An introduction to data analysis and graphics with R
A workbook for the 2009 SUNY-ESF R workshop

Sasha Hafner (sdhafner@gmail.com) and Adam Ryan (acnayr1@gmail.com)

Last modified: March 10, 2009

Contents

Objective 4
1. R example 4
   1.1. Example: data import, model fitting/analysis, data export, graphics 4
   2.1. R overview and history 9
   2.2. Finding and installing R 9
   2.3. Running R: GUI & scripts 9
   2.3. R basics: commands, expressions, assignments, operators, objects, and using the R GUI 10
   2.6. R data types 13
   2.7. R data structures 15
   2.8. Functions, arguments, and packages 17
   2.9. Missing, indefinite, and infinite values 19
   2.10. Getting help 20
   Exercises 20
   3.1. Creating and working with vectors 22
   3.2. Vector arithmetic, some common functions, and vectorized operations 24
   3.3. Matrices and arrays 27
   Exercises 30
4. Data frames, data input, and data output 31
   4.1. Reading data from files 31
   4.2. Creating data frames manually 32
   4.3. Working with data frames 33
   4.4. Writing data to files 35
   Exercises 36
5. Graphics, part I 37
   5.1 The plot function 37
   Exercises 41
6. Manipulating data 42
   6.1. Modes, attributes, length, and coercion 42
   6.2. Indexing, subsetting, and splitting data 44
   6.3. Factors 50
   6.4. Dates and times 51
   6.5. Combining data 54
   6.6. Summarizing and manipulating data 55
   Exercises 61
7. Exploratory data analysis 62
   7.1. Summary statistics 62
   7.2. Dealing with detection limits 63
   7.3. Histograms, box plots, and probability plots 68

7.4. Normal quantile plots and cumulative probability plots (Crawley 2007: 281) 73
Exercises 77
8. One- and two-sample tests (and the R approach to statistical output) 79
8.1. t tests 79
8.2. The R approach to statistical output 81
Exercises 82
9. Classical linear models 83
9.1. The lm function, model formulas, and more on statistical output 83
9.2. Linear regression 83
9.3. ANOVA and pairwise comparisons 101
9.4. ANCOVA 110
Exercises 116
10. Nonparametric analogs to t tests and ANOVA 117
10.1. Wilcoxon signed-rank test 117
10.2 Kruskal-Wallis test 118
Excercises 118
11. Graphics II 119
11.1 Arranging multiple plots per page 119
11.2 More on the plot function 124
11.3 Annotating plots 134
11.4 Other high-level plotting functions 144
11.5 Graphics output 146
Exercises 147
12. Generalized linear models 150
12.1 The glm function 150
Exercises 157
13. Generalized additive models 158
13.1. The gam function 158
Exercises 164
14. Nonlinear regression 165
14.1 The nls function 165
Exercises 171
15. Grouping, loops, and conditional execution 172
15.1. Loops and grouping 172
15.2. Conditional statements 179
Exercises 181
16. Distributions and simulations 182
16.1. Available distributions 182
16.2. Monte Carlo simulations 189
16.3 Numerical simulations 194
Exercises 197
17. Functions 199
17.1 Writing functions 199
Exercises 206
18. Batch processing 208
18.1. Running R in batch mode 208

19. Specialized packages, related documents, and additional information 210
References 211
Appendix: list of data files and their sources 211
Disclaimer: 212

Objective

The objective of this workshop is to introduce participants to data analysis and graphics with R. In addition to the wide array of functions available in the base packages that are installed with R, more than 1600 contributed packages available for download, each with its own suite of functions. Some of the individual packages are the subject of entire books. Obviously, it would not be possible to demonstrate 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-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 your 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.

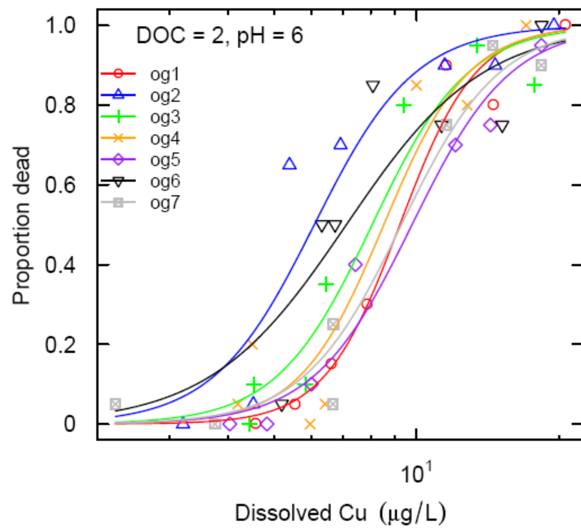
1. R example

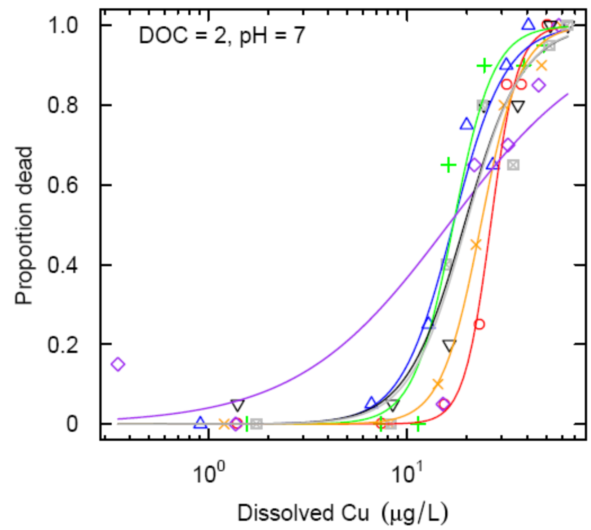
1.1. Example: data import, model fitting/analysis, data export, graphics

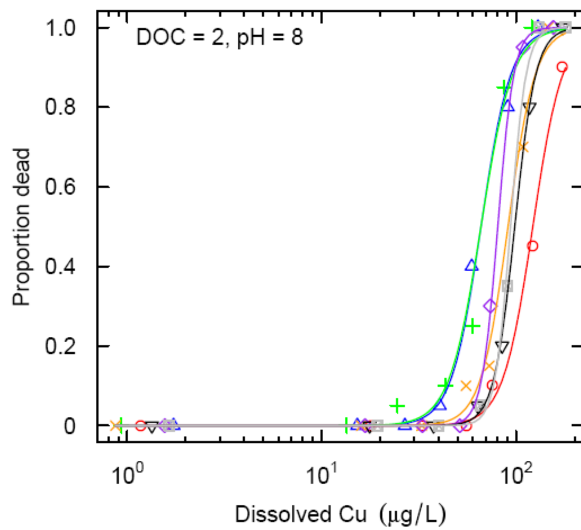
The following example demonstrates some of the capabilities of R for data analysis and production of graphics. The data analyzed in this example consist of copper Cu toxicity tests conducted with *Daphnia magna* designed to test the effects of dissolved organic matter (DOM) sources, dissolved organic carbon (DOC) concentration, and pH on Cu copper toxicity. The experimental design is a complete $7 \times 3 \times 3$ factorial design. Data consist of measured mortality at several Cu concentrations for each condition, replicated twice, resulting in about 1000 lines of data. The goal of our analysis in R is to fit a concentration-response model to each individual toxicity test (i.e. 120 models), use these models to estimate an LC50 for each condition, and finally fit an analysis of covariance model to the resulting LC50 data to test and quantify the effects of DOM source, DOC concentration, and pH on Cu toxicity. Additionally, we would like to make some figures along the way.

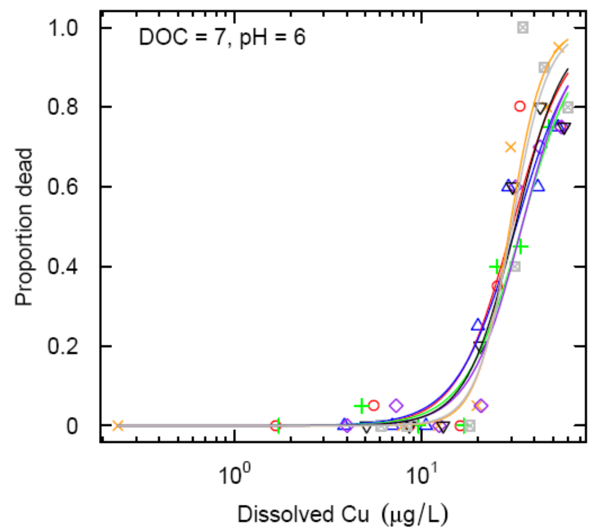
This entire analysis can be carried out relatively easily in R. The following results were produced using a single script file (just a text file that contains R code), which could be called up in the R GUI or Window's Command Prompt. The original 1000 or so lines of data are read in from a text file.

