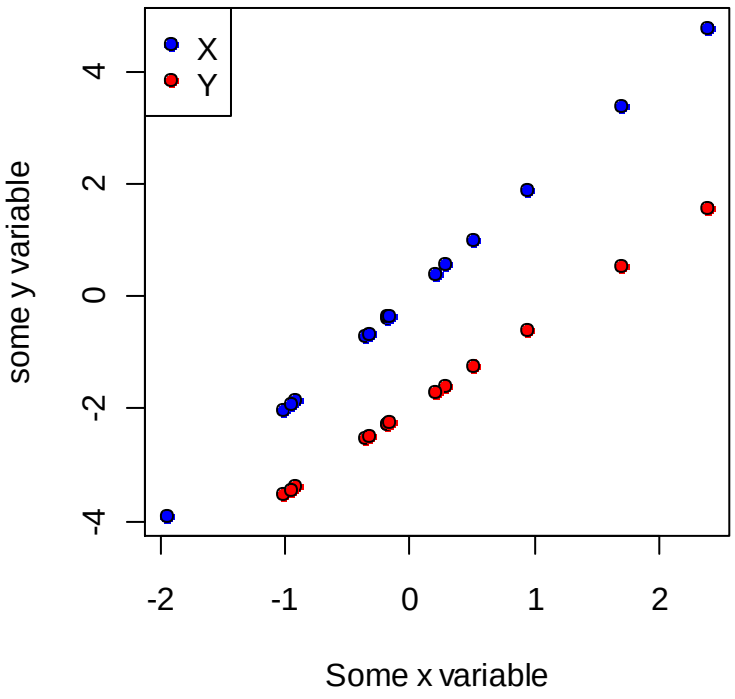
```
> r2(hard.dat$hardness,hard.dat$hard.pred)
[1] 0.9493278
```

If you ever forget the arguments or their order, just use the `args` function (or type the function name without parentheses to see the complete code).

```
> args(r2)
function (meas, mod)
NULL
```

It is possible (and very common) to call up functions from within other functions. For example, let's say you want a specific variation on the `plot` function so that the result always looks a certain way.

```
plot4me<-function(x,y,z) {
   plot(x,y,xlab="Some x variable",ylab="some y variable",pch=21, bg="blue")
   points(x,z,pch=21,bg="red")
   legend("topleft",c("X","Y"),pch=21,pt.bg=c("blue","red"))
}

> a<-rnorm(15)
> b<-2*a
> c<-1.5*a - 2

> plot4me(a,b,c)
```




When you write functions like this, it often makes sense to retain the option for the user to modify some of the arguments within the nested function(s). For example, with the above function plot4me, there are several optional plotting functions that cannot be modified, e.g. `cex`, `mgp`. We can leave room for these arguments by adding `...` (an ellipsis) to the end of the argument list when you define your function.

```
plot4me<-function(x,y,z,...) {
   plot(x,y,xlab="Some x variable",ylab="some y variable",
        pch=21,bg="blue",...)
   points(x,z,pch=21,bg="red",...)
   legend("topleft",c("X","Y"),pch=21,pt.bg=c("blue","red"))
}
```

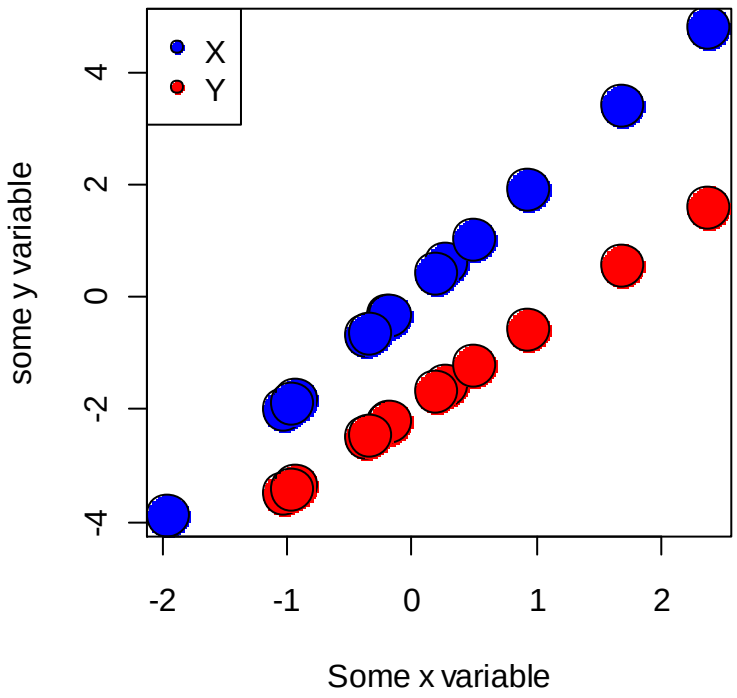
This code tells R that if there are any additional arguments passed to the function `plot4me` (in addition to `x`, `y`, and `z`), they should be passed to both `plot` and `points`. This is very handy, and saves a lot of code. One warning: any additional arguments will be passed to all the nested functions that contain `...`, so there is a bit of inflexibility here.

```
plot4me<-function(x,y,z,...) {
   plot(x,y,xlab="Some x variable",ylab="some y variable",pch=21,
        bg="blue",...)
   points(x,z,pch=21,bg="red",...)
```

```
    legend("topleft",c("X","Y"),pch=21,pt.bg=c("blue","red"))
}
```

Now we can modify the `cex` argument.

```
> plot4me(a,b,c,cex=3)
```




```
> plot4me(a,b,c,cex=3,log="x")
Warning messages:
1: In xy.coords(x, y, xlabel, ylabel, log) :
  4 x values <= 0 omitted from logarithmic plot
2: In plot.xy(xy.coords(x, y), type = type, ...) :
  "log" is not a graphical parameter
```

It is very easy to make a vectorized function—simply use expressions that carry out vectorized operations. Here is a simple example for converting lb (mass) to kg.

```
> lb2kg<-function(x) 0.453592*x
> weights<-rnorm(10,mean=175,sd=9)
> weights
 [1] 180.2066 175.0223 167.4249 167.5359 168.7322 165.2813 174.4403
175.7840 171.4901
[10] 167.9036
```

```
> lb2kg(weights)
[1] 81.74029 79.38870 75.94259 75.99295 76.53558 74.97028
 [7] 79.12472 79.73420 77.78655 76.15971
```

A slightly more complicated example is given below. The `alk2ic` function converts an alkalinity to a dissolved inorganic carbon concentration.

```
alk2ic<-function(alk,pH) {
   H<-10^-pH
   K1<-10^-6.352
   K2<-10^-10.329
   2E-5*alk*(((H/K1)+1+(K2/H))/(1+2*(K2/H)))
}

> alkalinity<-c(55,62.1,45.1)
> ph<-c(7.23,8.10,7.75)

> alk2ic(alkalinity,ph)
[1] 0.0012445716 0.0012566845 0.0009355203
```

The flow control structions that were discussed earlier can, of course, be used in functions.

```
alk2ic<-function(alk,pH) {
   if (max(alk)<500) {
      H<-10^-pH
      K1<-10^-6.352
      K2<-10^-10.329
      2E-5*alk*(((H/K1)+1+(K2/H))/(1+2*(K2/H)))
   } else print("Error: alkalinity too high for dilute solution assumption")
}

> alkalinity<-c(55,62.1,45.1)
> ph<-c(7.23,8.10,7.75)

> alk2ic(alkalinity,pH)
[1] 0.0012445716 0.0012566845 0.0009355203
> alkalinity<-c(550,62.1,45.1)
> alk2ic(c(alkalinity,700),c(pH,9.5))
[1] "Error: alkalinity too high for dilute solution assumption"
```

Here is a different way of solving the same problem:

```
alk2ic<-function(alk,pH) {
   ifelse (alk<500, {
      H<-10^-pH
      K1<-10^-6.352
      K2<-10^-10.329
      2E-5*alk*(((H/K1)+1+(K2/H))/(1+2*(K2/H)))
   },NA)
}

> alk2ic(alkalinity,ph)
[1]           NA 0.0012566845 0.0009355203
```

When you are writing simple functions, such as those shown above, it is pretty easy to debug your function. If you have a more complicated function, it can be very difficult to isolate a problem in your code, since the function acts like black box. However, you can use the `debug` function to step through your function code one line at a time, all the while being able to see the value of internal variables. For example, to debug the `alk2ic` function, submit:

```
> debug(alk2ic)
```

and then call it up. Once you have flagged a function using `debug`, you won't notice anything different until you call it up. To move through the function, just hit enter. R will print the line of code that it is about to submit on the line before the prompt. You can have R evaluate any other code by typing it at the prompt.

## *Exercises*

1. Write a function for calculting the root mean square error (RMSE), which is often used to compare model predictions to observed values. RMSE is given by

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}\left(x_{p,i} - x_{o,i}\right)^2}{n}}$$

where $x_{p,i}$ is i$^{th}$ predicted value, and $x_{o,i}$ is the i$^{th}$ observed value. Generate some model predictions somehow, and test your function.

2. The density of puer water (in kg/L or g/mL) can be approximated by the following equation.

$$0.9999 + 4.892\times10^{-5}T - 7.410\times10^{-6}T^2 + 3.998\times10^{-8}T^3 - 1.233\times10^{-10}T^4$$

where $T$ = temperature in °C. Write a function to calculate the density of water given a temperature and a temperature unit. Include the option to use °C, °F, and K. Note that °C = (°F – 32)×9/5 = K – 273.15. Test your function to make sure it works, and that it is vectorized.

3. R does not have a function in the base packages for error bars, but they can easily be added using `arrows`. Write a function that will add error bars to either a plot, and test it out.

4. This is tricky. The merge function is really handy, but one limitation is that it can only merge two data frames at once. Try to come up with a function that can merge any number of data frames. Test your function. Hint: we recommend that you store the data frames as lists.

# 14. Generalized linear models

*Crawley 2007: Chapter 13, R-Intro: Section 11.6.2, Faraway 2005b*

## *14.1. The glm function*

Generalized linear models (GLMs) are a very flexible class of statistical models. The generic form of a GLM is given by

$$g\left(E\left(Y\right)\right) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 ... + \beta_m x_m$$

where $g$ is the link function, $E\left(Y\right)$ represents the expected values of the response variable (i.e., the observed values minus the error) , $x_1$ through $x_m$ are predictors, and $\beta_0$ through $\beta_m$ are coefficients. In R, GLM models can be specified using the `glm` function. There are eight different error distributions available in `glm`, including binomial and poisson, each with a default link function. Arguments and default argument values can be found in the help file for `glm`:

```
glm(formula, family = gaussian, data, weights, subset,
    na.action, start = NULL, etastart, mustart,
    offset, control = glm.control(...), model = TRUE,
    method = "glm.fit", x = FALSE, y = TRUE, contrasts = NULL,
    ...)
```

The glm function can be used for data that do not follow the normal distribution. For example, when a response variable is a binomial (e.g. presence/absence, dead/alive) or a proportion (i.e. a value between zero and unity). In this case, errors are not normally distributed, since values greater than unity or less than zero are not possible (instead we need to use a binomial distribution), and the relationship between a predictor and response variable is not linear (but can be linearlized by applying the logit transformation). To carry out logistic regression in `glm`, we just need to specify a binomial distribution and a logistic link function (the logistic link function is the default for the binomial distribution in `glm)`. You can see a list of the available distributions and their default link functions in the help file for `family`.

```
family(object, ...)

binomial(link = "logit")
gaussian(link = "identity")
Gamma(link = "inverse")
inverse.gaussian(link = "1/mu^2")
poisson(link = "log")
quasi(link = "identity", variance = "constant")
quasibinomial(link = "logit")
quasipoisson(link = "log")
```

To demonstrate, let's read in the results of a single Cu toxicity test with *Daphnia magna*.

```
> tox.dat<-read.table("Cu_tox_test.txt",header=T)
> tox.dat
     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
```

With the data in `tox.dat`, we can specify the model using one of two forms—either specify the response as a fraction and specify an argument `weights`, which is the total number of organisms, so that R can calculate the number dead and alive, or we can use a matrix with the number dead and the number alive as the response variable. Both methods are shown.

```
> tox.dat$dead<-tox.dat$tot-tox.dat$alive
> tox.dat$prop.dead<-tox.dat$dead/tox.dat$tot
> tox.dat
     cu alive tot dead prop.dead
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
```

Now for the regression.

```
> mod.1<-glm(prop.dead~cu,binomial,weights=tot,data=tox.dat)
> summary(mod.1)

Call:
glm(formula = prop.dead ~ cu, family = binomial, data = tox.dat,
    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
```

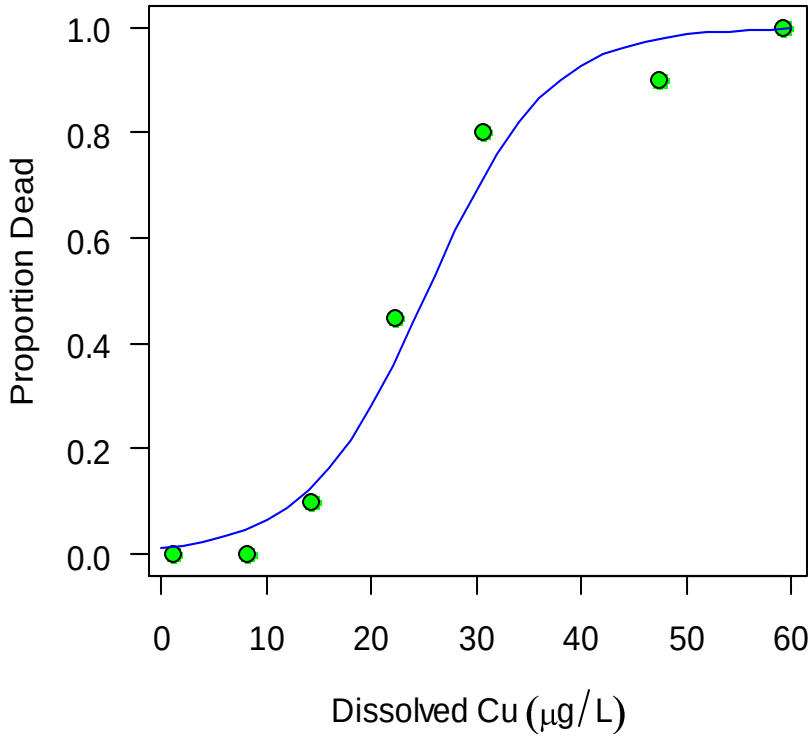
```
Number of Fisher Scoring iterations: 5
```

For the other (matrix) method, we can create a matrix using `cbind`.

```
> resp<-cbind(tox.dat$dead,tox.dat$alive)
> mod.2<-glm(resp~cu,binomial,data=tox.dat)
> summary(mod.2)

Call:
glm(formula = resp ~ cu, family = binomial, data = tox.dat)

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
```

Let's take a look at the data and model predictions.

```
> predict.dat<-data.frame(cu=seq(0,60,2))
> predict.dat$prop.dead<-predict(mod.1,newdata=predict.dat,
+ type="response")

> plot(tox.dat$cu, tox.dat$prop.dead,xlab=expression("Dissolved Cu"
+ ~(mu*g/L)),ylab="Proportion Dead",las=1,pch=21,bg="green",cex=1.2)
> lines(predict.dat$cu,predict.dat$prop.dead,col="blue")
```




The `glm` function is very flexible. To demonstrate a very different application, let's read in the data on insect numbers in response to insecticide spraying. This data set was analyzed above using an ANOVA, but recall that it required a transformation of the response.

```
> insects.dat<-InsectSprays
> summary(insects.dat)
     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
```

In this case, we want to carry out an ANOVA, but the GLM lets us use an appropriate distribution for count data: the Poisson distribution. Note that the default link function for the Poisson distribution is log.

```
> mod.1<-glm(count~spray,poisson,data=insects.dat)
> summary(mod.1)

Call:
glm(formula = count ~ spray, family = "poisson", data = insects.dat)
```

```
Deviance Residuals:
    Min       1Q   Median       3Q      Max
-2.3852  -0.8876  -0.1482   0.6063   2.6922

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  2.67415    0.07581  35.274  < 2e-16 ***
sprayB       0.05588    0.10574   0.528    0.597
sprayC      -1.94018    0.21389  -9.071  < 2e-16 ***
sprayD      -1.08152    0.15065  -7.179 7.03e-13 ***
sprayE      -1.42139    0.17192  -8.268  < 2e-16 ***
sprayF       0.13926    0.10367   1.343    0.179
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for poisson family taken to be 1)

    Null deviance: 409.041  on 71  degrees of freedom
Residual deviance:  98.329  on 66  degrees of freedom
AIC: 376.59

Number of Fisher Scoring iterations: 5
```

To get an ANOVA table, we can use the `anova` function.

```
> anova(mod.1,test="Chisq")
Analysis of Deviance Table

Model: poisson, link: log

Response: count

Terms added sequentially (first to last)


      Df Deviance Resid. Df Resid. Dev P(>|Chi|)
NULL                     71     409.04
spray  5   310.71        66      98.33 4.979e-65
```

To get diagnostic plots, use the `plot` function.

```
> plot(mod.1)
```

## *Exercises*

1. Organism response to metal is often a function of the logarithm of metal concentration. Repeat the above example on logistic regression, but use $\log_{10}$(cu) as the predictor variable. Additionally, try to fit a probit regression model to the data (just use `binomial(link="probit")`). Which model provides the best fit? Plot measured data and predictions from all three models.