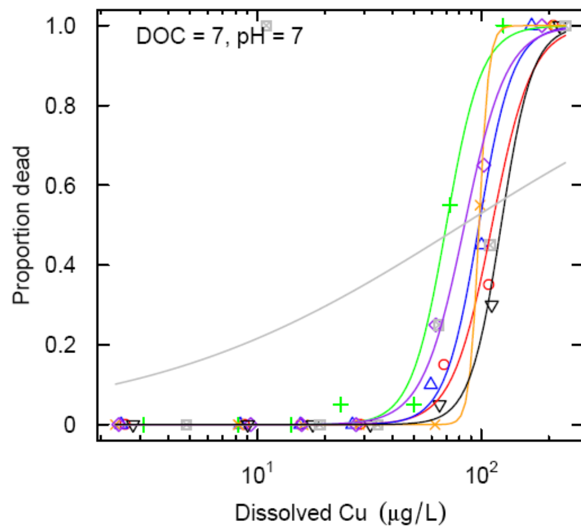
The plots on the next page show a sample of the concentration-response models fit to the individual toxicity tests. The script file specified that these plots should be exported to a pdf file. Other options include postscript, jpeg, and png.

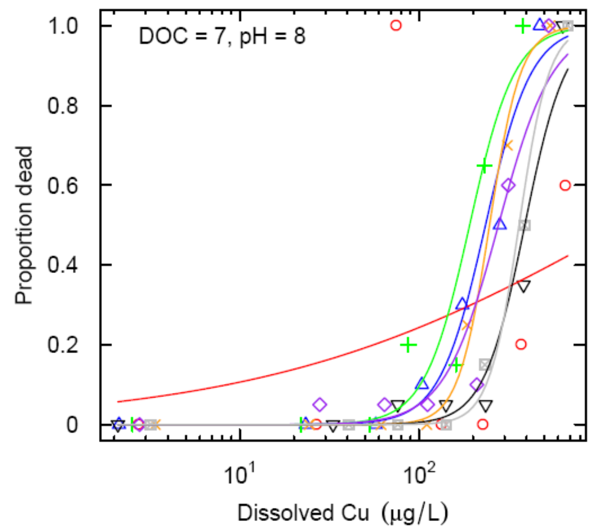

DOC = 2, pH = 6
og1
og2
og3
og4
og5
og6
og7
Proportion dead
Dissolved Cu (µg/L)


DOC = 2, pH = 7
Proportion dead
Dissolved Cu (µg/L)


DOC = 2, pH = 8
Proportion dead
Dissolved Cu (µg/L)


DOC = 7, pH = 6
Proportion dead
Dissolved Cu (µg/L)


DOC = 7, pH = 7
Proportion dead
Dissolved Cu (µg/L)


DOC = 7, pH = 8
Proportion dead
Dissolved Cu (µg/L)

The LC50s are extracted from the models shown above, and these data were written out to a tab-delimited text file, from which they could be copied and pasted directly into the following table. (Only a few of the 126 LC50s are shown below.)

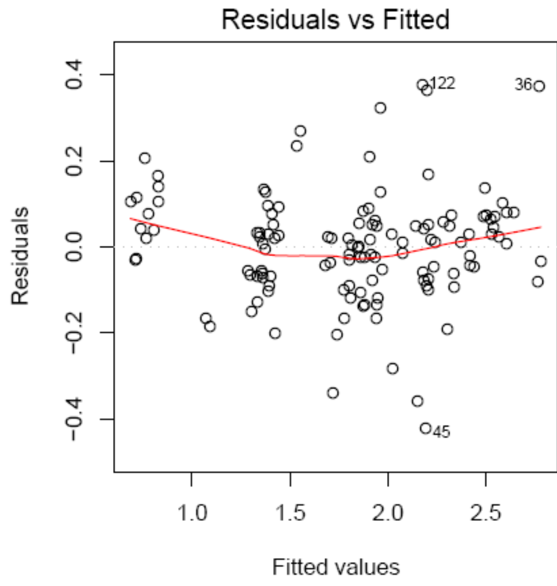
dom.source	***n.doc***	***n.ph***	***lc50***	***ph***	***c.doc***
Og1	2	6	9.23	6.38	0.894
Og2	2	6	6.08	6.33	0.9
Og3	2	6	8.06	6.3	0.896
Og4	2	6	8.53	6.35	0.929
Og5	2	6	9.79	6.33	0.971
...					

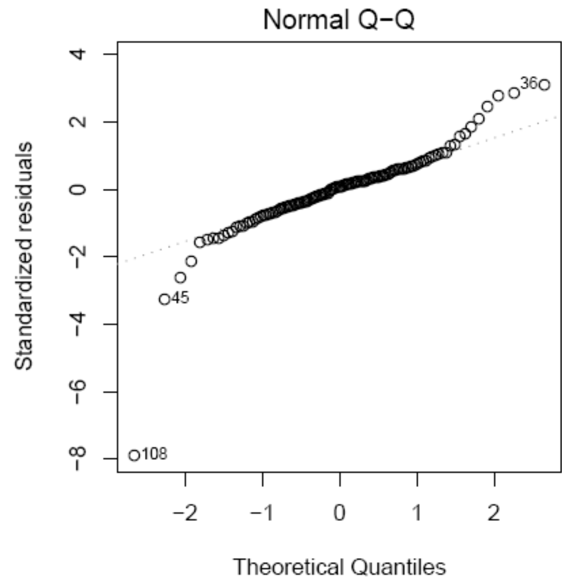
These LC50s were then analyzed with analysis of covariance to see if DOC concentration, DOM source, and pH have consistent effects on Cu LC50s. (Note that the data are passed to the ANCOVA analysis internally—there is no need to write them out.) Here is an ANOVA table from this analysis, again written out by R as a tab-delimited text file.

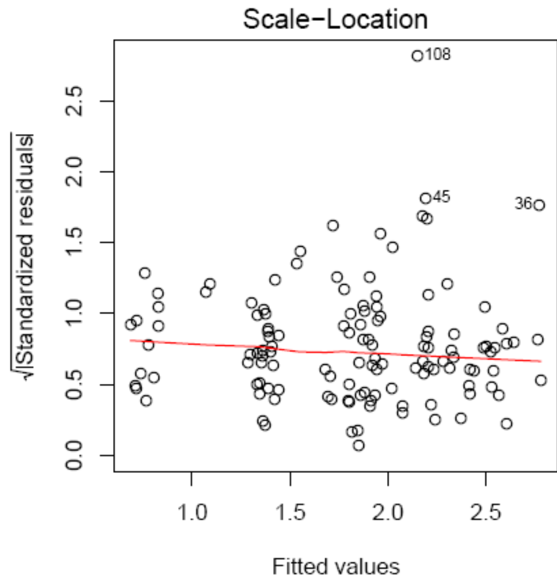
Source	***Df***	***Sum Sq***	***Mean Sq***	***F value***	***Pr(>F)***
dom.source	6	0.2371	0.03951	2.112	0.05745
l.doc	1	17.23	17.23	921.1	2.82e-55
ph	1	8.27	8.27	442.1	2.47e-40
dom.source:ph	6	8.54	1.423	76.09	7.396e-37
l.doc:ph	1	0.8759	0.8759	46.82	4.602e-10
Residuals	110	2.058	0.01871	NA	NA

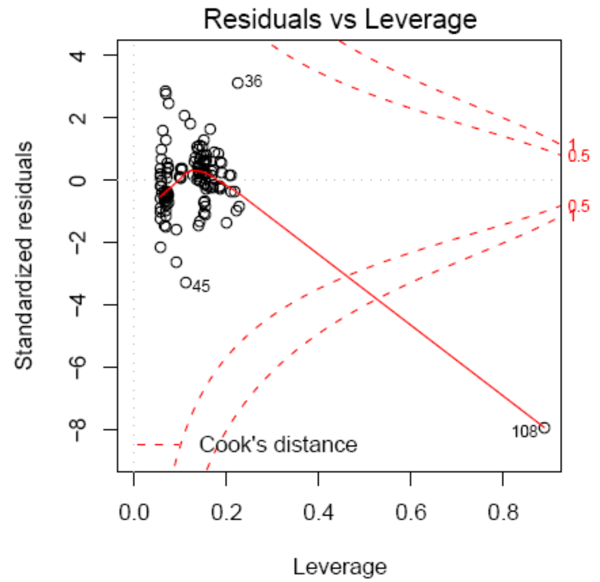
Finally, some plots are made to visualize the ANCOVA analysis. Some of the default plots produced by R for an ANCOVA model are shown on the next page for this model.

lm(l.lc50 ~ (dom.source + l.doc + ph)^2 − dom.source:l.doc)

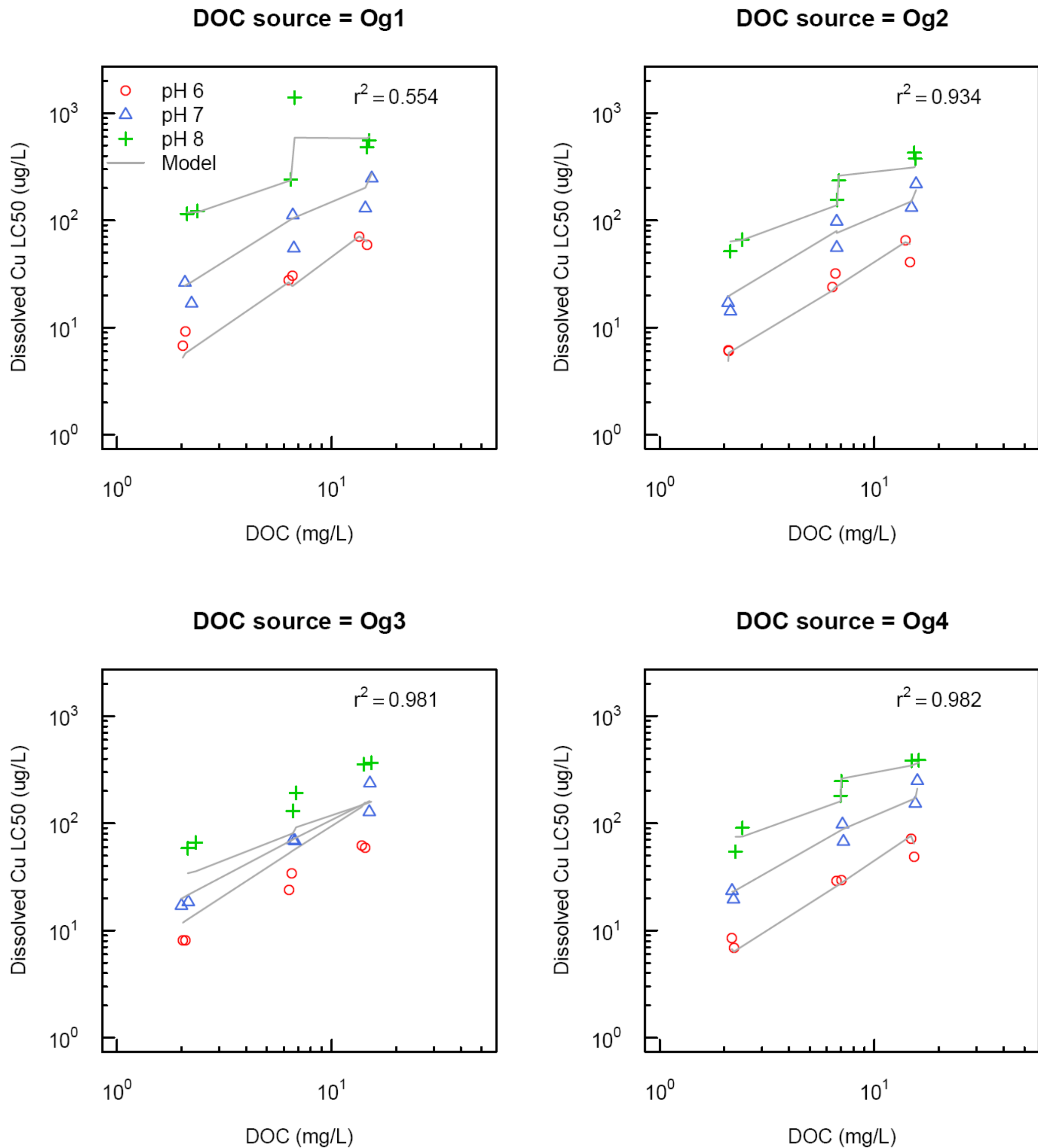








Lastly, all of the LC50s were plotted, along with the ANCOVA predictions. A sample is shown below.



We hope that this example demonstrates a few things about R: its utility for both common and specialized analyses, its integrated nature—allowing for the results of one analysis to be used in following analyses and for publication-quality graphics to be produced in conjunction with analyses, and its flexibility.

2. Introduction to R

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

2.1. 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). R uses a "dialect" of the S language, which was developed (principally) by J. Chambers at Bell Laboratories. R is currently maintained by the R Core Group, 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 (the current version as of March 3, 2009 is 2.8.1), however, changes are usually not major.

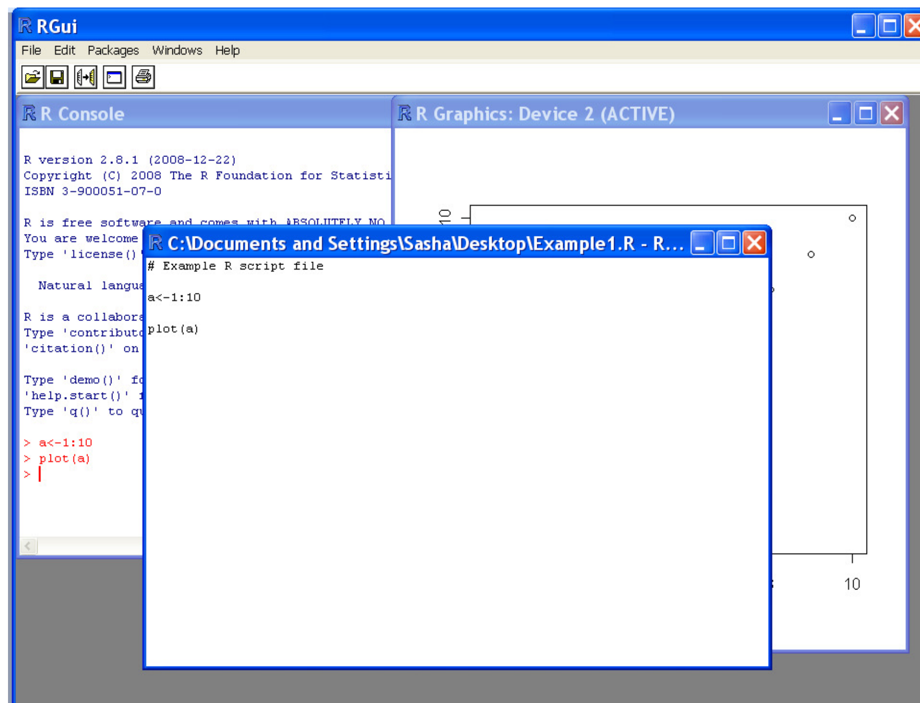
2.2. Finding and installing R

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 GUIs differ slightly across systems, the R commands do not.

2.3. Running R: GUI & scripts

There are two basic ways to use R on your machine: through the graphical user interface (GUI), where R evaluates your code and returns results as you work, or by writing, saving, and then running R script files. (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.) R 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-line interface (i.e. the DOS shell), other shells, batch files, or the R GUI. All the code covered in this workbook will (of course) work directy typing into the GUI, or by saving in a script file and then submitting this file to R. There is 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`. For any code that you want to save, rerun, and modify, you should consider working with R scripts.

So, how should 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` (Note that R.2.8.1/bin must be added to the PATH environment variable). The R GUI comes with a pretty basic script editor, shown below (the window in the center). This allows you to edit and create scripts, and also to submit commands to the R GUI with the click of a button.



Although this editor is a little better than, say, Notepad, the Windows version does not have syntax highlighting. The Mac version includes syntax highlighting and some other useful features.

There are some useful free text editors available that can be set up with R syntax highlighting and other features, including Notepad++ and TINN-R. TINN-R is a free text editor that is designed specifically for working with R script files (its name is recursive, and stands for TINN Is Not Notepad). It will automatically apply syntax highlighting to files that have the extension ".R" (or ".Q"), and it has the ability to send code directly to the R GUI with the click of a button.

TINN-R also allows you to have multiple files open at once. You can download this program at http://www.sciviews.org/Tinn-R/. Note that the newest version of TINN-R is less flexible than the older version (1.17.2.4).

2.3. R basics: commands, expressions, assignments, operators, objects, and using the R GUI

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 the command and prints the result.

```
> 1+1
```

```
[1] 2
```

For the above command, the result is printed to the screen and lost—there is no assignment involved. 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. 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 such as Fortran or C) using the assignment operator `<-`. When a command is used for assignment, the result is no longer printed to the GUI console (unless you surround the entire command in parentheses).

```
> 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. Spaces usually do not matter in R, as long as they do not separate a single operator or a variable name.

```
> 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 or specifying nested models.

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.

If you want to assign the same value to several symbolic variables, 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
```

If you find that you get stuck in a bad command, just hit the Esc key to get back to the regular prompt.

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

`$`

`[]`

Grouping commands
`{}`

Making sequences

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

Others

`#`	commenting
`;`	separating commands
`~`	model formula specification
`()`	order of mathematical operations

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. It is good practice to not rely on a saved workspace. Instead, you should save the commands that created it as a script 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 as a script file.

2.6. R data types

The term "data type" in R refers to the type of data that is present in a data structure, not the type of data structure itself. There are four different types of data in R. These 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 in the R GUI, you must surround individual elements with quotes. Otherwise, R will look for an object.

```
> name<-John Smith
Error: unexpected symbol in "name<-John Smith"
```

```
> 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 unless spaces are present in individual elements (e.g. John Smith).

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

or

```
> a<-T

> a
[1] TRUE
```

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.

```
> cnum1<-10 + 3i

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

You can use the class function to see what type of data is stored in any symbolic variable.

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

> class(a)
```

```
[1] "logical"

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

2.7. R data structures

Data in R are stored in data structures (or 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 discuss how to make these different data structures shortly; the following examples simply give you an idea of their structure, i.e. what they actually look like.

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. In R, data type is referred to as mode, so another way of saying this is these data structures can only contain elements of the same mode. (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.) The mode of any object can be checked using the function `mode`.

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

Data frames are similar to matrices: they are two-dimensional data. 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.
1,8.72),living=c(T,F,T))

> dat
     sp sex weight living
1   Dog   F  75.20   TRUE
2   Cat   M 186.10  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. We will use the variables defined above 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
```

2.8. 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 and assigns the given object (e.g. `a` or `b` above) to the correct argument. 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`:

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

If values are not given for the arguments `mean` and `sd`, the default values (0 and 1, respectively) are used.