2. The file Squirrel_color.txt contains observations on squirrel color in and near Syracuse. Carry out logistic regression to determine if the proportion of squirrels that are black increases as one moves toward the city center. Make sure you take a look at the data to decide how to specify the model formula. Try to plot the data to see if it is consistent with the model results.

3. The data frame `esoph`, from the `datasets` package, contains data on a case-control study of esophogeal cancer. Determine if age group, alcohol consumption, and tobacco usage had an effect on the occurrence of cancer. Check out the help file for the data set to get more information on the predictor variables if needed. Note that the predictors are ordered factors—R will automatically use orthogonal polynomials for regression (as you can see by applying the `model.matrix` function).

# 15. Generalized additive models

*Crawley 2008: Chapter 18*

## *15.1. The gam function*

If `glm` is not flexible enough for you, more flexibility can be found using generalized additive models (GAMs). GAMs can be used for analyzing the relationships between a continuous dependent variable and one to many predictors. A generic form for a GAM is given in the following equation.

$$g\left(E\left(Y\right)\right) = \beta_0 + f_1\left(x_1\right) + f_2\left(x_2\right) \ldots + f_m\left(x_m\right)$$

Here, $E\left(Y\right)$ is the expected value of the dependent variable, $g$ is the link function (analogous to the link function in GLM), and $f_1$ through $f_m$ are parametric (e.g. a linear response) or nonparametric (e.g. a smoothing function) functions that are applied to the predictor variables $x_1$ through $x_m$. The flexibility of GAMs make this approach useful for cases where relationships between variables is complex and not easily captured by a typical linear or nonlinear model, or where there is no reason to assume that the relationship between variables should follow a particular form. Unlike the other approaches that we have discussed for modeling the effect of continuous predictors on a dependent variable, generalized additive models (GAM) do not require the assumption of any particular mathematical relationship between predictor(s) and a dependent variable.

Multiple packages provide functionality for fitting GAMs. We use the `gam` function from the package `mgcv` package. As with `glm`, `gam` is a very flexible function, and the following treatment only scratches the surface.

We need to install and load the `mgcv` package.

```
> install.packages("mgcv")
...
> library(mgcv)
This is mgcv 1.4-1.1
```

Also install the package `gamair`, which contains the data sets used in Wood (2006).

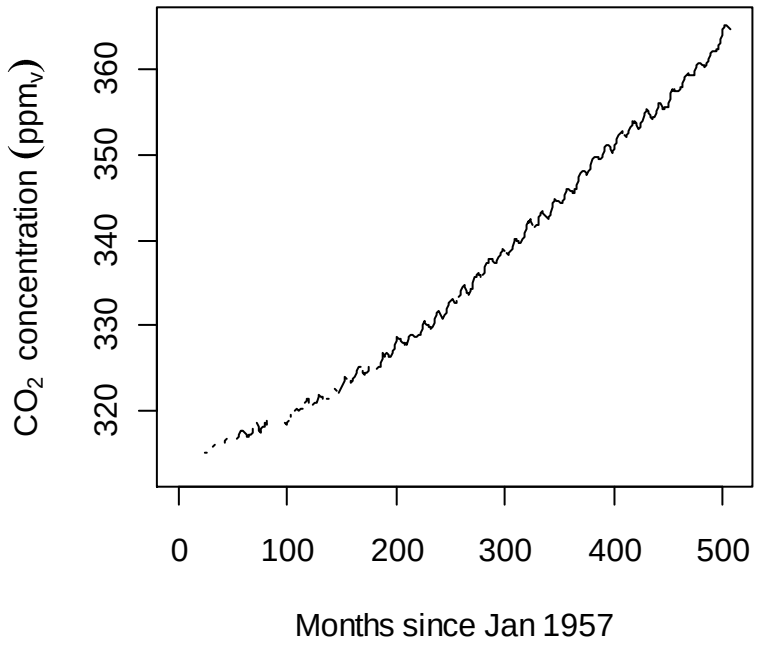
```
> install.packages("gamair")
> library(gamair)
```

You can see a list of the data sets available by typing

```
> ?gamair
```

Let's take a look at the `co2s` data set, which contains atmospheric $CO_2$ concentrations at the South Pole for the last 50 years or so.

```
> data(co2s)
> summary(co2s)
      co2           c.month          month
 Min.  :313.2   Min.  :  1.0   Min.  : 1.000
 1st Qu.:325.1   1st Qu.:127.5   1st Qu.: 3.000
 Median :337.7   Median :254.0   Median : 6.000
 Mean   :338.2   Mean   :254.0   Mean   : 6.473
 3rd Qu.:351.2   3rd Qu.:380.5   3rd Qu.: 9.000
 Max.  :365.2   Max.  :507.0   Max.  :12.000
 NA's   : 80.0

> plot(co2 ~ c.month,type="l",xlab="Months since Jan 1957",
+ ylab=expression(CO[2]~~"concentration"~~(ppm[v])),data=co2s)
```




With the `gam` function, model formulae are expressed similarly to `lm` and `glm`. From the help file for the gam function:

```
gam(formula,family=gaussian(),data=list(),weights=NULL,subset=NULL,
    na.action,offset=NULL,control=gam.control(),method=gam.method(),
    scale=0,knots=NULL,sp=NULL,min.sp=NULL,H=NULL,gamma=1,
    fit=TRUE,paraPen=NULL,G=NULL,in.out,...)
```

When specifying a formula in `gam`, use `s(x1)` to indicate that a smoothing function should be applied to predictor `x1`.

OK, so let's fit a GAM. Note that there are some NAs in the data set—we are going to remove them from the start.

```
> co2.dat<-na.omit(co2s)
> mod.1<-gam(co2 ~ c.month + s(month), data = co2.dat)
```

In this model, we are assuming a linear response to the time since 1957, and a smoothed response to the month of the year.

```
> summary(mod.1)

Family: gaussian
Link function: identity

Formula:
co2 ~ c.month + s(month)

Parametric coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 3.076e+02  1.962e-01  1568.0   <2e-16 ***
c.month     1.074e-01  6.225e-04   172.5   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Approximate significance of smooth terms:
           edf Ref.df     F  p-value
s(month) 3.342  3.842 5.035 0.000688 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

R-sq.(adj) =  0.986   Deviance explained = 98.6%
GCV score =  3.034   Scale est. = 2.996     n = 427
```

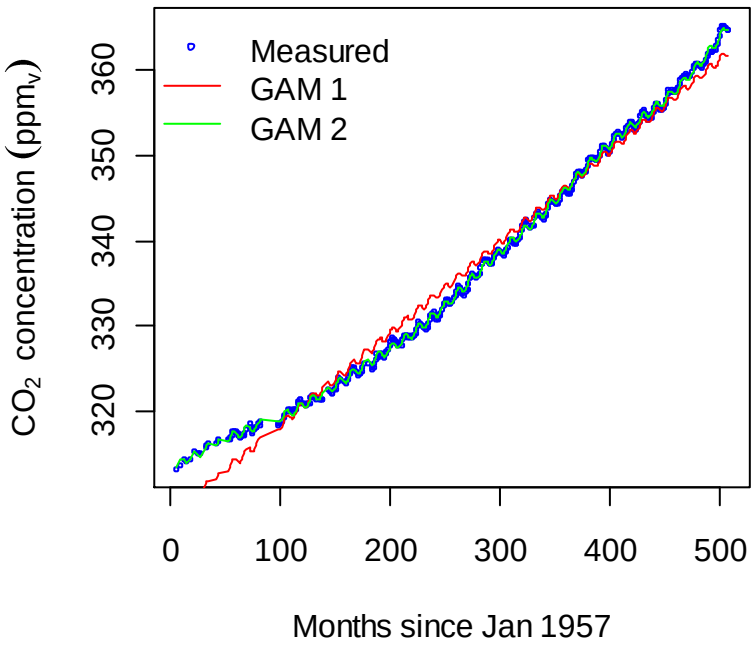
Let's try an alternate model.

```
> mod.2<-gam(co2 ~ s(c.month) + s(month), data = co2.dat)
> summary(mod.2)

Family: gaussian
Link function: identity

Formula:
co2 ~ s(c.month) + s(month)

Parametric coefficients:
             Estimate Std. Error t value Pr(>|t|)
```

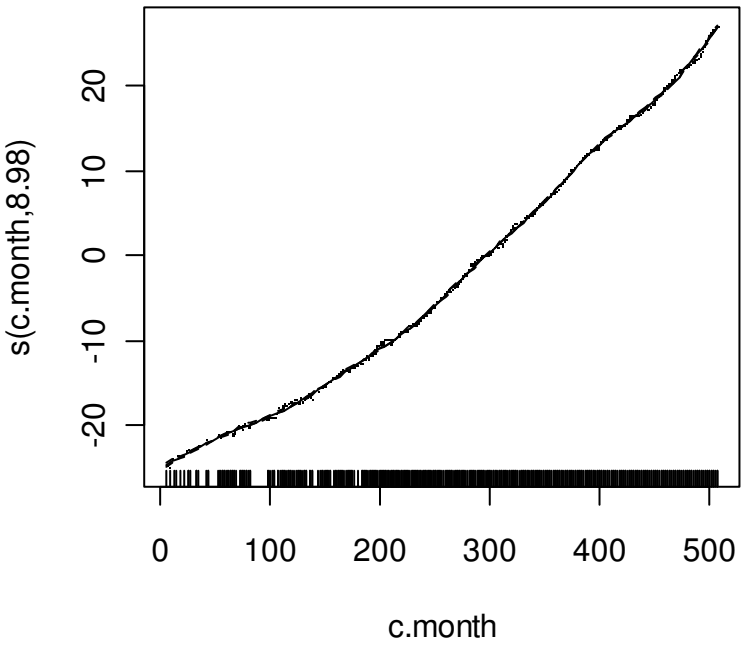
```
(Intercept) 338.24515    0.01308   25869   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Approximate significance of smooth terms:
             edf Ref.df        F p-value
s(c.month) 8.980  9.480 130546.9  <2e-16 ***
s(month)   5.921  6.421    132.9  <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

R-sq.(adj) =      1   Deviance explained =  100%
GCV score = 0.075826   Scale est. = 0.073002  n = 427
```

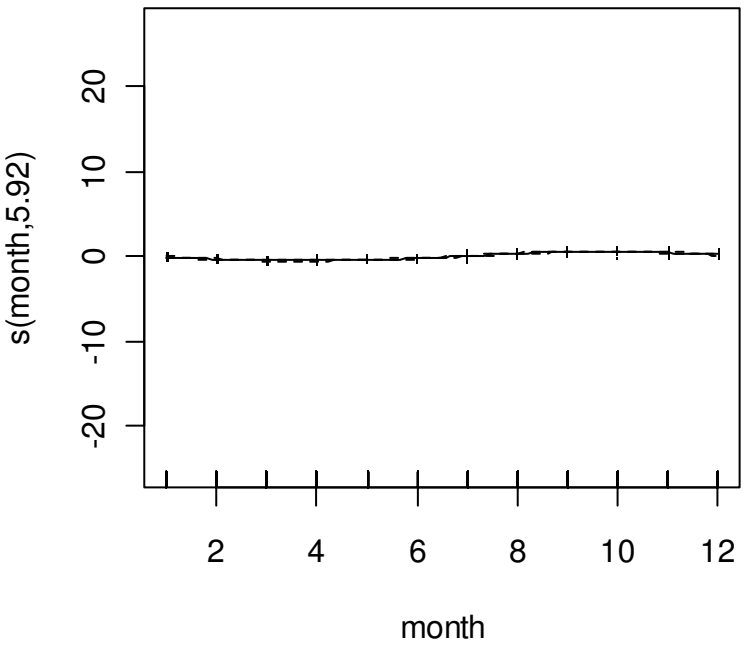
How do the models look?

```
> plot(co2 ~ c.month,pch=1,cex=0.4,col='blue',
+ xlab="Months since Jan 1957",ylab=expression(CO[2]~
+ ~"concentration"~~(ppm[v])),data=co2.dat)
> lines(co2.dat$c.month,predict(mod.1),col='red')
> lines(co2.dat$c.month,predict(mod.2),col='green')
> legend("topleft",c("Measured","GAM 1","GAM 2"),pch=1,
+ pt.cex=c(0.5,0,0),lty=c(0,1,1),col=c("blue","red","green"),bty='n')
```




```
> plot(mod.2,residuals=T)
```


s(c.month,8.98)
20
10
0
-10
-20
0
100
200
300
400
500
c.month


s(month,5.92)
20
10
0
-10
-20
2
4
6
8
10
12
month

While GLMs and GAMs are very flexibles approaches for statistical modeling, they certainly don't represent the full range of statistical models that can be carried out in R. R can also be used for mixed effect models (`lme4`, `nlme`, and `mgcv` packages), tree-based models (`tree` package), and local regression models (`loess` function). These topics are covered in the respective package documentation and several books on R. See the information in the section on R documents below.

## *Exercises*

1. The file isolation.txt (from Crawley 2008) contains data on the presence of a particular species of bird on some islands. Apply a GAM to quantify and test the effects of island size and isolation (distance from mainland) on the presence of this species.

2. The data frame `mtcars` contains data (from 1974) on cars' fuel economny. Use GAM to quantify and test the effects of the potential predictors on fuel economy. See the help file for `mtcars` for more information.