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 may 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. Of course, it may be easier to assign parts of the above line 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 the function `paste` in place of explicit character data. Keep this idea in mind, and you can write some very efficient code, especially when you need to repeat a task and can write your code in a loop (covered later).

Many functions (including `sum, plot and rnorm`) come with the R "base package", i.e. they are loaded and ready to go as soon as you open R (these packages contain the most common functions, you can find a list of the packages at http://cran.r-project.org/doc/FAQ/R-FAQ.html). 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 1600 contributed packages. These packages 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, or with the command:

```
> 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, which we will use later, 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 Rprofile.site that will be executed every time you start R.)

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 (note that the syntax is a little funny here):

```
> 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 should consider writing your own function. This topic is covered in a later section.

2.9. Missing, indefinite, and infinite values

It is often the case that real data sets contain missing values. R uses the marker `NA` (for "not available") to indicate a missing value. R returns NA for any operation carried out on an 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. To determine if a value is missing, use the `is.na` function. (This function can also be used to set elements 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`. Of course, any of these functions can be negated with a `!`.

2.10. Getting help

It is usually easy to find the answer to any question you have about a specific function or about R in general. There are several good books available, some of which are cited throughout this workbook and listed at the end. You can also find free detailed manuals on the CRAN website—some of these are cited throughout this workbook as well. 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. However, help files are very concise. You can bring up a help file by typing `?` and then the function name.

```
> ?aov
```

There is an R help mailing list, 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 very helpful answers to even seemingly obscure questions, but they are generally not shy about telling someone that they need to go back and read the posting guide.

Exercises

1. Caculate $3^{7.5}$ and have R print the result out to the R console window. Now assign this expression to the variable `x`. Assign the value 32 to the variable `y`. Now calculate the sum of `x` and `y` and assign it to the variable `z`. Print out the value of `z` to the GUI. Finally, ask R if `y` is greater than `x` and have the result printed out to the GUI.

2. Create two vectors, called `a` and `b`, by entering the two following lines of code. (We will cover both the colon notation for creating vectors and the `rnorm` function in more detail in later sections.)

```
> a<-21:30
> b<-rnorm(10)
```

Make a scatterplot of `b` vs. `a` using the `plot` function two different ways—first using positional arguments and then named arguments. If you are unsure of the arguments in the `plot` function, check out the help file.

3. 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.

4. If the above was a breeze for you, use the colon operator and the `c` function to create a vector that looks like this: `1 2 3 4 5 6 7 8 9 NA NA NA 21 22 38`. Now use the `subset` function to remove the `NA` values from your vector.

3. Vectors, matrices, and arrays

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

3.1. Creating and working with vectors

There are several ways to create vectors 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
```

The function `seq` (for sequence) is more flexible. Its typical arguments are `from`, `to`, and `by`.

```
> 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
```

Any of these commands could be assigned to a 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 2.5 2.6
[18] 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 these functions 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 vectors with a horizontal orientation, R does not have “columns vectors” and “row vectors”, and vectors do not have a fixed orientation. This makes the vectors in R very flexible.

Vectors do not need to contain numbers; R recognizes the following data types: numeric, logical, and character. In R terminology, these three types of data are called modes. As mentioned earlier, all the data in any vector must be of the same mode.

Logical vectors are very useful in R for subsetting data. For relational commands, the repeat rule that applied to arithmetic applies here.

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

Character data can be entered using single or double quotes.

```
> seasons<-c("Spring","Summer","Fall","Winter")

> seasons
[1] "Spring" "Summer" "Fall"   "Winter"
```

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), e.g.,

```
> paste("A","B","C",TRUE,42)
[1] "A B C TRUE 42"
```

Note that the `paste` function is very different from `c`. The former 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 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 (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 (the constant `letters` is a vector of lowercase letters).

3.2. 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), is a quality that makes R a very efficient programming language.

```
> 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
```

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
`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 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.

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

> (7 - 2)*4
[1] 20

> 10^1:5
[1] 10  9  8  7  6  5

> 10^(1:5)
[1] 1e+01 1e+02 1e+03 1e+04 1e+05
```

Many functions in R are capable of accepting vectors, matrices, arrays, lists, or data frames as the input for single arguments, and returning a data frame with the same dimensions. These functions are called vectorized, and they make vector manipulations very efficient. Examples of such functions include `log`, `sin`, and `sqrt`. For example,

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

or

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

```
[10] 3.162278
```

The previous is 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 2.645751
2.828427 3.000000
[10] 3.162278
```

But it is not the same as the following, where all the numbers are interpreted as individual function 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
```

When dealing with lists, matrices, and arrays, there are some functions designed for carrying out vectorized operations: `apply`, `lapply`, and `apply`. We will cover these in a later section.

3.3. Matrices and arrays

Arrays are multi-dimensional collections of elements. 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). Most 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. The arguments can be found in the associated help file:

```
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
```

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. 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
```

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 does have 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 must be a vectorized function.

```
> x<-y<-1:5

> outer(x,y,"*")
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    2    4    6    8   10
[3,]    3    6    9   12   15
[4,]    4    8   12   16   20
[5,]    5   10   15   20   25
```

Exercises

1. Generate a vector of numbers that contains the sequence 1 2 3. . .10. Calculate the square root of each of the values in this vector.

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).

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. Also try to determine the maximum, minimum, and mean.

4. 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.

4. Data frames, data input, and data output

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

4.1. Reading data from files

Using data frames in R (or other statistical programs such as SAS) requires different organization than what one might use in a spreadsheet program. In general, it usually makes the most sense to place all the values for a given variable in the same column. The easiest way to make 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, but `read.table` is the most commonly used approach. As listed in the associated help file, the first four arguments for the function are given below.

```
read.table(file, header = FALSE, sep = "", quote = "\"'")
```

In practice, only the arguments `file` and `header` are usually needed. R is very flexible in how it reads in data from text files. Typically, organization should follow one of the following two formats.

With row labels:

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

Without row labels:

```
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
. . .
```

If you use the second option (usually easier), you must specify `header=TRUE` as one of the arguments. With this option, R will assign row numbers, based on the order of the observations. So, for example, if the data shown above without row labels were in the text file River_flow.dat, they could be read in and assigned to the data frame `flow.dat` using the following command.

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

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) you can include a file path, but note that the path should have forward, not backward, slashes.

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.

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 with `NA`. You can also specify a different code for missing values, e.g. `na.strings="."`. (Note the use of `\t` to represent a tab character. This is an example of an escape sequence.)

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.

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. Alternatively, data can be saved directly from Excel, e.g. as "Formatted Text (Space delimited)" (*.prn) (although this can be problematic for spreadsheets with a large number of columns), or 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.

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

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

4.2. 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)

> data<-data.frame(sample.date=date,mass.mean=mass)
```

```
> data
   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(data)<-c("Date","Mass")
> data
          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.

```
> data<-
data.frame(sample.date=date,mass.mean=mass,row.names=c("D","E","A","B"))
> data
   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. This topic will be covered later. Row names can also be specified for an existing data frame with the `row.names` function (not to be confused with the `row.names` argument).

4.3. Working with data frames

So what do you do with data in R once it is in a data frame? Many R functions require vectors as arguments, and many operations are carried out on 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 different data from USGS:

```
> mud.crk.dat<-read.table('Muddy_Crk.txt',header=TRUE)
```

To see what the column names are, we can use the function `names` (again, this function can be used for both checking and changing column names):

```
> names(mud.crk.dat)
 [1] "site.no"        "date"           "discharge"      "discharge.code"
 [5] "max.temp"       "max.temp.code"  "min.temp"       "min.temp.code"
 [9] "DO.max"         "DO.max.code"    "DO.min"         "DO.min.code"
```

The `$` notation just uses a `$` between the data frame name and column name to specify a particular column.

```
> mud.crk.dat$min.temp
  [1]  3.1  2.8  3.2  3.4  3.1  2.9  3.1  3.3  3.2  3.0  3.3  3.3  3.2
 [14]  2.9  2.8  2.8  2.7  2.9  2.8  2.9  3.2  2.8  2.9  2.9  2.9  3.0
 . . .
[625]  7.6  7.3  7.6  7.6  7.5  7.7  7.0  6.5  6.3  6.4  8.0  7.7  7.9
[638]  7.9  7.9  7.8  7.9
```

The expression `mud.crk.dat$min.temp` could be used just as you would any other vector.

The second option 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(mud.crk.dat)
> min.temp
  [1]  3.1  2.8  3.2  3.4  3.1  2.9  3.1  3.3  3.2  3.0  3.3  3.3  3.2
 [14]  2.9  2.8  2.8  2.7  2.9  2.8  2.9  3.2  2.8  2.9  2.9  2.9  3.0
 . . .
[625]  7.6  7.3  7.6  7.6  7.5  7.7  7.0  6.5  6.3  6.4  8.0  7.7  7.9
[638]  7.9  7.9  7.8  7.9
```

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(mud.crk.dat)
> min.temp
Error: object "min.temp" 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 be used to add columns to a data frame. For example, if we want to add a column to this data frame that has discharge in m^3/s, we can use the following code.

```
> mud.crk.dat$discharge.m3<-0.0283*mud.crk.dat$discharge
```

Both the `$` notation and the `attach` and `detach` functions can be used to specify data vectors within any other function. However, there are other (generally preferable) options when using functions. For some functions, you can specify the data frame that should be used with the data argument, e.g. `data=mud.crk.dat`, and then refer to the column within the data frame directly by name. For other functions, you can use the `with` function. Although it looks a bit clunky, the `with` function can save code and help prevent user errors.

R functions have various options for dealing with `NA`s, depending on the function. Observations that have missing values can be removed from a vector using `na.omit`.

```
> x<-c(1:5,rep(NA,10),16:20)
> x
 [1]  1  2  3  4  5 NA NA NA NA NA NA NA NA NA NA 16 17 18 19 20

> na.omit(x)
 [1]  1  2  3  4  5 16 17 18 19 20
attr(,"na.action")
 [1]  6  7  8  9 10 11 12 13 14 15
attr(,"class")
[1] "omit"
```

Note that the result of `na.omit` contains more information than just the non-NA values. Alternatively, to determine if a row is a complete case, use the function `complete.cases`.

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

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

4.4. 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 mud.crk.dat dataframe we could use the following code:

```
> write.table(mud.crk.dat,"mud_crk.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.

It is possible to save all R output in a file using the function `sink`. 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 Ni EC50s for barley root elongation in soils, along with relevant soil chemistry. Open the file to see how it is formatted, and then read the data into R using the function `read.table`. Print the resulting data frame to the screen to make sure the header is read in correctly. Now try reading this file in again, but this time, specify the separator. Notice any problems? Read in the data in wheat.txt, which contains wheat yield data from the US and Mexico, using both the default separator and specifying it using the `sep` argument. Does this file seem different from Thakali_Ni_EC50s.txt?

2. Determine the minimum and maximum soil pH in your new data frame that contains the Thakali data, and print these values to the screen. Next, add a new column to the data frame that contains the $\log_{10}$ of the EC50.

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. See if you can eliminate the row labels. Try changing the separator to a comma, and then to a tab.

4. Open the Excel file Carion_beetles.xls. 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`. Once you get this figured out, take a look at the original file again and see what you might have changed to make things easier.

5. Graphics, part I

Dalgaard 2008: Section 2.2, R-Intro: Section 12

5.1 The plot function

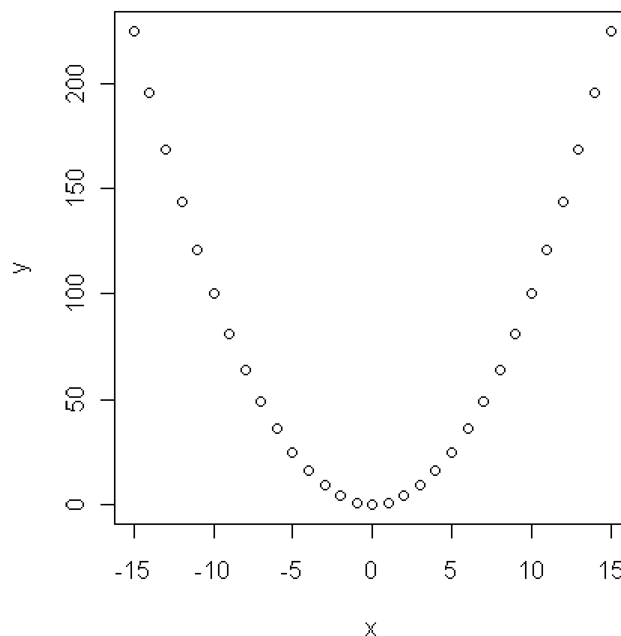
It is possible to produce publication-quality graphics relatively easily in R. However, in this first section on graphics, we are only going to skim the surface of graphics in R, primarily through the `plot` function. This function produces a plot as a side effect, but the type of plot produced depends 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, ...)
```

where "`...`" (an ellipsis) represents additional, optional arguments.

Here we create two simple series and plot them with the most basic plot command (i.e. no arguments other than x and y are specified).

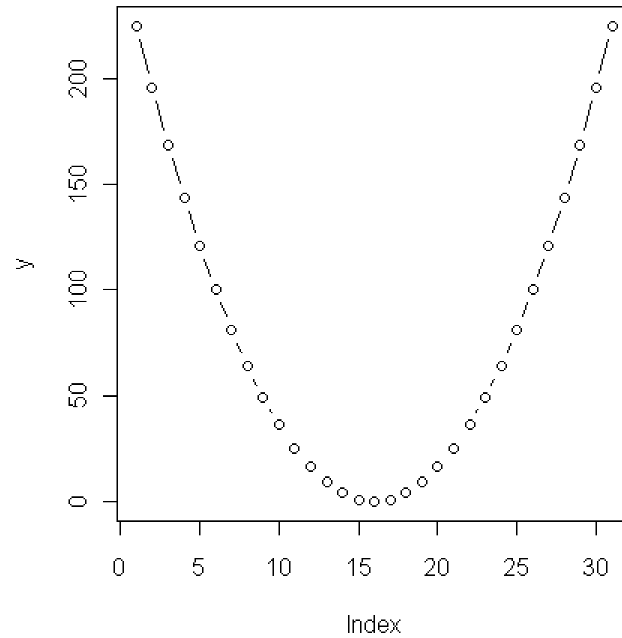
```
> x<-c-15:15
> y<-x^2
> plot(x,y)
```



There are several types of plots that can be specified with the `type` argument. Some examples include `type="n"` for no data series, `type="p"` for points, `type="l"` for lines, `type="b"` for

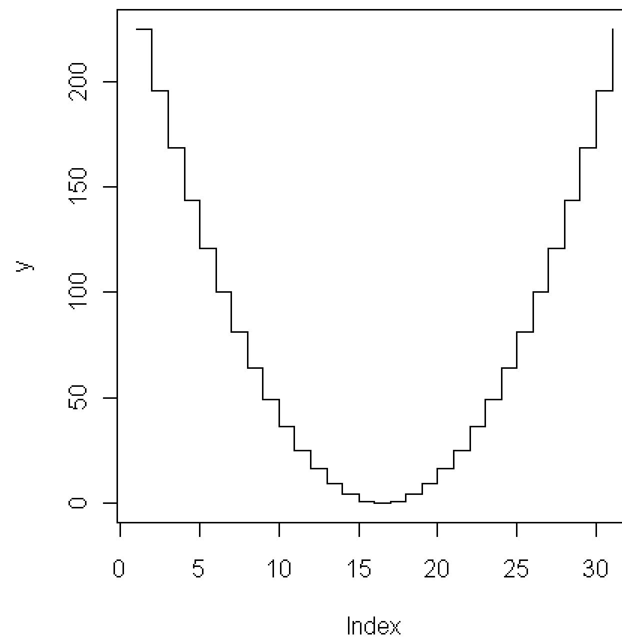
both lines and points, and `type="s"` for step function. The code below shows the same data presented above, but uses `type="b"`.

```
> plot(y,type="b")
```



Now try using `type="s"`:

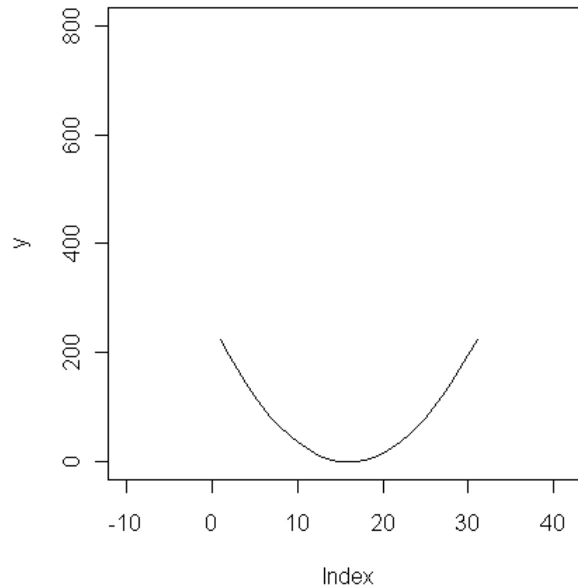
```
> plot(y,type="s")
```



Notice that in the two plots above, the x-axis label and scale are different than in the first plot, and this is because no `x` series argument was supplied in the plot command in the last two plots. This results in the x position being indexed by its order of occurrence.

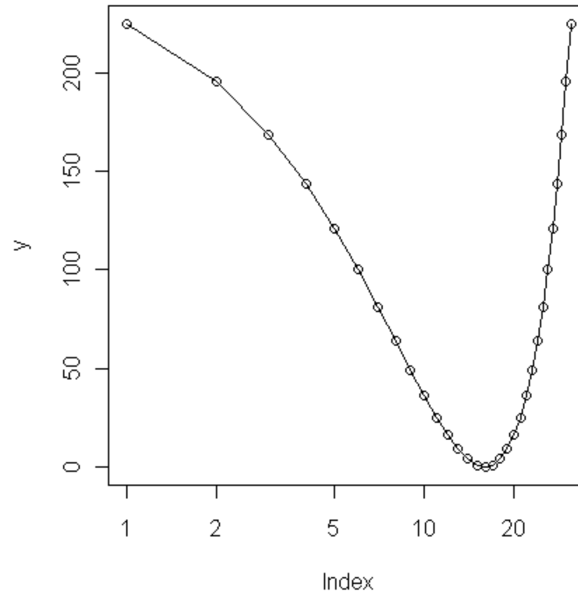
It is very easy to change the limits of the x and y scales. The arguments `xlim` and `ylim` are used, but note that the values for the limits must be specified as a vector.