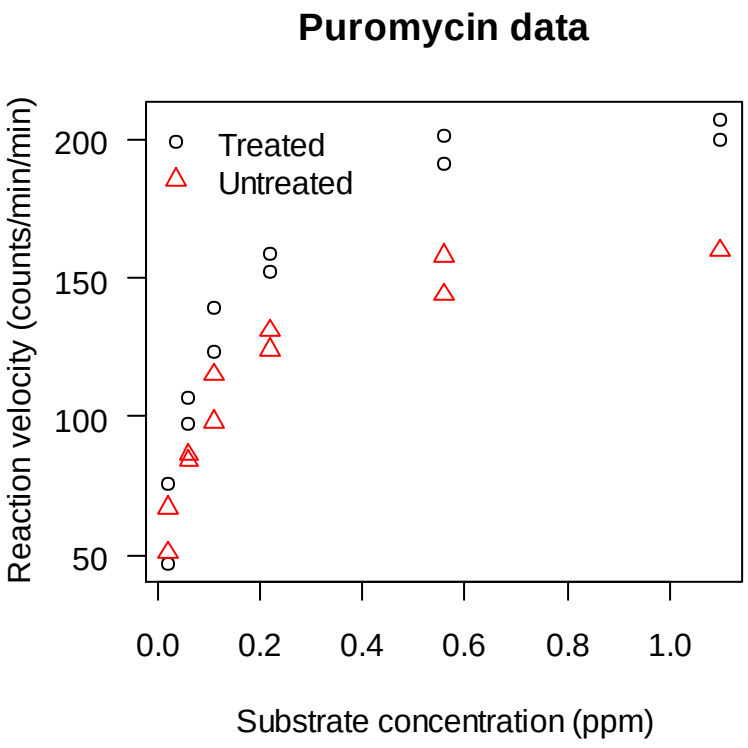
# 16. Nonlinear regression

*Dalgaard 2008: Chapter 16, Ritz & Streibig 2009*

## *16.1. The nls function*

R has some powerful algorithms for nonlinear regression. Let's demonstrate this with a data frame called `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.

```
> puromycin.dat<-Puromycin

> plot(rate ~ conc, las = 1, xlab = "Substrate concentration (ppm)",
+ ylab = "Reaction velocity (counts/min/min)", pch=
+ as.numeric(state),col=as.numeric(state),main="Puromycin data",
+ data=puromycin.dat)
> legend("topleft",c("Treated","Untreated"),pch=1:2,col=1:2,bty="n")
```




To model these data, let's use the Michaelis-Menten equation:

$$y = \frac{V_{\max} x}{1 + K_m x}$$

The asymptote $V_{max}$ is the maximum reaction velocity, and $K_m$ is the Michaelis-Menten constant. Let's fit a model to the "treated" group—as with `lm` and other functions, this can be specified in the `subset` argument in the `nls` function call, but we will just make a subset of the original data frame that contains only these data.

```
> puromycin.dat<-subset(Puromycin,state=="treated")
> mod.1<-nls(rate ~ Vm*conc/(K + conc),data=puromycin.dat,
+ start=c(Vm=250,K=0.1),trace=TRUE)
3993.976 :  250.0   0.1
1196.486 :  212.02378920   0.06342989
1195.456 :  212.63961223   0.06405238
1195.449 :  212.67946886   0.06411461
1195.449 :  212.68333056   0.06412064
1195.449 :  212.68370347   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).

```
> summary(mod.1)

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: 8.813e-06
```

The extractor functions that we used earlier for `lm` output can also be used with `nls` output. For example:

```
> coef(mod.1)
          Vm            K
212.68357944   0.06412103

> predict(mod.1)
 [1]  50.56601  50.56601 102.81096 102.81096 134.36161 134.36161
 [7] 164.68469 164.68469 190.83292 190.83292 200.96883 200.96883
```

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

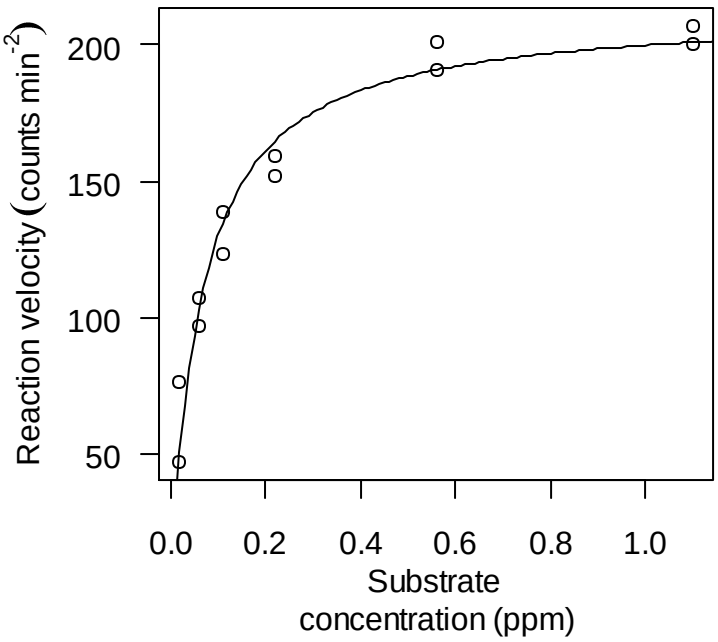
values, `nls` will probably do a good job. If you provide very poor guesses, nls may not be able to find least-squares estimates. For some common models that can be linearized, R has self-starting functions, which estimate starting values automatically using data transformations and linear regression. Here is an example of the models fit in the example above, but this time, a self-starting function (`SSmicmen`) is used:

```
> mod.2<-nls(rate ~ SSmicmen(conc,Vm,K),data=puromycin.dat)
> summary(mod.2)

Formula: rate ~ SSmicmen(conc, Vm, K)

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: 0
Achieved convergence tolerance: 1.917e-06
```

Comparing the results of the model fitting exercises, we see that the self-starting function provided the same parameter estimates as the `nls` in which we had to specify initial guesses.

Now let's make some model predictions and show them on a new figure.

```
> mod.dat<-data.frame(conc=seq(0,1.2,0.01))
> mod.dat$rate.pred<-predict(mod.1,newdata=mod.dat)
> plot(puromycin.dat$conc,puromycin.dat$rate,xlab="Substrate
+ concentration (ppm)",ylab=expression("Reaction velocity"~~
+ (counts~min^"-2")),las=1)
> lines(mod.dat$conc,mod.dat$rate.pred)
```




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 Crawley (2007: Table 20.1) and Ritz & Steibig (2008: Table B.1). Several of these models have associated self-starting functions in R.

Asymptotic models:
Michaelis-Menton
2-parameter asymptotic exponential
3-parameter asymptotic exponential

S-shaped models:
2-parameter logistic
3-parameter logistic
4-parameter logistic
Weibull
Gompertz

Humped curves:
Ricker curve
First-order compartment
Bell-shaped
Biexponential

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.

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 algoritm, which may be more reliable. We have had good luck with `nls.lm` on problems that `nls` has had difficultly with.

## *Exercises*

1. Read in the data in the file Dimethyl-death.txt. This file contains fabricated data on the concentration of the toxicant dimethyl-death in a water body over time, following a spill. Time (t) is given in days, while concentration is in μg/L. Fit a first-order decay model to these data using the function nls. First-order decay is described by:

$$c = c_0 e^{-\lambda t}$$

where $c$ is the concentration at time $t$, $c_0$ is the initial concentration, and $\lambda$ is the decay constant. Calculate the half-life ($\frac{\ln(2)}{\lambda}$). Be sure to use the `coef` function. Plot the data and the model predictions. Hint: if you need some help in estimating the starting value for $\lambda$, note that the slope of ln($c$) vs. $t$ provides one. Try fitting a similar model, but with some low "background" concentration of dimethyl-death.

2. The data set DNase, from the datasets package, contains data on the measured optical density of protein solutions. Create a subset that consits of only a single run, and fit a logistic model using a self-starter function (see the help file for `SSlogis` for more information).

# 17. Survival Analysis

*Dalgaard 2002: Chapter 12, Crawley 2008: Chapter 25*

## *17.1. Log-rank test and Cox proportional hazards model*

The analysis of lifetimes is an important topic within many fields of study including biology, medicine, toxicology, and engineering. In ecotoxicology, standardized toxicity tests often consist of dosing studies designed to determine an LC50 for a substance when organisms are exposed for fixed duration (e.g. a 96-h LC50). Survival analysis allows for the more rigorous investigation of the combined effects of concentration and exposure duration on lifetimes. R supports a wide range of tools for the analysis of survival data, including methods to evaluate the effects of multiple treatments (both continuous and discrete) simultaneously.

The organization of survival data for analysis in R is geared toward what you might expect with human-based data; data are organized so that each row represents an individual subject (e.g. person). It is essential to understand that we are dealing with time to death in the following examples, not simply the fraction of individuals surviving.

When analyzing survival data, it is necessary to indicate when subjects are removed from the study due to something other than death by the cause of interest. This is referred to as censoring in survival analysis. An example of censoring that is relevant to aquatic toxicity testing is a daphnid being crushed by a pipette during the course of the experiment. This death would not be attributed to the toxicant, so the organism would have to be censored from the dataset. Another case of censoring involves organisms that are still alive by the end of the experiment. We do not know when these organisms will die, so we do not have a complete understanding of their lifetime.

For this we will use two experiments on survival of *Daphnia* exposed to copper.

```
> install.packages('survival')
> library(survival)
> daph.surv<-read.table("Daphnids_surv.txt", header=T)
> attach(daph.surv)
> daph.surv

   exp.id c.cu tt.death status
1      1a   64        2      1
2      1a   64        2      1
3      1a   64        2      1
4      1a   64        2      1
5      1a   64        3      1
6      1a   64        3      1
7      1a   64        3      1
8      1a   64        3      1
9      1a   64        3      1
10     1a   64        4      1
11     1a   64        4      1
12     1a   64        4      1
```

```
13      1a    64         9       1
14      1a    64         9       1
15      1a    64        21       0
16      1a    64        21       0
17      1a    64        21       0
18      1a    64        21       0
19      1a    64        21       0
20      1a    64        21       0
...
40      1b    48        21       0
```

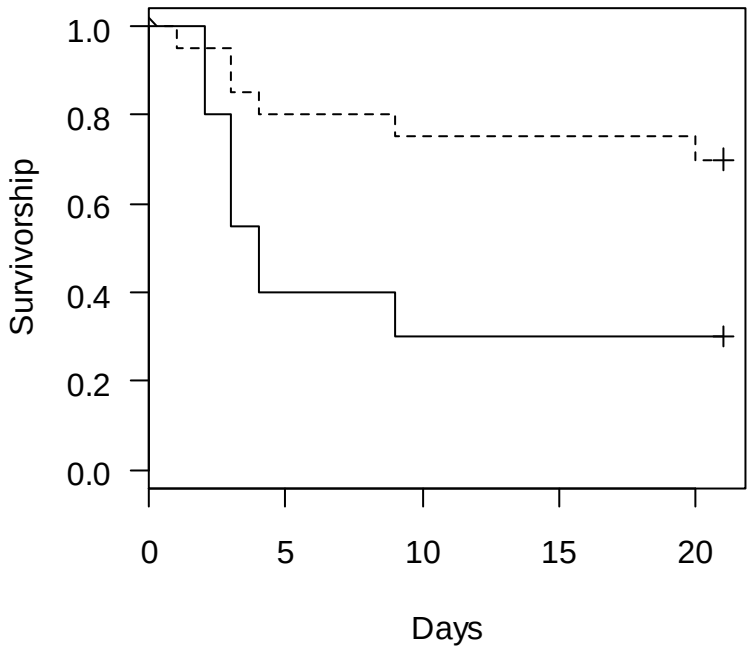
This data frame contains two blocks of 20 rows (although only the first set of complete blocks is shown). Each row corresponds to a single organism, and in column `tt.death`, the time to death (in days) of that organisms is recorded. In this example, the column `status` contains a value of 1 if the organisms was dead at the end of the experiment, and a value of 0 if the organism did not die. This is used for censoring.

The function `survfit` is used to generate an estimate of a survival curve (i.e. the trajectory of survival over time). This function requires at least a survival object that contains the data. The function `Surv` can be used to generate a survival object, which simply combines the time to death and censor data into a single vector.

```
> mod1=survfit(Surv(tt.death,status)~exp.id)
> summary(mod1)
Call: survfit(formula = Surv(tt.death, status) ~ exp.id)

                exp.id=1a
 time n.risk n.event survival std.err lower 95% CI upper 95% CI
    2     20       4     0.80  0.0894        0.643        0.996
    3     16       5     0.55  0.1112        0.370        0.818
    4     11       3     0.40  0.1095        0.234        0.684
    9      8       2     0.30  0.1025        0.154        0.586

                exp.id=1b
 time n.risk n.event survival std.err lower 95% CI upper 95% CI
    1     20       1     0.95  0.0487        0.859        1.000
    3     19       2     0.85  0.0798        0.707        1.000
    4     17       1     0.80  0.0894        0.643        0.996
    9     16       1     0.75  0.0968        0.582        0.966
   20     15       1     0.70  0.1025        0.525        0.933
```

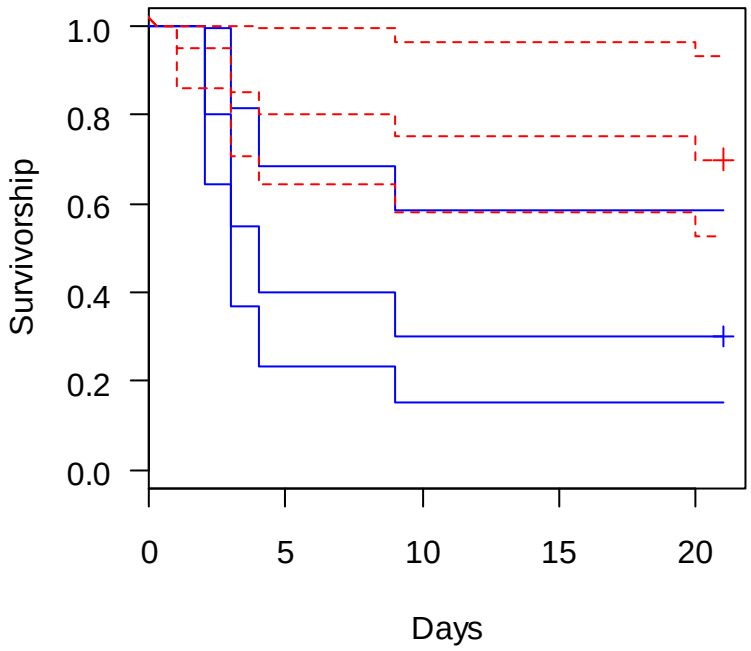
This is a tabular summary of the survival function at the times when a death occurred. The output corresponds to the time at which an event occurred (i.e. a death), the number at risk of dying during that time interval (the number present), the number dying during that time interval, and the value of the survival function at each time interval. This is typically viewed graphically, and can be accomplished as follows:

```
> plot(mod1,ylab="Survivorship",xlab="Days",lty=c(1,2),las=1)
```




This shows the results for `exp.id=1a` as a solid line and the results for `exp.id=2` as a dashed line. Note that the end of the survival curve shows a "+" symbol, which indicates where censoring occurred. In both experiments, some organisms still remained alive. When a single survival curve is plotted, 95% confidence intervals are included by default, but we can also force them to be added to our current plot by specifying the argument `conf.int=T`. This can get confusing when there are multiple curves plotted, but you can specify different colors for each series:

```
> plot(mod1,ylab="Survivorship",xlab="Days",conf.int=T,
+ col=c("blue","red"), lty=c(1,2),las=1)
```




R contains a tool for comparing different survival curves, which is much more desirable than comparing various point estimates from the curves. In this case, we might want to test if the two survival curves shown above are identical. For this, R uses the log-rank test, which is invoked with the `survdiff` function:

```
> survdiff(Surv(tt.death,status)~exp.id)
Call:
survdiff(formula = Surv(tt.death, status) ~ exp.id)

            N Observed Expected (O-E)^2/E (O-E)^2/V
exp.id=1a 20       14      8.6      3.38      6.82
exp.id=1b 20        6     11.4      2.55      6.82

 Chisq= 6.8  on 1 degrees of freedom, p= 0.00901
```

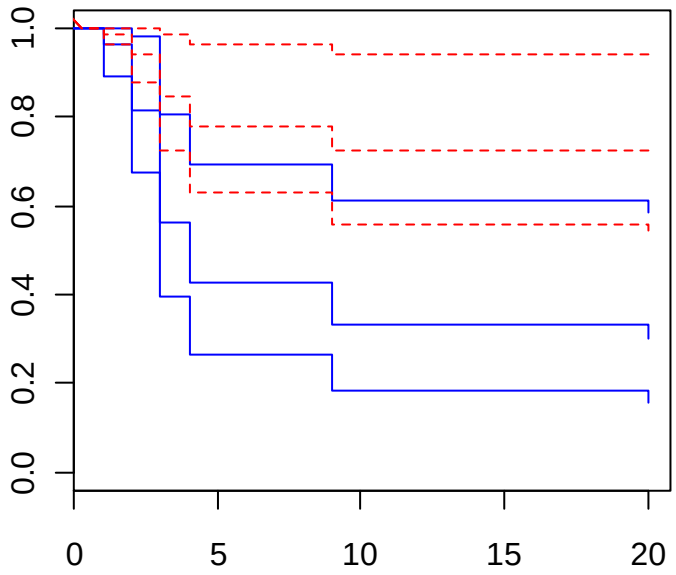
From this analysis, we would conclude that the survival curves do appear to be different.

There are other methods for conducting survival analysis in R, including the Cox proportional hazards model (semi-parametric) and Accelerated Failure Time (AFT) models (parametric). Cox's Proportional-Hazards Model uses a baseline hazard function $\alpha(t)$ (hazard is the instantaneous risk of death), which can take any form. Covariates (i.e. predictor variables) are assumed to have a linear effect on hazard, and the relative effect of a covariate on hazard is always constant (i.e. the ratio of hazards is independent of time, hence the name proportional-hazards). The Cox proportional-hazards model can be applied using the `coxph` function.