```
> plot(y,type="l",xlim=c(-10,42),ylim=c(0,800))
```



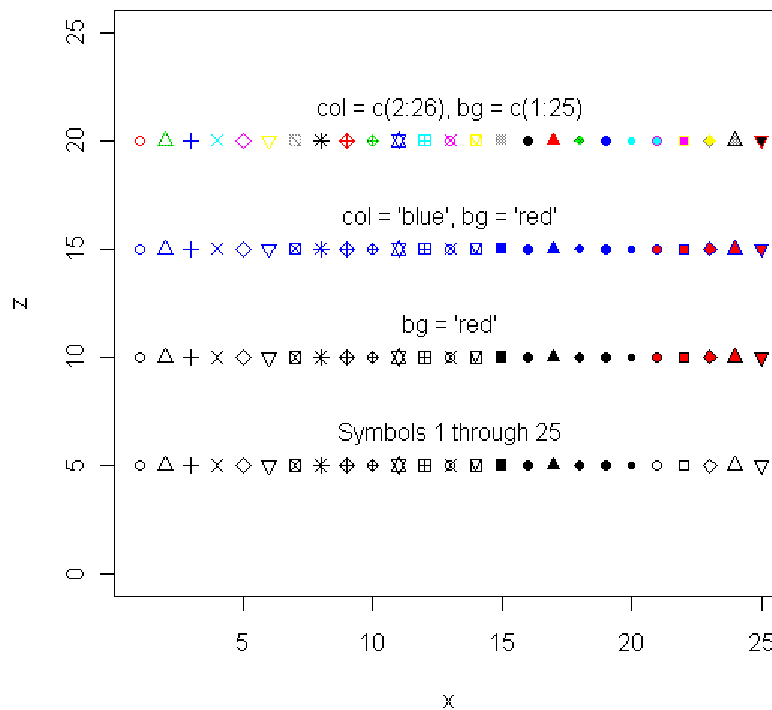
To make logarithmic axes, use the `log` argument. This can be specified for one axis at a time (i.e. `log="x"` or `log="y"`) or for both axes at once (i.e. `log="xy"`).

```
> plot(y,type="o",log="x")
```



There are several different plotting symbols that are available in R, as demonstrated in the following code.

```
> sym<-1:25
> x<-1:25
> z<-rep(5,25)
> plot(x,z,pch=sym,ylim=c(0,25))
>   text(13,6.5,"Symbols 1 through 25")
> points(x,z+5,pch=sym,bg="red")
>   text(13,11.5,"bg = 'red'")
> points(x,z+10,pch=sym,col="blue",bg="red")
>   text(13,16.5,"col = 'blue', bg = 'red'")
> points(x,z+15,pch=x,col=x+1,bg=x)
>   text(13,21.5," col = c(2:26), bg = c(1:25)")
```



Symbols can be specified by a single number or by a vector of numbers. At this time, 25 symbols are available. Note that text and numbers can be plotted in place of symbols–this is useful when data points need to be identified. Symbols 21 through 25 have a background color, which is set with the `bg` argument. Currently, 657 different colors are available in R. It is also handy to use vectors of numbers for colors when many different data series are being plotted. Internally, all colors are assigned a number, and if a number is specified for a color, R will interpret it correctly as a color.

Exercises

1. Generate a vector called `x` that goes from -2π to 2π in whatever interval you want. Now calculate the cosine of `x`, and put both `x` and its cosine together in a new data frame. Plot cosine(x) vs. x as a smooth line, and then repeat but plot using a stepped line.

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 vs. the distance from the suspected source (OxyChem). Try using linear and logarithmic axes. Try a few plotting symbols and colors.

6. Manipulating data

Crawley 2007: Chapters 2 & 4, Dalgaard 2008: Sections 1.2 & 10.1.3, R-Intro: Section 5.4, R-Lang: Section 3.4

6.1. Modes, 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
```

Attributes are important internally for determining how objects should be handled by various functions. The two attributes mentioned above, mode and length, are called intrinsic. All other attributes are considered non-intrinsic, and can be gotten for any object by using the `attributes` function. Attributes are a bit confusing, but you will probably never have to even think about them. If you are interested, two examples are shown below.

```
> y<-c(1:10,rep(NA,5))
> y
 [1]  1  2  3  4  5  6  7  8  9 10 NA NA NA NA NA
> attributes(y)
NULL

> y.2<-na.omit(y)
> attributes(y.2)
$na.action
[1] 11 12 13 14 15
attr(,"class")
[1] "omit"

> x<-1:10
> y<-2*x + rnorm(10,mean=0.5,sd=1)
> mod<-lm(y ~ x)
> class(mod)
[1] "lm"
> attributes(mod)
$names
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "xlevels"       "call"          "terms"         "model"

$class
[1] "lm"
```

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)
NULL
> 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
```

It is often necessary to change the class 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 class 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 common examples are `as.numeric` and `as.vector`.

6.2. Indexing, subsetting, and splitting 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(5,1,3.2,8)
> v1
[1] 5.0 1.0 3.2 8.0
```

If we want the 3rd observation:

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

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

Return the 1st through 3rd observation:

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

Return 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. For example, using the data frame that was originally read in above:

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

> 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 (note that you need to include the comma). Therefore:

```
> 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 working with statistical output) or with data frames whose column order may change in the future (e.g. if you add columns to the data file.

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

Note that you can specify multiple column names using the `c` function.

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

Since you are using character data to identify columns in the last two examples (both `"site"` and `c("agency","site")` are actually chacter vectors), it is also possible to use the function `paste`.

```
> f.half<-"agen"
> s.half<-"cy"
> flow.dat[1:5,paste(f.half,s.half,sep="")]
[1] USGS USGS USGS USGS USGS
Levels: USGS
```

This example is not very useful, but if you have some code written into a loop, and want to select a different row or column on each pass of the loop, the above procedure can save a lot of code and effort.

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. 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.

```
> 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. Multiple constraints can also be used:

```
> subset(flow.dat,discharge>4000 & site!=4232730)
    agency    site       date discharge flag.discharge
438   USGS 1509000 2006-03-14      4540              A
544   USGS 1509000 2006-06-28      6050              A
545   USGS 1509000 2006-06-29      4760              A
```

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. Of course, if the element within the list has multiple elements, you could use indexing to select specific elements within it. If you do all this indexing in one step, you can end up with some strange looking code.

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

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

(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.)

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.

```
sp.flow.dat<-split(flow.dat,flow.dat$site)
```

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

```
> sp.flow.dat
$`1509000`
    agency    site       date discharge flag.discharge
366   USGS 1509000 2006-01-01       812              A
. . .
730   USGS 1509000 2006-12-31       628              A

$`4232730`
    agency    site       date discharge flag.discharge
1     USGS 4232730 2006-01-01        75              P
. . .
365   USGS 4232730 2006-12-31      1560             Ae
```

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. However, there are usually other ways to arrive at this type of result.

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.

```
> flow.dat[order(flow.dat$discharge),]
    agency    site       date discharge flag.discharge
77    USGS 4232730 2006-03-18        31              P
92    USGS 4232730 2006-04-02        50              P
. . .
544   USGS 1509000 2006-06-28      6050              A
273   USGS 4232730 2006-09-30        NA           <NA>
```

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

Note that 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. These row labels can be handy, especially when they are user-specified (using the `row.names` argument in `data.frame` or `read.table`).

The previous discussion in this section shows how to get data that meet certain criteria from a data structure. But sometimes it is important to know where that 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(60,flow.dat$discharge)
[1] 97
```

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

```
> match(c(55,60,72),flow.dat$discharge)
[1] 50 97 66
```

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

```
> match(max(flow.dat$discharge),flow.dat$discharge)
[1] 273
```

Let's take a look at the value.

```
> flow.dat$discharge[273]
[1] NA
```

Oops. Try again.

```
> match(max(na.omit(flow.dat$discharge)),flow.dat$discharge)
[1] 544
> flow.dat$discharge[544]
[1] 6050
```

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

```
> which(flow.dat$discharge<60)
[1]  50  77  78  80  92 100 106 116
```

Of course, you can specify multiple constraints.

```
> which(flow.dat$discharge<60 & flow.dat$discharge>50)
[1]  50  78  80 100 106 116
```

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

```
> which(is.na(flow.dat$discharge))
[1] 273
```

6.3. 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. R automatically recognizes non-numerical data as factors when data are read in, but if numerical data are to be used as a factor, this 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)
> 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.

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.

6.4. Dates and times

Dates in time are handled quite 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, 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.

```
> 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)
[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(2007,2007,20
08),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$yr,paste(sepa
rate.times$hr,separate.times$min,sep=":"))
> comb.times
[1] "April 8 2007 7:10" "April 9 2007 7:12" "March 9 2008 9:1"
```

Oops, looks like the minutes with only a single character are going to give us trouble.