```
> mod2<-coxph(Surv(tt.death,status)~exp.id)
> summary(mod2)
Call:
coxph(formula = Surv(tt.death, status) ~ exp.id)

  n= 40
          coef exp(coef) se(coef)     z     p
exp.id1b -1.23     0.292    0.492 -2.5 0.012

         exp(coef) exp(-coef) lower .95 upper .95
exp.id1b     0.292       3.42     0.111     0.767

Rsquare= 0.159   (max possible= 0.967 )
Likelihood ratio test= 6.94  on 1 df,   p=0.00844
Wald test            = 6.24  on 1 df,   p=0.0125
Score (logrank) test = 7.02  on 1 df,   p=0.00806
```

To plot model predictions, we can use the `survfit` function within a `plot` command, as above. Note that we need to use a new data frame to generate model predictions for both treatments.

```
> cox.plot.data<-data.frame(exp.id=c('1a','1b'))
> plot(survfit(mod2,newdata=cox.plot.data),conf.int=T,
+ col=c("blue","red"),lty=c(1,2))
```



### *Exercise*

1. Use the dataset `rats`, which is included with the `survival` package, to fit a Cox's proportional hazard model. This dataset has survival times (`time`) for rats that were injected with a carcinogen, and then treated with one of two drugs (`rxn` = 1 or 0). Fit a Cox's proportional hazard model to these data to determine if the drugs had differing effects on survival time.

# 18. Distributions and simulations

*Crawley 2007: Chapter 7, Dalgaard 2008: Chapter 3, Kuhnert & Venables 2005: pp. 38-40*

## *18.1. Available distributions*

Many of the distributions associated with statistical modeling have been built into R. There are nearly 30 such distributions, including normal (`norm`), *t* (`TDist`), and *F* (`FDist`). These distributions can be used for simulating data, determining quantiles, probabilities, and density functions. For each distribution, R has four functions available, the names of which start with a prefix (`p`, `q`, `d`, `r`, which represent distribution, quantile, density, and random, respectively) and the distribution name. For example, to calculate a cumulative probability for the normal distribution:

```
> pnorm(2.5,mean=0,sd=1)
[1] 0.9937903
```

The first argument is a quantile or a vector of quantiles, which simply represent the number of standard deviations from the mean.

Suppose that we wanted to know the cumulative probabilities associated with a vector of quantiles representing 1, 2, and 3 standard deviations above and below the mean (i.e. *z* values):

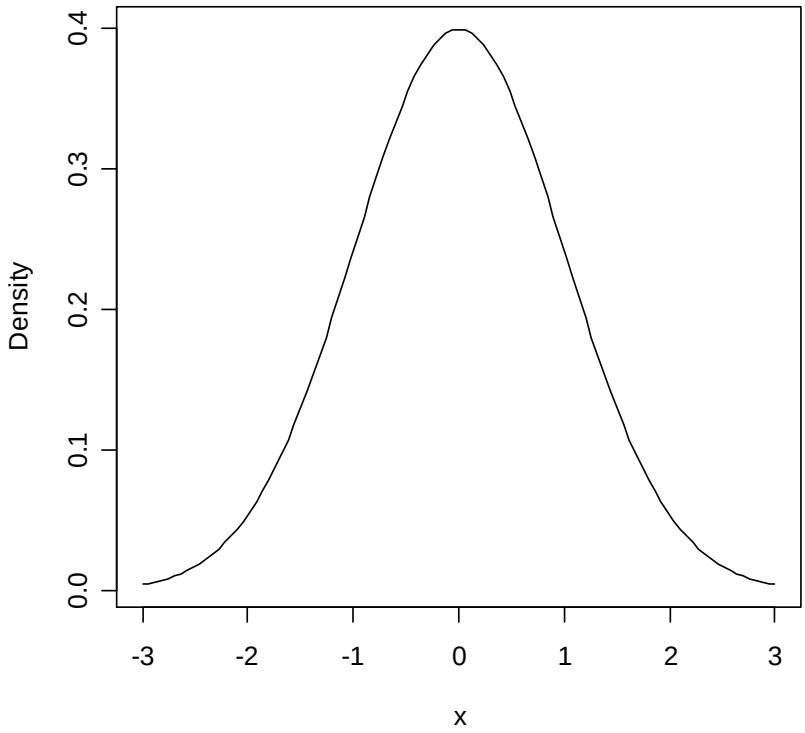
```
> pnorm(-3:3)
[1] 0.001349898 0.022750132 0.158655254 0.500000000 0.841344746
[6] 0.977249868 0.998650102
```

We can calculate two-tail results by difference. For example, for the probability of obtaining a value that was within one standard deviation of the mean:

```
> pnorm(1)-pnorm(-1)
[1] 0.6826895
```

The density function is not used as frequently as the other four functions associated with the distributions, but one of its uses is to provide the well-known shape of various distributions. In the case of the normal distribution, this is of course, the bell-shaped curve. The probability density represents the slope of the cumulative probability distribution. The area under a specified section of the density curve represents the probability of obtaining a value from within that interval. However, we just demonstrated that it was easy to do that with `pnorm`.

```
> curve(dnorm(x),-3,3,ylab="Density")
```




Either a single quantile or a vector of quantiles (i.e. *z* values) is the necessary argument for the `pnorm` and `dnorm` functions. Since the quantile function is the inverse of the probability function, a probability or a vector of probabilities is the necessary argument for `qnorm`. Suppose we wanted to know the quantile below which 50% of the distribution lies.

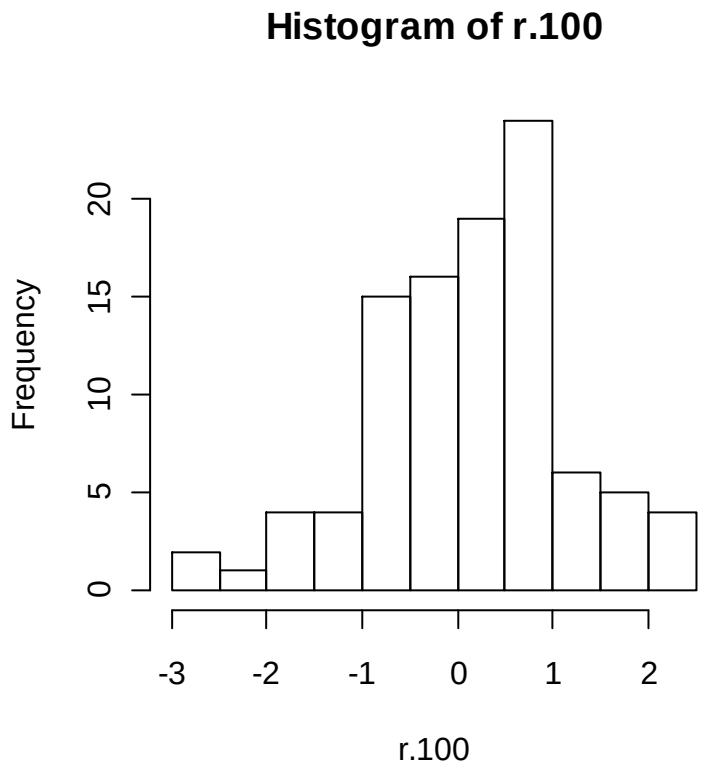
```
> qnorm(0.5)
[1] 0
```

Lastly, rnorm and analogous functions return (pseudo-)random samples.

```
> r.100<-rnorm(100)
```

Let's see what it looks like.

```
> hist(r.100)
```




## *18.2. Monte Carlo simulations*

*Robert & Casella 2010*

R is a great language for Monte Carlo simulations. Although it is possible to carry out sophisticated simulations, we will cover some very simple examples in this section. Say we are interested in determining if the river flow from two different rivers is different. This sounds like it could be an appropriate situation for using a *t* test. Using the River_flow.dat dataset, let's conduct a *t* test to see if the flow is different for two different rivers.

```
rivers.dat<-read.table("River_flow.txt",header=T)
```

```
> names(rivers.dat)
[1] "agency"         "site"           "date"           "discharge"
[5] "flag.discharge"
```

There are two different rivers in this dataset, so subset them so that we have 2 different dataframes. It has also been noticed that some of the observations are "NA", meaning that values were not reported.

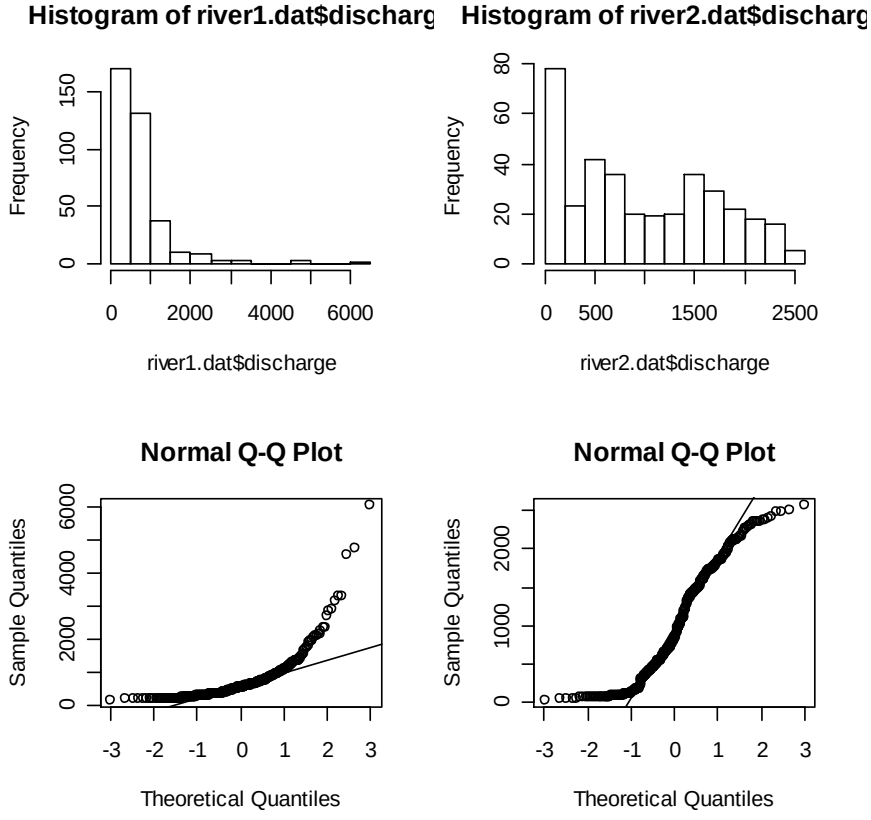
```
> river1.dat<-na.omit(subset(rivers.dat,site==1509000))
> river2.dat<-na.omit(subset(rivers.dat,site==4232730))
```

```
> t.test(river1.dat$discharge,river2.dat$discharge)
```

```
        Welch Two Sample t-test

data:  river1.dat$discharge and river2.dat$discharge
t = -5.4734, df = 721.814, p-value = 6.096e-08
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -386.0811 -182.2323
sample estimates:
mean of x mean of y
 701.0356  985.1923
```

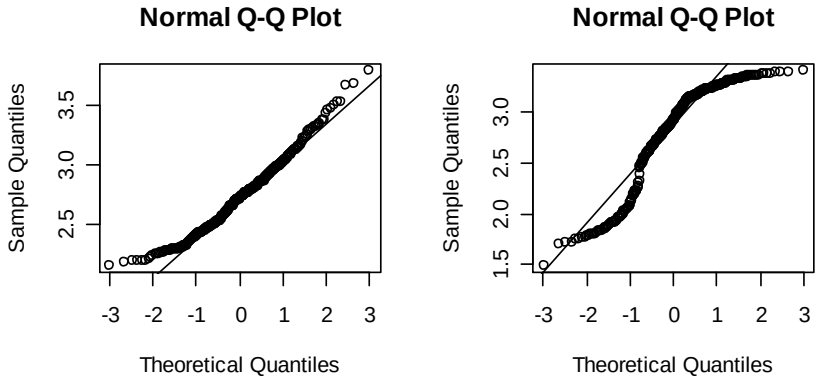
This result suggests that the mean flow from each river is in fact different. After conducting this t-test, you might wonder how appropriate this method is for this test, since you have assumed that the assumptions were not violated. In this case, the assumption that the data are normally distributed. Lets take a look at the distributions:

```
> par(mfrow=c(2,2))
> hist(river1.dat$discharge)
> hist(river2.dat$discharge)
> qqnorm(river1.dat$discharge)
> qqline(river1.dat$discharge)
> qqnorm(river2.dat$discharge)
> qqline(river2.dat$discharge)
```



The data do not appear to be normally distributed. Perhaps the non-parametric alternative, the Wilcoxon signed rank test, would have been more appropriate, or maybe a log transformation might have helped. It looks like it would help with `river1.dat`, but maybe not with `river2.dat`. Log transform the data and see what they look like:

```
> river1.dat$l.discharge=log10(river1.dat$discharge)
> river2.dat$l.discharge=log10(river2.dat$discharge)
> qqnorm(river1.dat$l.discharge)
> qqline(river1.dat$l.discharge)
> qqnorm(river2.dat$l.discharge)
> qqline(river2.dat$l.discharge)
```




This did appear to help with the river1.dat dataset, but not so much for the river2.dat dataset. Now we wonder how adversely the t-test method is affected by violation of assumptions. This will probably not answer the original question dealing with assessing the difference in the mean flows for each river, but it will be an interesting exercise, nonetheless.

We can answer this general question regarding violation of assumptions with a simple Monte Carlo simulation. The example that follows is a slightly modified version of an example described in Alberts (2007).

We want to determine the true significance level of a *t* test, given various population distributions and variances.

For this, the true significance level is calculated as the probability that the absolute value of the calculated *t* statistic is greater than or equal to the calculated critical t-value. To do this, we have to write some code to calculate the pooled standard deviation ($s_p$) and the t-statistic ($t$).

$$s_p = \sqrt{\frac{(m-1)*s_x^2 + (n-1)*s_y^2}{m+n-2}},$$

$$t = \frac{\bar{x} - \bar{y}}{s_p * \sqrt{\frac{1}{m} + \frac{1}{n}}},$$

where m and n are the sample sizes, $s_x^2$ and $s_y^2$ are the standard deviations, and $\bar{x}$ and $\bar{y}$ are the means. In R this can look like:

```
> sp<-sqrt(((m-1)*sd(x)^2+(n-1)*sd(y)^2)/(m+n-2))
> t<-(mean(x)-mean(y))/(sp*sqrt(1/m+1/n))
```

Normally, it would be beneficial to write this into a function, but that will not be covered until the next section, so for now we will specify this code for each simulation that we will conduct.

To set up the simulation, we will need to take random samples from the first population, and random samples from the second population. This can be easily accomplished with `rnorm`. Then we will need to compute the t-statistic from the two samples, and we will have to determine if the absolute value of the t-statistic is equal to or greater than the critical t-value. When this occurs, it represents a rejection of the null hypothesis that there is no difference in means.

For each simulation, we will keep track of the number of null hypothesis rejections, and we will use that to estimate the true significance level, which is calculated by the number of null hypothesis rejections divided by the total number of simulations. Let's set up the first problem:

```
a<-0.05
m<-20
n<-20
n.sim<-10000
n.reject<-0
for (i in 1:n.sim) {
      x<-rnorm(m,mean=10,sd=2)
      y<-rnorm(n,mean=10,sd=2)
      sp<-sqrt(((m-1)*sd(x)^2+(n-1)*sd(y)^2)/(m+n-2))
      t<-(mean(x)-mean(y))/(sp*sqrt(1/m+1/n))
      if (abs(t)>qt(1-a/2,n+m-2)) {
            n.reject<-n.reject+1
      }
}

> est.sig.level<-n.reject/n.sim
> est.sig.level
[1] 0.0489
```

From this simulation, we see that the estimated true significance level is 0.0489, which is very close to our specified alpha (`a`) of 0.05. The simulation above tested a situation in which the distributions were the same, but what happens if we specify different distributions? In the river flow example above, it looked like the first river had nearly log-normally distributed data, and the second river had a strange distribution of data. Lets see what happens if we assume that the

first set of samples comes from a log normal distribution, and the second set of samples comes from a normal distribution:

```
n.reject<-0
for (i in 1:n.sim) {
      x<-rlnorm(m,mean=log(10),sd=log(2))
      y<-rnorm(n,mean=10,sd=2)
      sp<-sqrt(((m-1)*sd(x)^2+(n-1)*sd(y)^2)/(m+n-2))
      t<-(mean(x)-mean(y))/(sp*sqrt(1/m+1/n))
      if (abs(t)>qt(1-a/2,n+m-2)) {
            n.reject=n.reject+1
      }
}
```

```
> est.sig.level<-n.reject/n.sim
> est.sig.level
[1] 0.1469
```

In this case, it appears that the estimated actual significance level is highly affected by the type of distribution. So, a violation of the normal distribution assumption appears to be cause for concern. Lets now consider the effect of unequal variance.

```
n.reject<-0
for (i in 1:n.sim) {
      x<-rnorm(m,mean=10,sd=3)
      y<-rnorm(n,mean=10,sd=1)
      sp<-sqrt(((m-1)*sd(x)^2+(n-1)*sd(y)^2)/(m+n-2))
      t<-(mean(x)-mean(y))/(sp*sqrt(1/m+1/n))
      if (abs(t)>qt(1-a/2,n+m-2)) {
            n.reject<-n.reject+1
      }
}
```

```
> est.sig.level<-n.reject/n.sim
> est.sig.level
[1] 0.056
```

It appears that the significance level is much less affected by violation of the equality of variance assumption. We can also use this simulation to look at the importance of sample size. Let's say that we were only able to collect 3 samples from each population:

```
a<-0.05
m<-3
n<-3
n.sim<-10000
n.reject<-0
for (i in 1:n.sim) {
      x<-rnorm(m,mean=10,sd=2)
      y<-rnorm(n,mean=10,sd=2)
      sp<-sqrt(((m-1)*sd(x)^2+(n-1)*sd(y)^2)/(m+n-2))
      t<-(mean(x)-mean(y))/(sp*sqrt(1/m+1/n))
      if (abs(t)>qt(1-a/2,n+m-2)) {
            n.reject<-n.reject+1
      }
}
```

```
> est.sig.level<-n.reject/n.sim
> est.sig.level
[1] 0.0542
```

This suggests that the true significance level is not largely affected by the sample size. However, this is not saying that the power of the tests is not affected. This is simply a testament to the central limit theorem. A different type of test can be constructed to look at the effect of sample size on the power of a *t* test. To evaluate the power of a *t* test, we can simply make some minor modifications to the code that we have been using. For this example, lets say that the actual difference in means is 1, and the standard deviation is 2. We will set the sample size at 30, and we will run the Monte Carlo simulation. Again, we will keep track of the number of rejections. With different means specified, the resulting calculation will provide us with an estimate of the probability that the test will correctly reject the null hypothesis when it is false (i.e. the power).

```
m<-30
a<-0.05
n.sim<-10000
n.reject<-0
for (i in 1:n.sim) {
      x<-rnorm(m,mean=8,sd=2)
      y<-rnorm(m,mean=9,sd=2)
      sp<-sqrt(((m-1)*sd(x)^2+(m-1)*sd(y)^2)/(m+m-2))
      t<-(mean(x)-mean(y))/(sp*sqrt(1/m+1/m))
      if (abs(t)>qt(1-a/2,m+m-2)) {
            n.reject<-n.reject+1
      }
}
est.power<-n.reject/n.sim
```

```
> est.power
[1] 0.4786
```

The power is not very high, so in this case, it might be advisable to increase the number of observations. We could write a little more code to iteratively determine the sample size that is required to achieve a power of 0.8, but R has built-in functions for that. The function `power.t.test` can be used to calculate power. Let's use that function for comparison with our Monte Carlo results. We have to specify the sample size, the true difference in means, and the standard deviation:

```
> power.t.test(n=30,delta=1,sd=2)

     Two-sample t test power calculation

              n = 30
          delta = 1
             sd = 2
      sig.level = 0.05
          power = 0.477841
    alternative = two.sided
```

```
 NOTE: n is number in *each* group
```

From this, it is apparent that the Monte Carlo results are very similar to the results of the explicit calculation of power.

## *18.3. Numerical simulations*

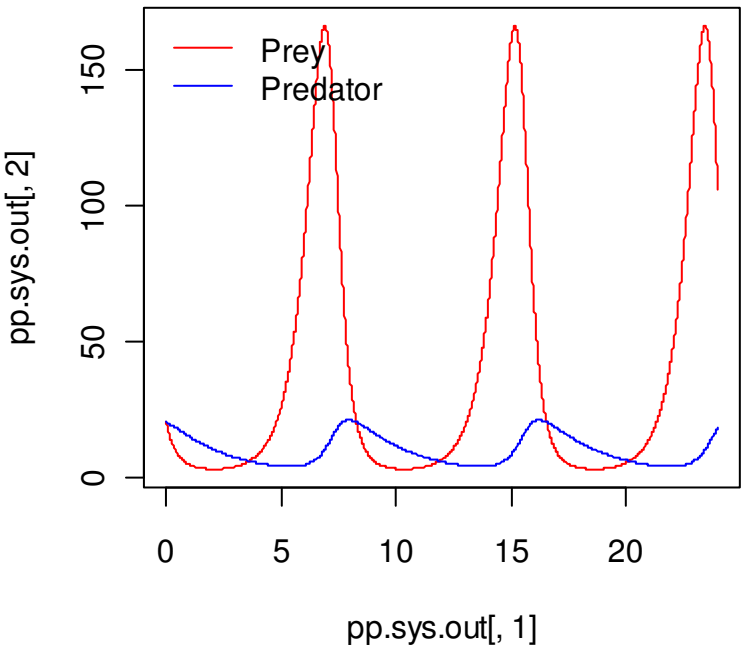
Although R may not be as capable as Matlab or Octave nor as flexible as Fortran (nor as capable) for simulation modeling, it is possible to carry out numerical modeling with R. Use of vectors, matrices, arrays, and lists, as well as vectorized operations, can make for very compact and efficient code. The package `deSolve` contains some powerful ordinary differential equation (ODE) solvers.

```
> install.packages("deSolve")
> library(deSolve)
```

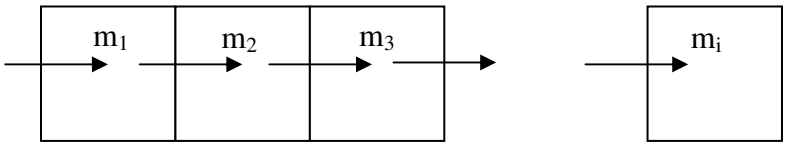
The ODE solvers available in deSolve require the initial state of a vector of state variables, plus a function that will calculate the state variable. A simple example is shown below.

Let's model population growth of a predator-prey system using the Lotka-Volterra equations.

```
pops.calc<-function(t,y,parms) {
   a<-parms$a
   b<-parms$b
   g<-parms$g
   d<-parms$d

   dprey.dt<-y[1]*(a - b*y[2])
   dpred.dt<- -y[2]*(g - d*y[1])
   return(list(c(dprey.dt,dpred.dt)))
}

a<-2
b<-0.2
g<-0.4
d<-0.01

prey<-20
pred<-20

times<-c(0,1:2400/100)

pp.sys.out<-lsoda(c(prey,pred),times,pops.calc,parms=list(a=a,b=b,g=g,d=d))

plot(pp.sys.out[,1],pp.sys.out[,2],type="l",col="red")
points(pp.sys.out[,1],pp.sys.out[,3],type="l",col="blue")
legend("topleft",c("Prey","Predator"),lty=1,col=c("red","blue"),bty="n")
```




Here is a slightly more complicated (and useful) example. Say we want to simulate the diffusion of $Na^+$ through groundwater.




The flux of $Na^+$ movement across each boundary is given by:

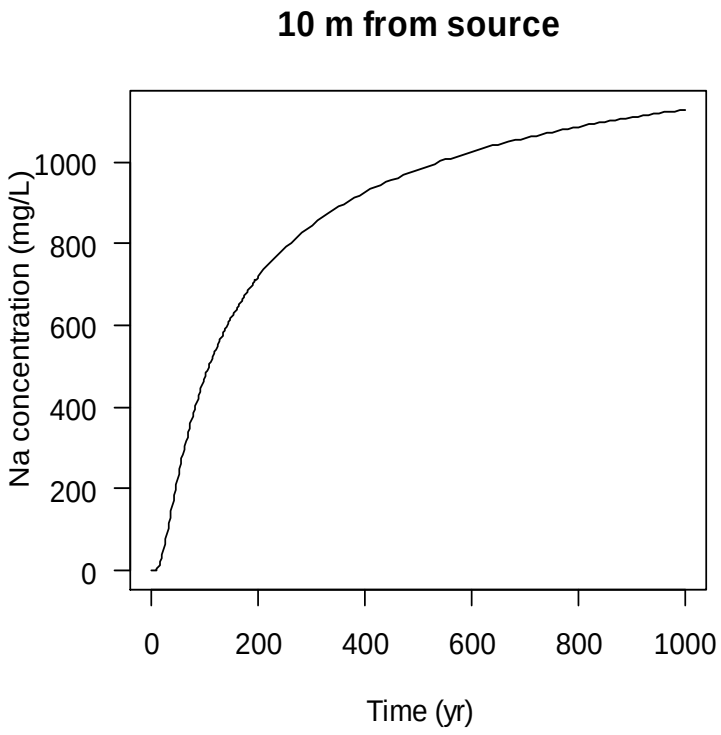
$$j = -D\frac{\Delta c}{\Delta x}$$

where $c$ = concentration (mass/length$^3$) and $x$ = position (length). Concentration is given by:

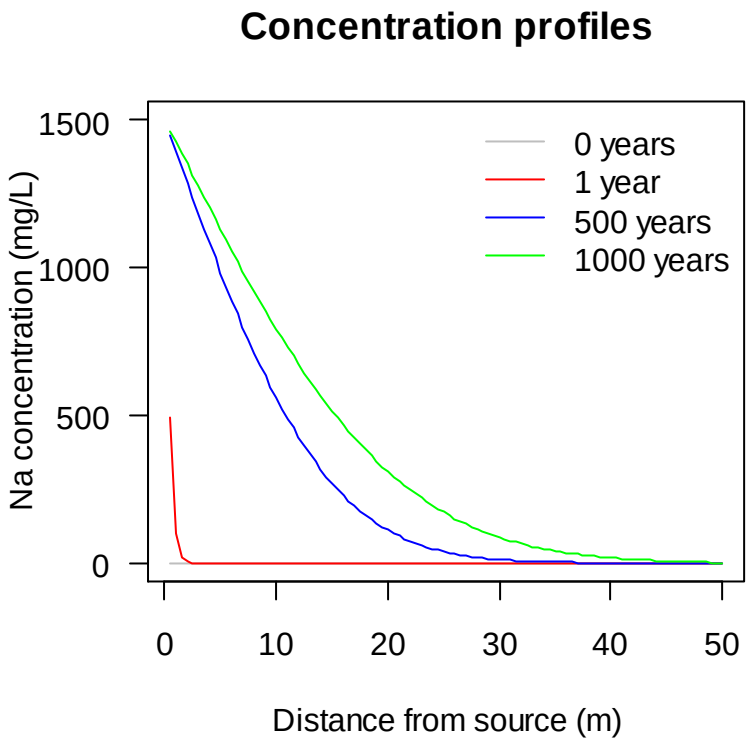
$$c = \frac{m}{width^3}$$

where $m$ = mass of $Na^+$ in our model cells (or layers) and $width$ = the width of our model cells. Lastly, let's assume that there is an infinite pool with 1500 mg/L $Na^+$ at the far left side of the system, and an infinite pool with no $Na^+$ at the far right.