```
rate.times$min<-as.character(separate.times$min)
> separate.times$min<-
ifelse(nchar(separate.times$min)==1,paste(0,separate.times$min,sep=""),sep
arate.times$min)
```

```
> comb.times<-
paste(separate.times$month,separate.times$day,separate.times$yr,paste(sepa
rate.times$hr,separate.times$min,sep=":"))
> comb.times
[1] "April 8 2007 7:10" "April 9 2007 7:12" "March 9 2008 9:01"
```

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 
```

6.5. 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 (i.e. "row bind"). The function `cbind` (i.e. "column bind") carries out an analogous operation with columns of data. The `merge` function is used to merge data frames by some common variable.

```
> 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

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


> wide.stuff.dat<-cbind(stuff.dat,y=c(3,1,5))
> 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 two new data sets. Both of these data sets are on metal toxicity to barley seedlings.

```
> ni.tox.dat<-read.table("Thakali_Ni_EC50s.txt",header=TRUE)
> ni.tox.dat
         soil ec50.ni ph.soil   oc ph.sol c.ni c.na  c.mg   c.k   c.ca
1   Houthalen    54.5    3.56  1.7   3.74 36.4 15.3   9.9  21.1   32.8
2     Zegveld  1928.2    4.11 33.1   3.88 72.9 56.3 148.0  88.3  980.0
3   Rhydtalog   533.5    4.20 12.5   4.19 42.7 36.1 110.0  46.8  551.0
4    Jyndevad    73.9    4.48  1.3   4.55 21.2 14.6  26.3  72.8  342.0
5 Kovlinge II   202.5    5.13  2.5   5.44 27.5 20.1  86.9  47.8  469.0
6      Borris   193.0    5.58  1.3   5.82 28.8 28.9 121.0 207.0 1060.0
7      Woburn   707.6    6.07  4.3   5.70 24.2 23.2 219.0  17.6  907.0
8   Ter Munck   248.2    6.74  1.1   6.83 17.5 45.3 106.0 249.0 1110.0

> cu.tox.dat<-read.table("Thakali_Cu_EC50s.txt",header=TRUE)
> cu.tox.dat
          soil ec50.cu ph.soil    oc ph.sol c.cu c.na  c.mg   c.k  c.ca
1   Nottingham   150.5    3.36  5.20   3.72 1.33 29.0  34.3  65.6  98.4
2    Houthalen    38.9    3.38  1.90   3.71 0.70 10.1   4.0  31.0  21.4
3    Rhydtalog   194.0    4.20 12.90   4.47 0.83 25.1  49.4  44.1 214.0
4      Zegveld   570.5    4.75 23.30   3.88 0.48 38.6 124.0  76.7 799.0
5   Kovlinge I   101.4    4.76  1.60   4.88 0.30 12.7  37.5  30.2 140.0
6      Souli I    64.8    4.80  0.41   5.09 0.06 11.1  13.4   0.0  72.0
7  Kovlinge II   168.8    5.06  2.40   5.22 0.27 14.0  41.1  34.7 218.0
8  Montpellier    61.2    5.18  0.76   5.63 0.95 24.9  38.4  23.9 147.0
9    Aluminusa   147.3    5.44  0.87   6.12 0.05 62.3  43.2  15.7 120.0
10      Woburn   392.2    6.36  4.40   7.37 0.50 17.6  94.5  24.1 412.0
11   Ter Munck   227.6    6.80  0.98   7.35 0.32 32.7  65.9 124.0 704.0
```

The function `merge` can be used to combine data frames, matching observations based on a specified column or columns. To merge these data sets by soil:

```
> tox.dat<-merge(ni.tox.dat,cu.tox.dat,by="soil",suffixes=c(".n",".c"))
> names(tox.dat)
 [1] "soil"      "ec50.ni"   "ph.soil.n" "oc.n"      "ph.sol.n"
 [6] "c.ni"      "c.na.n"    "c.mg.n"    "c.k.n"     "c.ca.n"
[11] "ec50.cu"   "ph.soil.c" "oc.c"      "ph.sol.c"  "c.cu"
[16] "c.na.c"    "c.mg.c"    "c.k.c"     "c.ca.c"
```

6.6. Summarizing and manipulating data

R has some built-in functions for producing summaries of data (the most obvious being `summary`, which is described in a following section) but custom-made summaries can be produced with the functions `table`, `sapply`, `tapply`, `aggregate`, and others. These functions are useful for other tasks as well.

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
```

The function `apply` can be used to apply a function to sections of an array (includes matrices), e.g. to multiple columns within a matrix. The function `lapply` applies a function to each individual element of a structure. So, if applied to a vector, the function is applied to each element in the vector—this isn't particularly useful in most cases. However, if applied to a list, it is much more useful. To demonstrate, let's make a list by splitting a data frame.

Working with the flow.dat data frame again:

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

First, let’s add a column to the data frame with actual names for these rivers instead of numbers (this is not necessary, but it makes interpretation a bit easier). There are multiple ways to do this—the following method uses indexing.

```
> flow.dat$name[flow.dat$site==1509000]<-"Tioughnioga"
> flow.dat$name[flow.dat$site==4232730]<-"Seneca"

> discharge.lst<-split(flow.dat$discharge,flow.dat$name)

> discharge.lst
$Seneca
  [1]   75  493 1380 1910 1940 1890 1860 1840 1850 1870 1850 1850 1830
 [14] 1940 1700 1800 1940 2140 2090 2220 2290 2270 2300 2360 2490 2560
```

```
 [27] 2400 2240 2260 2380 2370 2350 2360 2480 2500 1780  536  530  541
...

$Tioughnioga
  [1]  812  736  802  841  892  925  795  726  670  648  648 1170 1240
 [14] 2020 2360 1540 1200 2160 3170 2090 1660 1470 1220 1080  973  850
 [27]  740  720  720 1090 1270 1210 1030 1340 1730 1960 1720 1360 1140
...
```

Now let's use `lapply`. The help file gives the following information.

```
lapply(X, FUN, ...)

> lapply(discharge.lst,mean)
$Seneca
[1] NA

$Tioughnioga
[1] 701.0356
```

For some reason a mean cannot be calculated for the Seneca River—a likely cause is a missing value for discharge. To check:

```
> flow.dat[is.na(flow.dat$discharge),]
    agency    site       date discharge flag.discharge   name
273   USGS 4232730 2006-09-30        NA           <NA> Seneca
```

One way to handle this problem is to add the additional argument `na.rm=TRUE` to the end of the argument string. Any additional arguments are applied to the internal function, i.e. argument `FUN`. Note that only arguments that are options for the function `FUN` can be used in this way.

```
> lapply(discharge.lst,mean,na.rm=TRUE)
$Seneca
[1] 985.1923

$Tioughnioga
[1] 701.0356
```

The function `sapply` is similar to `lapply`, but simplifies the results.

```
> sapply(discharge.lst,mean,na.rm=TRUE)
     Seneca Tioughnioga
   985.1923    701.0356
```

These functions can be useful when working with lists, but if you are already working with a data frame and want to apply a function by the levels of some factor, the functions `tapply` and `aggregate` are easier to use. Both apply a specified function to subsets of data that comprise groups within the levels of a column (or columns) specified. The function `tapply` has the following arguments, as described in the associated help file:

```
tapply(X, INDEX, FUN = NULL, ..., simplify = TRUE)
```

As a simple example, say we want the mean and standard deviation of the discharge given in `flow.dat` for the individual sites. To apply the function `mean` by river name, we use:

```
> tapply(flow.dat$discharge,flow.dat$name,mean,na.rm=TRUE)
     Seneca Tioughnioga
   985.1923    701.0356
```

OK, but this isn't a particularly useful format, especially if we want to apply other functions as well, e.g. `sd`. We can use the function `data.frame` to combine results into a table (note that it is also possible to use `cbind` here).

```
>
data.frame(mean=tapply(flow.dat$discharge,flow.dat$name,mean,na.rm=TRUE),s
d=tapply(flow.dat$discharge,flow.dat$name,sd,na.rm=TRUE))
                mean       sd
Seneca      985.1923 729.0028
Tioughnioga 701.0356 671.4678
```

Although it is somewhat code-intensive, this is much more useful. If it is easier, you can always set up this procedure using multiple lines. If you need to use multiple `by` variables (i.e. the `INDEX` argument in `tapply`), use the function `aggregate` instead. Using `tapply` within a `data.frame` call, combined with writing data out using `write.table`, can be very handy for generating data summaries, which can then be pasted into, e.g. MS Word or Excel. Note that `mean` and `sd` are just examples of the many functions that can be used in this way (of course, you can even use your own functions here).