```
# Set up function for calculating derivatives
diff.calc<-function (t,y,parms) {
   D<-parms$D
   w<-parms$w
   por<-parms$por
   c<-y
   flux<- -D*diff(c(1500,c,0))/w
   dc.dt<- -diff(flux)/(w*por)
   return(list(dc.dt=dc.dt))
}

# Set model parameters
D<-0.5 # m2/yr
times<-c(0,1:3,2*2:99,10*20:100)
w<-1.0
por<-0.5
dimens<-100

# Set initial concentrations
c<-rep(0,dimens)

# Now solve system
na.diff.out<-ode.band(c,times,diff.calc,nspec=1,parms=list(D=D,w=w,por=por))

plot(na.diff.out[,1],na.diff.out[,11],type='l',xlab="Time (yr)",ylab="Na
concentration (mg/L)",las=1,main="10 m from source")
```



```
plot((w-0.5)*(1:dimens),na.diff.out[1,-1],type='l',ylim=c(0,1500),
xlab="Distance from source (m)",ylab="Na concentration
(mg/L)",las=1,col="gray",main="Concentration profiles")
lines((w-0.5)*(1:dimens),na.diff.out[2,-1],col="red")
lines((w-0.5)*(1:dimens),na.diff.out[57,-1],col="blue")
lines((w-0.5)*(1:dimens),na.diff.out[62,-1],col="green")
legend("topright",c("0 years","1 year","500 years","1000
years"),lty=1,col=c("gray","red","blue","green"),bty="n")
```




Of course, one could come up with an analytical expression for this (relatively) simple scenario, but the approach used in this example can be used to implement much more complicated models.

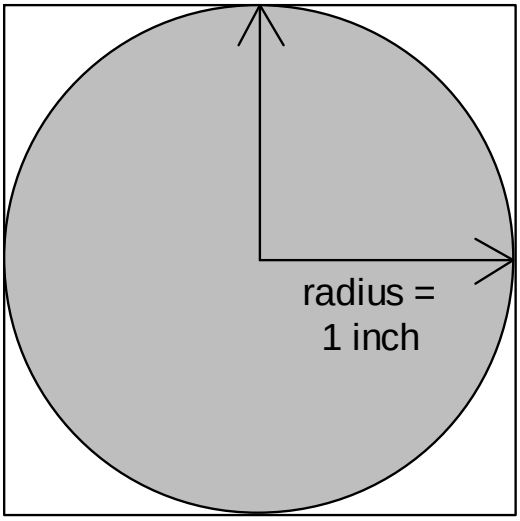
## *Exercises*

1. Use Monte Carlo simulation to estimate the 95% confidence limits for population size (N) estimates for 5 and 10 years, when the intrinsic rate of natural increase (r) is 0.3 $yr^{-1}$ (with standard deviation of 0.1 $yr^{-1}$) and the initial population size ($N_o$) is 200. The `quantile` function will be useful for summarizing the results of the simulation. For this example, assume exponential growth (i.e. $N=N_oe^{rt}$, where t = time in years). If you are really ambitious, create a simulation that estimates the confidence limits for 1 through 20 years, create a summary table, and summarize the results with a plot.

2. Use Monte Carlo to estimate the constant π. This can be tricky to set-up, and will probably be easiest if you deal with only 1 quadrant of the figure below. For this, you might want to think of throwing darts at a dartboard and quantify the number of hits within the circle vs. within the

square. The number of hits within each area (i.e. circle or square) is proportional to the area of that portion. Therefore, if you are only dealing with ¼ of the figure:

$$\frac{number\ of\ darts\ in\ circle}{number\ of\ darts\ in\ square} = \frac{\frac{1}{4}\pi r^2}{r^2} = \frac{1}{4}\pi$$




Hint: you will have to use the Pythagorean theorem to determine if your "dart" lands within the circle. Also, think about the type of distribution you will need to use for your random sampling!

Create a figure that summarizes your analysis. Show points in red if they are within your circle, and points in blue if they are outside the circle.

3. Write a simple program that simulates exponentional population growth using the ODE solver `lsoda`.

4. Bored? Here is a challenge. Using Newton's laws of gravity and motion, develop a simulation model of the Earth's orbit around the Sun using `lsoda`. Relevant laws are:

Newton's law of gravitation:

$$F = G\frac{m_1 m_2}{r^2}$$

where $F$ = force (N), $G$ = gravitational constant ($6.67428 \times 10^{-11}$ $m^3$ $kg^{-1}$ $s^{-1}$), $m$ = mass (kg), and $r$ = distance between objects (m).

Newton's second law of motion:

$$F = ma$$

where $a$ = acceleration (m $s^{-2}$).

Constants:
Earth's mass: $5.9736 \times 10^{24}$ kg
Sun's mass: $1.9891 \times 10^{30}$ kg
Perihelion (shortest distance between Earth and Sun): $1.4709 \times 10^{11}$ m (147 million km)
Maximum orbital velocity: 30.287 km/s
And, for checking your results:
Aphelion (greatest distance between Earth and Sun): $1.521 \times 10^{11}$ m (152 million km)
Minimum orbital velocity: 29.29 km/s

# 19. Batch processing

*R-Intro: Appendix B*

## *19.1. Running R in batch mode*

For most uses of R, it is perfectly efficient to write and save script files, and call them up in the R GUI with the `source` function, or even paste code directly into the R GUI. Of course, simple analyses can be typed directly into the R GUI. However, for automating data analysis, it is possible to execute an R script without opening the R GUI, using the Windows Command Prompt, Bash shell in Cygwin, or other shells. As listed in the `BATCH` help file, the command for running an R script in batch mode is:

```
R CMD BATCH [options] infile [outfile]
```

One useful option is `--no-save`, which will prevent R from saving your workspace to a file named .RData.

If you ever get the following error:

```
'R' is not recognized as an internal or external command, operable
program or batch file.
```

it means that Windows cannot find the R software. In this case, you need to manually add the directory that contains the R executable to your computer's list of directories that it should look in for executables, i.e. the Path variable. With Windows XP, this is Control Panel → Performance and Maintenance → System → Advanced tab → Environment Variables → find and highlight Path in the list → Click Edit, and then add `;C:\Program Files\R\R-2.8.1\bin` (or whatever path is correct) to the list. For more information or for other operating systems, search online.

As an example, let's create a script file with the following code:

```
# Set up function for calculating derivatives
diff.calc<-function (t,y,parms) {
   D<-parms$D
   w<-parms$w
   por<-parms$por
   c<-y
   flux<- -D*diff(c(1500,c,0))/w
   dc.dt<- -diff(flux)/(w*por)
   return(list(dc.dt=dc.dt))
}

# Set model parameters
D<-0.5 # m2/yr
times<-c(0,1:3,2*2:99,10*20:100)
w<-1.0
por<-0.5
dimens<-100
```

```
# Set initial concentrations
c<-rep(0,dimens)

# Now solve system
na.diff.out<-ode.band(c,times,diff.calc,nspec=1,parms=list(D=D,w=w,por=por))

# Open a pdf for exporting plots
pdf("Na_diffusion_sim.pdf",width=8,height=11)
   par(mfrow=c(2,1),oma=c(2,5,2,5))
   plot(na.diff.out[,1],na.diff.out[,11],type='l',xlab="Time (yr)",ylab=
       "Na concentration (mg/L)",las=1,main="10 m from source")

   plot((w-0.5)*(1:dimens),na.diff.out[1,-1],type='l',ylim=c(0,1500),
        xlab="Distance from source (m)",
        ylab="Na concentration (mg/L)",las=1,col="gray",
        main="Concentration profiles")
   lines((w-0.5)*(1:dimens),na.diff.out[2,-1],col="red")
   lines((w-0.5)*(1:dimens),na.diff.out[57,-1],col="blue")
   lines((w-0.5)*(1:dimens),na.diff.out[62,-1],col="green")
   legend("topright",c("0 years","1 year","500 years",
         "1000 years"),lty=1,col=c("gray","red","blue","green"),bty="n")
dev.off()
```

If we save this file as Na_diff_sim.R, we can call it up with the following command in the Windows Command Prompt.

```
> R CMD BATCH Na_diff_sim.R
```

The simulation runs, and the pdf is created.

Once you are familiar with running R via batch mode, it will be trivial to integrate R scripts with other software. For example, you may want to make some type of predictions using an external numerical model, and export the results for plotting in R. To do this, simply write a batch file that first calls the numerical model, and then calls an R script that plots the results.

# 20. Specialized packages, related documents, and additional information

User contributions to the CRAN website have made R very capable for many specialized analyses. Before writing a new function or developing a code-intensive analysis, it is a good idea to search CRAN to see if someone has already solved the problem for you.

If you typically carry out a certain type of analyses, say econometrics or "environmetrics", there are collections of useful packages on the CRAN website called task views. You can find a list of all the task views at http://cran.r-project.org/web/views/. For econometrics, for example, you should look at the econometrics task view. To install all the packages in a task view, you first need to have the `ctv` package installed.

```
> install.packages("ctv")
trying URL
'http://lib.stat.cmu.edu/R/CRAN/bin/windows/contrib/2.8/ctv_0.5-1.zip'
Content type 'application/zip' length 222936 bytes (217 Kb)
opened URL
downloaded 217 Kb

package 'ctv' successfully unpacked and MD5 sums checked

The downloaded packages are in
        C:\Documents and Settings\Sasha\Local
Settings\Temp\RtmpkEfLER\downloaded_packages
updating HTML package descriptions

> library(ctv)
```

Then, to load a task view, say envirometrics for analysis of environmental and ecological data:

```
> install.views("Environmetrics")
```

will install the few dozen or so packages included in the task view.

There are also many books on R available—check out the list on CRAN: http://www.r-project.org/doc/bib/R-publications.html. In addition to general texts, e.g. Dalgaard (2008) and Crawley (2008), there are books dedicated to specific types of analyses, such as Introductory Time Series with R (Cowpertwait & Metcalfe 2009) and Generalized Additive Models: An Introduction with R (Wood 2006). Many of these authors have posted R code and data sets online.

Lastly, there are several free documents on R on CRAN: http://cran.r-project.org/other-docs.html, including documents in multiple languages. After checking out Venables et al. (2008), you might want to look at these documents next.

# References

*Note: the R-something documents (R-Intro, R-Data, R-Lang) can be downloaded from CRAN (http://cran.r-project.org/manuals.html).*


Albert, J. 2007. *Bayesian Computation with R*. New York: Springer.

Cowpertwait, P., Metcalfe, A. 2009. *Introductory Time Series with R*. New York: Springer.

Crawley, Michael J. 2007. *The R Book*. Chichester, England: Wiley.

Dalgaard, Peter. 2008. *Introductory Statistics with R*. 2nd ed. New York: Springer.

Faraway, J. 2002. Practical Regression and ANOVA using R. Available at: http://cran.r-project.org/other-docs.html.

Faraway, J. 2005a. *Linear Models with R*. New York: Chapman and Hall/CRC.

Faraway, J. 2005b. *Extending the Linear Model with R: Generalized Linear, Mixed Effects and Nonparametric Regression Models*. New York: Chapman and Hall/CRC.

Lee, L., Helsel, D. 2005. Statistical analysis of environmental data containing multiple detection limits: S-language software for regression on order statistics. *Computers in Geoscience* 31: 1241-1248.

MeasuringWorth 2010. What Was the U.S. GDP Then? Annual Observations in Table and Graphical Format for years 1790 to the Present. http://www.measuringworth.org/usgdp/

Murrell, P. 2005. R Graphics. London: CRC Press.

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.

R Development Core Team. 2008. R Data Import/Export. (R-Data)

R Development Core Team. 2008. R Language Definition. (R-Lang)

Ritz, C., Streibig, J. 2009. *Nonlinear Regression with R*. New York: Springer.

Robert, C., Castella, G. 2010. *Introducing Monte Carlo Methods with R*. New York: Springer.

Spector, Phil. 2008. *Data Manipulation with R*. New York: Springer.

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.

Venables, W. N., D. M. Smith, and the R Development Core Team. 2008. An Introduction to R. (R-Intro)

Wilcock, R. J., C. D. Stevenson, and C. A. Roberts. 1981. An Interlaboratory Study of Dissolved Oxygen in Water. *Water Research* 15: 321-325.

Wood, S. N. 2006. *Generalized Additive Models: An Introduction with R*. Boca Raton, FL: Chapman & Hall/CRC.

Zar, Jerrold H. 1999. *Biostatistical Analysis*. 4th ed. Upper Saddle River, NJ: Prentice Hall.

# Appendix 1. Solutions to exercises

## *Section 1. Introduction to R*

```
#1.
128*2
256+12
268/2
134-128

#2. A vector. A data frame.

#3.

#4.
my.name<-"Sasha Hafner"
my.name-10
class(my.name)
?class

#5.
??"generalized additive"
```

## *Section 2. Vectors, matrices, and arrays*

```
# 1.
x<-1:10
log10(x)


#2.
x<-seq(0,2*pi,length.out=100)
y<-sin(2*x - 0.5)
min(y)
max(y)

#3.
v1<-rnorm(10)
v2<-rnorm(10)
v3<-rnorm(10)
v4<-rnorm(10)
v5<-rnorm(10)
s<-v1+v2+v3+v4+v5
s
mean(s)


#4.
matrix(1:25,nrow=5,byrow=T)

#5.
# This can be set up using matrix algebra:

# C%*%X = R
```

```
# Set up a coefficient matrix:
C<-matrix(c(27.2,32,-10.8,1,-1.48,0,409.1,0,13.5),nrow=3,byrow=T)

# A response matrix
R<-matrix(c(401.2,0,2.83),nrow=3)

# Now solve for X
X<-solve(C)%*%R
X

# or
X<-solve(C,R)
X

# Check solution
C%*%X
```

## *Section 3. Data frames, data import, and data export*

```
# 1.
read.table("Thakali_Ni_EC50s.txt",header=T)
ni.dat<-read.table("Thakali_Ni_EC50s.txt",header=T)
ni.dat
ni.dat<-read.table("Thakali_Ni_EC50s.txt",header=T,sep="\t")
wheat.dat<-read.table("Wheat.txt",header=T)
wheat.dat<-read.table("Wheat.txt",header=T,sep="\t")

# 2.
names(ni.dat)
min(ni.dat$ph.soil)
max(ni.dat$ph.soil)
ni.dat$l.ec50.ni<-log10(ni.dat$ec50.ni)
ni.dat

# 3.
ni.m.dat<-
data.frame(ec50=mean(ni.dat$ec50.ni),ph=mean(ni.dat$ph.soil),oc=mean(n
i.dat$oc))
write.table(ni.m.dat,"Ni_EC50_means1.out")
write.table(ni.m.dat,"Ni_EC50_means3.out",row.names=F,sep="\t")


# 4. There are a few ways to produce a file that you can read into R: select,
copy, and then paste the data into a text file (will produce a tab-delimited
file); use the "Save as" option in Excel to save the data as a tab delimited
text file (*.txt), as a comma-delimited file (*.csv), or as a space-delimited
file (*.prn) (except for this last option, you would have to add quotes
around the site names).

# This doesn't work because there are spaces in the site names:
beetles.dat<-read.table("Carion_beetles.txt",header=TRUE)

# This does:
```

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

# Or, if you used csv:
beetles.dat<-read.table("Carion_beetles.csv",header=TRUE,sep=",")
```

## *Section 4. Graphics, part I*

```
#1.
x<-seq(-2*pi,2*pi,0.05)
y<-cos(x)
dat<-data.frame(x=x,y=y)
plot(dat$x,dat$y,type="l")
plot(dat$x,dat$y,type="l",lty=4,col="darkgray")
plot(dat$x,dat$y,type="l",lty="52",col="darkgray")

#2.
ochem.dat<-read.table("Oxychem.txt",header=T)
names(ochem.dat)

plot(ochem.dat$dist,ochem.dat$dechlor,pch=21,bg="blue",xlab="Distance
from OxyChem (km)",ylab="Conc. in tree bark (ng/kg)",main="Dechlorane Plus
contamination")

plot(ochem.dat$dist,ochem.dat$dechlor,pch=21,bg="blue",log="xy",xlab="Distanc
e from OxyChem (km)", ylab="Conc. in tree bark (ng/kg)",main="Dechlorane Plus
contamination")

plot(ochem.dat$dist,ochem.dat$dechlor,pch=2,col="red",log="xy",xlab="Distance
from OxyChem (km)", ylab="Conc. in tree bark (ng/kg)",main="Dechlorane Plus
contamination")

# For a better log axis (a bit more advanced)
source('Functions.R') # This is the file of functions that we provided
plot(ochem.dat$dist,ochem.dat$dechlor,axes=FALSE,pch=2,col="red",log="xy",
xlab="Distance from OxyChem (km)", ylab="Conc. in tree bark
(ng/kg)",main="Dechlorane Plus contamination")
logaxis(1,1,1000)
logaxis(2,0.01,100)
```

## *Section 5. Manipulating data, part I*

```
# 1.
gdp.dat<-read.table("US_GDP.txt",header=T)
gdp.dat[1:10,c("year","gdp.real")]
gdp.dat[gdp.dat$year>1799 & gdp.dat$year<1900,"gdp.nom"]
gdp1800s.dat<-subset(gdp.dat,year>=1800 & year<1900)

# 2.
match(max(gdp.dat$gdp.real),gdp.dat$gdp.real)
gdp.dat[order(gdp.dat$gdp.real),]

# 3.
cvtort.dat<-read.table("Cacti_v_tort.txt",header=T)
names(cvtort.dat)
cvtort.2.dat<-subset(cvtort.dat,tortoises=="Yes")
```

## *Section 6. Manipulating data, part II*

```
#1.
react.dat<-read.table("Reactors.txt",header=T)
h2.dat<-read.table("Biohydrogen.txt",header=T)
all.dat<-merge(react.dat,h2.dat)
all.dat

# 2.
# There are two steps involved in conversion to a date-time object
h2.dat$date.time<-as.POSIXct(paste(h2.dat$date,h2.dat$time),format='%m/%d/%Y
%H:%M')

# There are at least two ways to calculate the elapsed time
h2.dat$time.e<-h2.dat$date.time - as.POSIXct('2006-09-18 11:12:00')
h2.dat$time.e<-difftime(h2.dat$date.time,'2006-09-18 11:12:00',units='hours')

# And, to get rid of the units label
h2.dat$time.e<-as.numeric(h2.dat$time.e)

# Here is an alternate approach using ave
h2.dat$date.time<-as.numeric(h2.dat$date.time)
h2.dat$time.e<-
ave(as.numeric(h2.dat$date.time),h2.dat$reactor,FUN=function(x) x-x[1])

# And another approach, which does essentially what ave does:
h2.dat$time.e<-
unsplit(lapply(split(h2.dat$date.time,h2.dat$reactor),function(x) x-
x[1]),f=h2.dat$reactor)

# 3.
eagles.dat<-read.table("Eagles.txt",header=T)
summary(eagles.dat)

# Gives you the data you want, but not good for data frames
by(eagles.dat$achlor,eagles.dat$site,mean)
by(eagles.dat$achlor,eagles.dat$site,sd)
by(eagles.dat$achlor,eagles.dat$site,length)

# This will give you a data frame
eagles.summ.dat<-data.frame(mean=tapply(eagles.dat$achlor,
eagles.dat$site,mean),SD=tapply(eagles.dat$achlor,eagles.dat$site,sd),
n=tapply(eagles.dat$achlor,eagles.dat$site,length))

write.table(eagles.summ.dat,"Eagle_summary.out")

# 4.
now<-Sys.time()
bd<-as.POSIXlt("1978-01-02 23:12")
age<- now - bd
age
age<-difftime(now,bd,units="hours")
age
```

## *Section 7. Exploratory data analysis*

```
# 1.
install.packages("ISwR")
library(ISwR)
insects.dat<-InsectSprays
summary(insects.dat)
tapply(insects.dat$count,insects.dat$spray,summary)
# or
by(insects.dat$count,insects.dat$spray,summary)
# or
aggregate(insects.dat$count,list(spray=insects.dat$spray),summary)
boxplot(count~spray,data=insects.dat,xlab="Spray",ylab="Count",las=1)

# 2.
cu.dat<-read.table("StreamCu.txt",header=T)
cu.dat
# Assume log-normal dist
cu.ros<-ros(cu.dat$cu,cu.dat$nondetect)
cu.ros
plot(cu.ros)
# If we want the geomean
cu.est<-data.frame(cu.ros)
cu.est$l.cu.mod<-log10(cu.est$modeled)
cu.est
mean(cu.est$l.cu.mod)
sd(cu.est$l.cu.mod)
10^mean(cu.est$l.cu.mod)
# Another option for log statistics
cu.ros<-ros(log10(cu.dat$cu),cu.dat$nondetect,forwardT=NULL)
cu.ros

# 3.
summary(IgM)
IgM
qqnorm(IgM)
qqline(IgM)
# Try log-transformed
qqnorm(log10(IgM))
qqline(log10(IgM))

# 4.
# Find range in ppoints
range(ppoints(IgM))
plot(qnorm(ppoints(IgM)),sort(IgM),log="y",xaxt="n")
axis(1,qnorm(c(0.001,0.01,0.1,0.1,0.5,0.9,0.99,0.999)),c(0.001,0.01,0.1,0.1,0
.5,0.9,0.99,0.999))
```

## *Section 8. One- and two-sample tests*

```
#1.
?sleep
summary(sleep)
t.test(extra~group,data=sleep)

#2.
sleep.tt<-t.test(extra~group,data=sleep)
```

```
names(sleep.tt)
attributes(sleep.tt) # Gives two rows
sleep.tt.summ.dat<-
data.frame(tstat=sleep.tt$parameter,Pval=sleep.tt$p.value,CI=sleep.tt$conf.in
t)
sleep.tt.summ.dat<-
data.frame(tstat=sleep.tt$parameter,Pval=sleep.tt$p.value,LCL=sleep.tt$conf.i
nt[1],UCL=sleep.tt$conf.int[2])
# A bit shorter
sleep.tt.summ.dat<-
with(sleep.tt,data.frame(tstat=parameter,Pval=p.value,LCL=conf.int[1],UCL=con
f.int[2]))
write.table(sleep.tt.summ.dat,"Sleep_ttest.out")
```

## *Section 9. Classical linear models*

```
#1.
mammalsleep
summary(mammalsleep)
# Note that there are some missing values. We can leave them in, and lm will
# skip any observations that contain them, but let's delete them from the
# start
# Let's focus on the variables we are going to use
s.dat<-na.omit(mammalsleep[,-(3:4)])
pairs(s.dat)
cor(s.dat)
# Danger indices are highly correlated, so I am going to pick one
mod.1<-lm(sleep ~ body + brain + lifespan + gestation + predation,data=s.dat)
summary(mod.1)
summary(mod.2<-update(mod.1, ~. - body))
summary(mod.3<-update(mod.2, ~. - lifespan))
summary(mod.4<-update(mod.3, ~. - brain))
plot(mod.4)
# Elephants have a lot of leverage (rule-of-thumb is 2p/n). Try dropping.
summary(mod.5<-update(mod.4,subset=body<2000))
# Or, for clarity
s.dat<-subset(s.dat,body<2000)
mod.5<-lm(sleep ~ gestation + predation,data=s.dat)
summary(mod.5)
plot(mod.5)
prplot(mod.5,1)
prplot(mod.5,2)
# Plot results
preds<-predict(mod.5,int='c')
preds<-preds[order(preds[,1]),]
plot(predict(mod.5),s.dat$sleep)
matlines(preds[,1],preds,lty=c(1,2,2),col=c('black','red','red'))
preds2<-predict(mod.5,int='p')
preds2<-preds2[order(preds2[,1]),]
matlines(preds2[,1],preds2[,-1],lty=3,col='blue'))

#2.
install.packages('MASS')
library(MASS)
cab.dat<-cabbages
summary(cab.dat)
# To plot the data
```

```
interaction.plot(Date,Cult,HeadWt,data=cab.dat)
# Or, something like this
plot(HeadWt ~ as.numeric(Date),data=cab.dat,col=as.numeric(cab.dat$Cult),
pch=as.numeric(cab.dat$Cult))
mod<-aov(HeadWt ~ (Cult + Date)^2, data = cab.dat)
summary(mod)
model.tables(mod,type="mean")
plot(mod)

#3.
library(faraway)
ff.dat<-fruitfly
summary(ff.dat)
plot(longevity ~ thorax, pch=as.numeric(ff.dat$activity),col=ff.dat$activity,
data=ff.dat)
legend('topleft',levels(ff.dat$activity),pch=1:5,col=1:5)
mod1<-lm(longevity ~ (thorax + activity)^2, data = ff.dat)
anova(mod1)
mod2<-update(mod1, ~. - thorax:activity)
anova(mod2)
summary(mod2)
plot(mod2)
# Try with a log transformation to eliminate heteroscedasticity
mod3<- lm(log10(longevity) ~ thorax + activity, data = ff.dat)
anova(mod3)
summary(mod3)
plot(mod3)
plot(longevity ~ thorax, pch=as.numeric(ff.dat$activity),col=ff.dat$activity,
data=ff.dat)
legend('topleft',levels(ff.dat$activity),pch=1:5,col=1:5)
# Code for adding predictions is a bit more advanced (there are many
# different ways to do this
preds<-10^predict(mod3)
for(i in 1:5) {
   lines(ff.dat$thorax[as.numeric(ff.dat$activity)==i],
   preds[as.numeric(ff.dat$activity)==i],col=i)
}
# An alternate approach
preds<-10^predict(mod3,newdata=data.frame(thorax=rep(thorax<-
seq(0.6,0.95,length.out=20),5),
activity=rep(levels(ff.dat$activity),each=20)))
matlines(thorax,matrix(preds,nrow=20),col=1:5,lty=1)

#4.
cact.dat<-read.table("Cactus_width.txt",header=T)
summary(cact.dat)
cact.dat$hw.ratio<-cact.dat$height/cact.dat$width
cact.dat$tortoise<-factor(cact.dat$tortoise)
mod.1<-lm(hw.ratio~tortoise + understory,data=cact.dat)
summary(mod.1)
anova(mod.1)

# Some plotting options
coplot(hw.ratio ~ understory|tortoise, data = cact.dat)
plot(cact.dat$understory,cact.dat$hw.ratio,type="n")
points(cact.dat$understory[cact.dat$tortoise==1],cact.dat$hw.ratio[cact.dat$t
ortoise==1],col="red")
```

```
points(cact.dat$understory[cact.dat$tortoise==0],cact.dat$hw.ratio[cact.dat$t
ortoise==0],col="blue")
```

## *Section 10. Nonparametric alternatives to* t *tests and ANOVA*

```
# 1.
sleep.dat<-sleep
split(sleep.dat,sleep.dat$group)
by(sleep.dat$extra,sleep.dat$group,mean)
boxplot(extra~group,data=sleep.dat)
wilcox.test(extra~group,data=sleep.dat)
```

## *Section 11. Groups, looping, and conditional execution*

```
#1.
n<-1000000
x<-1:n
system.time( for (i in 1:n) { sqrt(x[i]) } )
system.time(sqrt(x))

#2.
eagles.dat<-read.table("Eagles.txt",header=TRUE)
# See what we have
summary(eagles.dat)
# Note that there are other ways to do this. . .
eagles.lst<-split(eagles.dat,eagles.dat$site)
for(i in levels(eagles.dat$site)) {
   write.table(summary(eagles.lst[[i]]),paste(i,"out",sep="."))
}

#3.
x<-rnorm(100)
a<-ifelse(x>0,"H","L")
```

## *Section 12. Graphics II*

```
# 1.
par(mfrow=c(1,2))
curve(sin,-10,10,col="red",las=1,xlab="x",ylab="Sine(x)")
curve(cos,-10,10,col="blue",lty=2,las=1,xlab="x",ylab="Cosine(x)")

# Now for both series on one plot
# First method
curve(sin,-10,10,col="red",las=1,xlab="x",ylab="Sine(x) or Cosine(x)")
curve(cos,-10,10,col="blue",lty=2,las=1,add=TRUE)

# Second method
curve(sin,-10,10,col="red",las=1,xlab="x",ylab="Sine(x) or Cosine(x)")
par(new=TRUE)
curve(cos,-10,10,col="blue",lty=2,las=1,axes=F,xlab="",ylab="")

# Third method
x<-seq(-10,10,0.1)
y<-sin(x)
z<-cos(x)
matplot(x,cbind(y,z),col=c("red","blue"),las=1,xlab="x",ylab="Sine(x) or
Cosine(x)",lty=1:2,type="l")
```

```
# 2.
ll.dat<-Loblolly
ll.dat$height<-ll.dat$height*0.3048
summary(ll.dat)
plot(height~age,type="n",xlab="Age (yr)",ylab="Height
(m)",las=1,xlim=c(0,25),data=ll.dat)
points(ll.dat$age[ll.dat$Seed==307],ll.dat$height[ll.dat$Seed==329],pch=1,col
="red",type="o")
points(ll.dat$age[ll.dat$Seed==311],ll.dat$height[ll.dat$Seed==307],pch=6,col
="blue",type="o")
points(ll.dat$age[ll.dat$Seed==311],ll.dat$height[ll.dat$Seed==311],pch=22,co
l="black",type="o")
legend("topleft",legend=c(311,307,329),pch=c(1,6,22),col=c("red","blue","blac
k"))

# 3.
# Code for plots
for (i in names(wind.dat)[3:14]) {
   hist(wind.dat[,i],xlim=c(0,50),ylim=c(0,0.25),freq=FALSE,
   col="lightgray",xlab="Wind speed (m/s)",ylab="Density",
   main=paste("Wind speed,",i),breaks=10)
}

# jpeg files first
jpeg("Wind%02d.jpg",height=4,width=4,units="in",res=300)
   for (i in names(wind.dat)[3:14]) {
      hist(wind.dat[,i],xlim=c(0,50),ylim=c(0,0.25),freq=FALSE,
      col="lightgray",xlab="Wind speed (m/s)",ylab="Density",
      main=paste("Wind speed,",i),breaks=10)
   }
dev.off()

pdf("Wind.pdf",height=11,width=8.5)
   for (i in names(wind.dat)[3:14]) {
      hist(wind.dat[,i],xlim=c(0,50),ylim=c(0,0.25),freq=FALSE,
      col="lightgray",xlab="Wind speed (m/s)",ylab="Density",
      main=paste("Wind speed,",i),breaks=10)
   }
dev.off()

# 4.
hard.dat<-read.table("Janka.txt",header=T)
layout(matrix(c(1,1,2,3,3,4,3,3,4),ncol=3,byrow=T))
layout.show(4)

hist(hard.dat$density,xlab="Wood Density",col="grey",main="Histogram of Wood
Density")

plot(1,type="n",xlab="",ylab="",axes=F)

mtext(paste("Meanhardness\n=",signif(mean(hard.dat$hardness),3)),
side=3,line=0,cex=0.8)

mtext(paste("Mean density\n=",signif(mean(hard.dat$density),3)),side=3,line=-
3,cex=0.8)
```

```
plot(hard.dat$density,hard.dat$hardness,xlab="Wood density",pch=23,
bg="green", ylab="Wood Hardness",las=1)

abline(lm(hard.dat$hardness~hard.dat$density),lty=2,col="red")

legend("topleft",c("data series","linear model"), pch=c(23,-1),
pt.bg="green", lty=c(0,2), col=c("black","red"),bty="n")

mean.hard<-mean(hard.dat$hardness)
mean.dens<-mean(hard.dat$density)

lines(c(20,mean.dens),c(mean.hard,mean.hard),lty=5)
lines(c(mean.dens,mean.dens),c(240,mean.hard),lty=5)
text(40,2200,"Mean Hardness",pos=3)
arrows(40,2200,33,mean.hard,length=0.2,angle=20)
boxplot(hard.dat$hardness,las=1,col="blue")

dev.off()
```

## *Section 13. Functions*

```
# 1.
rmse<-function(obs,pred) {
   sqrt(sum((pred-obs)^2)/length(obs))
}

# Generate model predictions
x<-1:100
y<-2*x + rnorm(100,5)
y.pred<-predict(lm(y ~ x))
rmse(y,y.pred)
# Compare to
sd(y)


# 2.
wdens<-function(T,units='C') {
   if (sum(units==c('C','F','K'))==0) return(paste('Error, expect C, F, or
                                                   K for units, got',units))
   if (units=='K') T<-T - 273.15
   if (units=='F') T<-(T - 32)*5/9
   if (max(T)>100 | min(T)<0) return('Temperature outside 0-100 C range')
   0.9999 + 4.8916e-05*T - 7.4098E-06*T^2 + 3.9982E-08*T^3 - 1.2329E-10*T^4
}

# Test it
wdens(0:30,'K')
wdens(274:295,'K')


# 3.
ebars<-function(x,y,y.upper,y.lower,angle=90,length=0.02,...) {
   arrows(x,y,x,y.upper,angle=angle,length=length,...)
   arrows(x,y,x,y.lower,angle=angle,length=length,...)
}

# Test it, using InsectSprays
```

```
dat<-data.frame(mean=tapply(InsectSprays$count,list(InsectSprays$spray),
mean),sd= tapply(InsectSprays$count,list(InsectSprays$spray),sd))
plot(dat$mean)
ebars(1:6,dat$mean,dat$mean+dat$sd,dat$mean-dat$sd,col='red')
# Or
x<-barplot(dat$mean,ylim=c(0,25))
ebars(x,dat$mean,dat$mean+dat$sd,dat$mean-dat$sd,lwd=2,col='green')


# 4.
mmerge<-function(dframes,bys,...) {
   m.dat<-dframes[[1]]
   if (length(dframes)>2) {
      for (i in 2:length(dframes)) {
         m.dat<-merge(m.dat,dframes[[i]],
                      by.x=bys[[i-1]],by.y=bys[[i]],...)
      }
   }
   m.dat
}

# Test it
react.dat<-read.table("Reactors.txt",header=T)
h2.dat<-read.table("Biohydrogen.txt",header=T)
junk.dat<-data.frame(bottle=levels(react.dat$reactor),nothing=1:15)
junk.dat$nothing[5:8]<- -20

test.dat<-mmerge(list(react.dat,h2.dat,junk.dat),
list('reactor','reactor','bottle'))
test.dat
```

## *Section 14. Generalized linear models*

```
#1.
tox.dat<-read.table("Cu_tox_test.txt",header=T)
tox.dat$dead<-tox.dat$tot-tox.dat$alive
tox.dat$prop.dead<-tox.dat$dead/tox.dat$tot
tox.dat$l.cu<-log10(tox.dat$cu)
mod.1<-glm(prop.dead~cu,binomial,weights=tot,data=tox.dat)
mod.2<-glm(prop.dead~l.cu,binomial,weights=tot,data=tox.dat)
mod.3<-glm(prop.dead~l.cu,binomial(link="probit"),weights=tot,data=tox.dat)
summary(mod.1)
summary(mod.2)
summary(mod.3)
# Make predictions
tox.pred.dat<-data.frame(cu=x<-seq(0.001,60,1),l.cu=log10(x))
tox.pred.dat$pred.1<-predict(mod.1,newdata=tox.pred.dat,type="response")
tox.pred.dat$pred.2<-predict(mod.2,newdata=tox.pred.dat,type="response")
tox.pred.dat$pred.3<-predict(mod.3,newdata=tox.pred.dat,type="response")
# And plot them
plot(tox.dat$cu,tox.dat$prop.dead,pch=21,bg="green",xlab=expression("Cu
concentration"~~(mu*g/L)),ylab="Proportion dead",las=1)
lines(pred.1~cu, col="blue",data=tox.pred.dat)
lines(pred.2~cu, col="red",data=tox.pred.dat)
```

```
lines(pred.3~cu, col="green",data=tox.pred.dat)
legend("topleft",c("Logistic","Logisic
(log)","Probit"),lty=1,col=c("blue","red","green"),bty="n")

#2.
squirrel.dat<-read.table("Squirrel_color.txt",header=T)
summary(squirrel.dat)
squirrel.dat$black<-factor(squirrel.dat$black)
mod.1<-glm(black~dist2ctr,binomial,data=squirrel.dat)
summary(mod.1)
boxplot(squirrel.dat$dist2ctr~squirrel.dat$black)
# Try to visualize data
plot(squirrel.dat$dist2ctr,squirrel.dat$black) # Hard to see much
plot(black~dist2ctr,data=squirrel.dat) # Useful but what is it? See
?plot.factor
spineplot(black~dist2ctr,data=squirrel.dat,breaks=100,col=c('gray70','gray25'
))


#3.
esoph.dat<-esoph
summary(esoph.dat)
# Are alcohol and tobacco consumption groups ordered factors? They should be.
is.ordered(esoph.dat$alcgp)
is.ordered(esoph.dat$tobgp)
is.ordered(esoph.dat$agegp)
# All in one line of code
mod.1<-glm(cbind(ncases,ncontrols) ~ agegp + alcgp + tobgp, data = esoph.dat,
family=binomial)
summary(mod.1)
```

## *Section 15. Generalized additive models*

```
#1.
library(mgcv)
isolation.dat<-read.table("Isolation.txt",header=T)
summary(isolation.dat)
mod.1<-gam(incidence~s(area)+s(isolation),binomial,data=isolation.dat)
summary(mod.1)
plot(mod.1,resid=T,cex=3)
# One way to view data
plot(isolation.dat$area,isolation.dat$isolation,pch=isolation.dat$incidence,c
ol=isolation.dat$incidence+1)
legend('topright',c('Absent','Present'),pch=0:1,col=1:2)

#2.
cars.dat<-mtcars
summary(cars.dat)
mod.1<-gam(mpg~s(disp)+s(hp)+s(wt),data=cars.dat)
summary(mod.1)
plot(mod.1,resid=T,cex=3)
# One way to view these data
coplot(mpg ~ disp | factor(cyl), data=cars.dat, rows=1)
```

## *Section 16. Nonlinear regression*

```
#1.
```

```
dimethyl.dat<-read.table("Dimethyl-death.txt",header=T)
summary(dimethyl.dat)
plot(dimethyl.dat$t,log(dimethyl.dat$dmd.conc))
# Plot suggests a guess of around 4/100 = 0.04 per d for rate
mod.1<-nls(dmd.conc~c0*exp(-d*t),start=list(c0=100,d=0.04),data=dimethyl.dat)
summary(mod.1)
coef(mod.1)
pred.dat<-data.frame(t=1:110)
pred.dat$conc.pred<-predict(mod.1,newdata=pred.dat)
plot(dimethyl.dat$t,dimethyl.dat$dmd.conc,pch=24,bg="red",xlab="Time
(days)",ylab="DMD concentration (g/L)",las=1)
lines(pred.dat$t,pred.dat$conc.pred,col='red')
# Model with background concentration
mod.2<-nls(dmd.conc~c0*exp(-d*t) + bg,start=list(c0=100,d=0.04,bg=5),
data=dimethyl.dat)
summary(mod.2)
pred.dat$conc.pred.2<-predict(mod.2,newdata=pred.dat)
lines(pred.dat$t,pred.dat$conc.pred.2,col='blue')

#2.
summary(DNase)
dnase.dat<-subset(DNase,Run==5)
summary(dnase.dat)
plot(dnase.dat$conc,dnase.dat$density)
mod.1<-nls(density~SSlogis(conc,Asym,xmid,scal),data=dnase.dat)
summary(mod.1)
```

## *Section 18. Distributions and simulations*

```
# 1.
n.i<-2
n.years<-10
# Generate 1000 estimates of r to work with
r<-rnorm(1000,mean=0.3,sd=0.1)
# Let's go for 20 years
pred.dat<-data.frame(year=year<-0:20)
# Use outer to make calculations, and keep all estimates in a matrix
pop<-outer(year,r,FUN=function(yr,r) n.i*exp(r*yr))
# Calculate 5% and 95% percentile (note that we have to transpose a matrix)
pred.dat[,c('pop.lower','pop.med','pop.upper')]<-t(apply(pop,1,function(x)
quantile(x,c(0.05,0.5,0.95))))
plot(pred.dat$year,pred.dat$pop.upper,xlab='Time
(year)',ylab='Population',las=1,type='l',lty=2)
lines(pred.dat$year,pred.dat$pop.med,lty=1)
lines(pred.dat$year,pred.dat$pop.lower,lty=2)

#2.
# Recommended approach—no loops
# calculate pi with one call to runif, and use a data frame
n.samples<-10000
x<-runif(n.samples,0,1)
y<-runif(n.samples,0,1)
hyp<-sqrt(x^2+y^2)
pi.dat=data.frame(x=x,y=y,hyp=hyp)
n.in<-sum(pi.dat$hyp<1)
pi.est=4*n.in/n.samples
pi.est
```

```
plot(pi.dat$x,pi.dat$y,col=ifelse(pi.dat$hyp<1,'red','blue'),
xlab="x",ylab="y",pch=21,cex=0.5,asp=1)

# Calculation using a loop, without figure
n.sim<-10000
n.in.circle<-0
for (i in 1:n.sim) {
   x<-runif(1,0,1)
   y<-runif(1,0,1)
   hyp<-sqrt(x^2+y^2)
   if (hyp<=1) { n.in.circle<-n.in.circle+1 }
}
pi.est<-4*n.in.circle/n.sim
pi.est

# Calculated pi and draws a figure while working through the loop
# Looks cool
n.sim<-10000
n.in.circle<-0
plot(0,0,type="n",xlim=c(0,1),ylim=c(0,1),ylab="y",xlab="x",asp=1)
for (i in 1:n.sim) {
   x<-runif(1,0,1)
   y<-runif(1,0,1)
   hyp<-sqrt(x^2+y^2)
   if (hyp<=1) {
      n.in.circle<-n.in.circle+1
      points(x,y,pch=21,col="red",bg="red",cex=0.5)
   }
   if (hyp>1) {
      points(x,y,pch=21,col="blue",bg="blue",cex=0.5)
   }
}
pi.est<-4*n.in.circle/n.sim
pi.est
legend("topleft",paste("estimate of pi=",pi.est),bty="n")

#3.
install.packages("deSolve")
library(deSolve)

pop.calc<-function(t,y,parms) {
   dy.dt<-y*parms$r
   return(list(dy.dt))
}

t<-0:100
out<-lsoda(1,t,func=pop.calc,parms=list(r=0.1))
plot(t,out[,2],xlab='Time',ylab='Population')

#4.
# orb.calc calculates four derivatives: dx/dt, dz/dt, dv(x)/dt, and dv(y)/dt
orb.calc<-function(t,y,parms) {
   m.s<-parms$m.s
   m.e<-parms$m.e
   G<-parms$G

   x<-y[1]
```

```
   z<-y[2]
   v.x<-y[3]
   v.z<-y[4]

   # Calculate force (N) acting on Earth--Newton's law of gravitation
   d<-sqrt(x^2 + z^2)
   F.x<- -x/d*m.s*m.e*G/d^2
   F.z<- -z/d*m.s*m.e*G/d^2

   # Calculate acceleration (m/s2) based on Newton's second law of motion
   a.x<-F.x/m.e
   a.z<-F.z/m.e

   # Summarize
   dv.x.dt<-a.x # dv(x)/dt (m/s^2)
   dv.z.dt<-a.z # dv(y)/dt (m/s^2)
   dp.x.dt<-v.x # dx/dt (m/s)
   dp.z.dt<-v.z # dy/dt (m/s)

   # Collison with sun
   if(d<=6.96E8) dv.x.dt<-dv.z.dt<-dp.x.dt<-dp.z.dt<-0

   # Return output
   return(list(c(dp.x.dt,dp.z.dt,dv.x.dt,dv.z.dt)))
}

# orb.mod solves ODEs
orb.mod<-function(m.s=1.98892E30,m.e=5.9742E24,x0=0,z0=147098290000,v.x=-
30287,v.z=0,G=6.67428E-11,times=0:365*86400) {

   out<-data.frame(lsoda(c(x=x0,z=z0,v.x=v.x,v.z=v.z),times,orb.calc,
           parms=list(m.s=m.s,m.e=m.e,G=G)))
   out$d<-sqrt(out$x^2 + out$z^2)
   return(data.frame(out))
}

# Now write function to call up and plot results intermittently
orb.plot<-function(m.s=1.98892E30,m.e=5.9742E24,x0=0,z0=147098290000,
          v.x=-30287,v.z=0,G=6.67428E-11,time=365,step=5,t.pause=0.2,
          new.plot=T) {
   steps<-ceiling(max(time)/step)
   if(new.plot==T) {
      plot(x0/1E9,z0/1E9,xlim=c(-160,160),ylim=c(-160,160),xlab='x
                  location (million km)',ylab='y location (million
                   km)',asp=1,las=1)
      points(0,0,pch=21,cex=2,bg='yellow')
   }
   for (i in 0:steps) {
      out<-orb.mod(m.s=m.s,m.e=m.e,x0=x0,z0=z0,v.x=v.x,v.z=v.z,G=G,
                  times=86400*c(max(0,(i-1)*step),i*step))
      points(out$x[1]/1E9,out$z[1]/1E9,pch=21,col='white',bg='white')
      text(out$x[1]/1E9,out$z[1]/1E9,as.character(ceiling(i*step)),
                  col='gray',cex=0.5)
      points(out$x[2]/1E9,out$z[2]/1E9,pch=21,col='blue',bg='blue')
      x0=out$x[2]
      z0=out$z[2]
      v.x=out$v.x[2]
```

```
      v.z=out$v.z[2]
      Sys.sleep(t.pause)
   }
}

# Check it out
orb.plot()
orb.plot(v.x=-10000,new.plot=F)
orb.plot(v.x=-3000,new.plot=F)
orb.plot(v.x=-35000,time=700,new.plot=F,t.pause=0.1)
orb.plot(v.x=-45000,time=1000,new.plot=F,t.pause=0.1)
```

## Appendix 2. list of data files and their sources

We thank James Gibbs, Amy Roe, and Joe Besessi for sharing data for this workshop. Data from USGS were downloaded from the USGS Surface-Water Data site (http://waterdata.usgs.gov/nwis/sw). Data from published papers were either copied from the paper itself or the online supporting information. Data labeled FAO are from the UN Food and Agriculture Organization (www.fao.org). Data from Kuhnert & Venables were downloaded from http://cran.r-project.org/doc/contrib/Kuhnert+Venables-R_Course_Notes.zip. Data from books were either entered manually or downloaded from associated websites. For other sources, see the list of references.

| File | Source |
|---|---|
| Oxychem.txt | Qui & Hites 2008 |
| River_flow.txt | USGS |
| Cacti_v_tort.txt | James Gibbs |
| Muddy_Crk.txt | USGS |
| Thakali_Ni_EC50s.txt | Thakali et al. 2006 |
| Thakali_Cu_EC50s.txt | Thakali et al. 2006 |
| Eagles.txt | Amy Roe |
| DO_methods_1.txt | Wilcock et al. 1981 |
| DO_methods_2.txt | Wilcock et al. 1981 |
| Janka.txt | Kuhnert & Venables |
| Ozone.txt | Crawley 2008 |
| Wheat.txt | FAO |
| Crabs.txt | Zar 1999 |
| Ogeechee_tox_summary.txt | Personal data |
| Cu_tox_test.txt | Personal data |
| Carion_beetles.xls | James Gibbs |
| Stream_Cu.txt | Generated data |
| Daphnids.txt | Personal data |
| Cactus_width.txt | James Gibbs |
| Squirrel_color.txt | James Gibbs |
| Isolation.txt | Crawley 2008 |
| Dimethyl-death.txt | Generated data |
| Biohydrogen.txt | Personal data |
| US_GDP.txt | MeasuringWorth 2010 |
| Mammal_sleep.txt | Faraway 2005 |

## Disclaimer:

The views expressed in this workbook do not necessarily represent the views of USDA or the United States.