It is quite easy to write out data frames that can be pasted directly into Microsoft Word tables. To do this, just set the `sep` argument to `"\t"` for tabs. For example, using the above data frame:

```
> flow.summary<-
data.frame(mean=tapply(flow.dat$discharge,flow.dat$name,mean,na.rm=TRUE),s
d=tapply(flow.dat$discharge,flow.dat$name,sd,na.rm=TRUE))
> write.table(flow.summary,"flow_summary.out",sep="\t")
```

The result can now be directly pasted into a 2x3 section of a Microsoft Word table (the header is pasted separated or can be entered manually):

	Mean	sd
"Seneca"	985.192307692308	729.002776617018
"Tioughnioga"	701.035616438356	671.467792213991

Let’s clean this up a bit:

```
>write.table(signif(flow.summary,3),"flow_summary.out",sep="\t",quote=FALS
E)
```

	Mean	sd
Seneca	985	729
Tioughnioga	701	671

Let's read in a more complicated data set.

```
> tort.length.dat<-read.table("tortoise_length.txt",header=TRUE)
> names(tort.length.dat)
[1] "tortoise" "birth"    "age"      "length"   "s.length" "sex"

> tort.length.dat
    tortoise  birth  age   length s.length    sex
1          4 1970.5  4.5 24.50000       16   MALE
2          4 1970.5  5.5 26.50000       16   MALE
3          4 1970.5  6.5 34.80000       16   MALE
...

> tapply(tort.length.dat$age,tort.length.dat$tortoise,mean)
        1         2         3         4         5         6         9
14.500000 16.071429 17.357143 17.250000 16.625000 25.750000 14.409091
       11        12        13        14        15        16        17
19.500000 14.772727 15.666667 13.785714 16.342105 17.045455 15.666667
       18        20        21        22        28        31        32
...
```

There is a wrapper for the `tapply` called `by` that will apply `tapply` to multiple vectors or an entire data frame. This is a great way to generate summaries for visual inspection, but is not very useful for writing data out.

```
> by(tort.length.dat[,c(2,3,4,5)],tort.length.dat$tortoise, mean)
tort.length.dat$tortoise: 1
   birth      age   length s.length
  1970.5     14.5     60.5      5.0
-------------------------------------------------------
tort.length.dat$tortoise: 2
     birth        age     length   s.length
1970.50000   16.07143   63.07143    7.00000
-------------------------------------------------------
tort.length.dat$tortoise: 3
     birth        age     length   s.length
```

```
1970.50000   17.35714   64.48571    7.00000
...
```

The above functions are useful for applying a function to specific groups within a data set and providing the 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. For example, perhaps you want to normalize the tortoise lengths to the mean of each individual. There are a couple code-intensive ways that you could do this: create a summary with e.g. `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`.

```
> ave(tort.length.dat$length, tort.length.dat$tortoise, FUN=mean)
  [1] 67.13750 67.13750 67.13750 67.13750 67.13750 67.13750 67.13750
  [8] 67.13750 67.13750 67.13750 67.13750 67.13750 67.13750 67.13750
 [15] 67.13750 67.13750 56.68182 56.68182 56.68182 56.68182 56.68182
...
```

We can add this result to the original data frame.

```
> tort.length.dat$length.m<-ave(tort.length.dat$length,
tort.length.dat$tortoise, FUN=mean)
> tort.length.dat
    tortoise  birth  age   length s.length    sex length.m
1          4 1970.5  4.5 24.50000       16   MALE 67.13750
2          4 1970.5  5.5 26.50000       16   MALE 67.13750
3          4 1970.5  6.5 34.80000       16   MALE 67.13750
...
17        12 1970.5  5.5 33.90000       12   MALE 56.68182
18        12 1970.5  6.5 39.60000       12   MALE 56.68182
19        12 1970.5  7.5 46.80000       12   MALE 56.68182
...
```

If we want to use these means to normalize length, we need to write our own function, which is covered in a later section. (However note that in this case, it actually would have been easier to just use vector arithmetic).

```
> tort.length.dat$length.norm<-ave(tort.length.dat$length,
tort.length.dat$tortoise, FUN=function(x)(x/mean(x)))

> tort.length.dat
    tortoise  birth  age   length s.length    sex length.m length.norm
1          4 1970.5  4.5 24.50000       16   MALE 67.13750   0.3649227
2          4 1970.5  5.5 26.50000       16   MALE 67.13750   0.3947123
3          4 1970.5  6.5 34.80000       16   MALE 67.13750   0.5183392
...
```

Exercises

1. 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.

2. Read in the data in the file Thakali_Cu_EC50s.txt. Create a subset of this data set for soils with pH > 4, and sort the resulting data frame by the Cu EC50. Write out the resulting data frame to a file. Try eliminating the quotes around the soil names in the output file.

3. Calculate your age (as of today) in days. How about in seconds or in weeks?

4. 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.

7. Exploratory data analysis

Dalgaard 2008: Chapter 4

7.1. Summary statistics

We have already covered how to calculate means and standard deviations. R has other useful functions for summarizing data. Let's demonstrate 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`. What it returns is dependent on the type of data submitted to it. 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
```

Notice the difference between numerical and categorical variables.

7.2. 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 & Helsell 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 required for `ros` are shown below, as given in its help file.

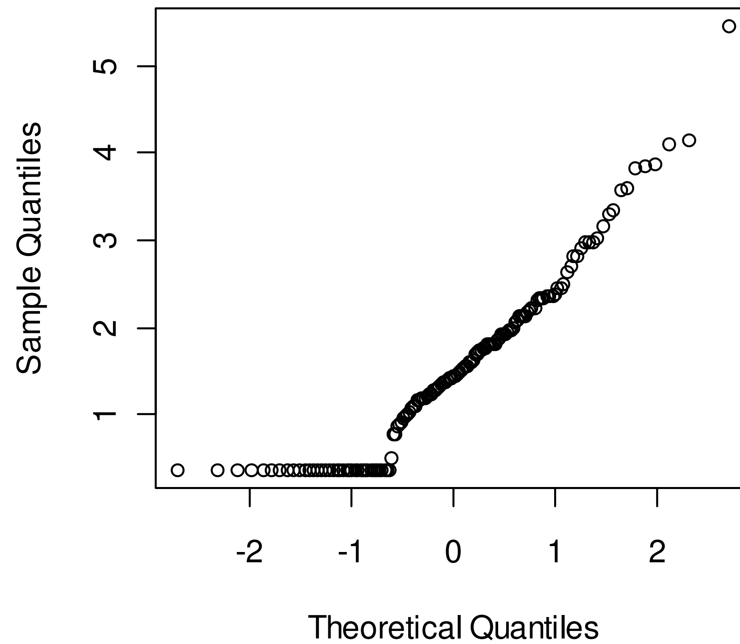
```
ros(obs, censored, forwardT="log", reverseT="exp")
```

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
```

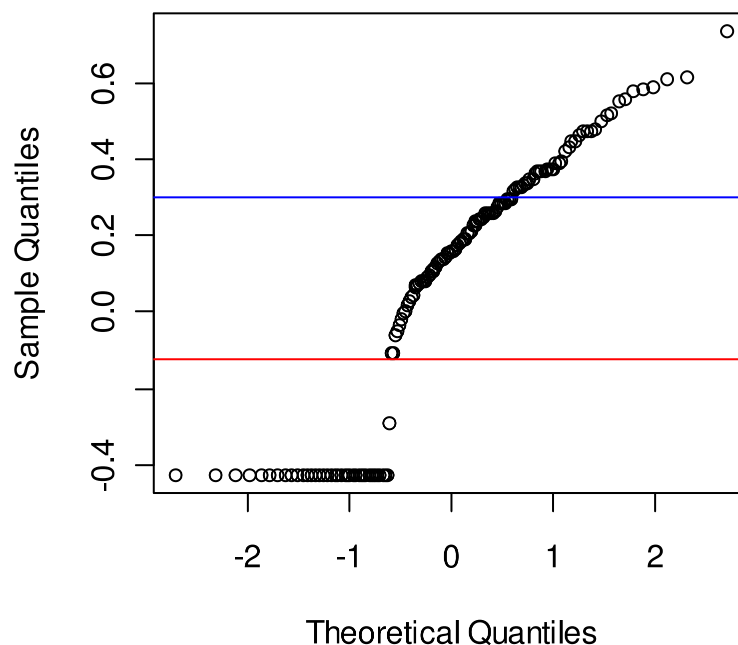
The detection limit in this study was 0.75 ng/g, but the limit of quantification was 2.0 ng/g. In this data set, nondetects are set to ½ of the detection limit. Let's see what these data look like to start out.

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



These data look surprisingly close to normally-dstributed. Of course, that is impossible for concentrations. 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$conc<0.75]<-T
> achlor.dat$conc[achlor.dat$conc<0.75]<-0.75
> achlor.dat[1:20,]
   conc nondetect
1  1.25     FALSE
2  1.55     FALSE
3  1.74     FALSE
4  1.77     FALSE
5  1.82     FALSE
6  2.33     FALSE
7  2.36     FALSE
8  2.49     FALSE
9  3.57     FALSE
10 3.88     FALSE
11 0.75      TRUE
12 1.21     FALSE
13 1.92     FALSE
14 2.22     FALSE
15 0.75      TRUE
```

```
16 0.75      TRUE
17 1.28     FALSE
18 1.54     FALSE
19 1.89     FALSE
20 2.33     FALSE
...

> ros(achlor.dat$conc,achlor.dat$nondetect)
          n       n.cen      median        mean
147.0000000  40.0000000   1.4400000   1.6350897
         sd
  0.9199055
```

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 `as.data.frame` function.

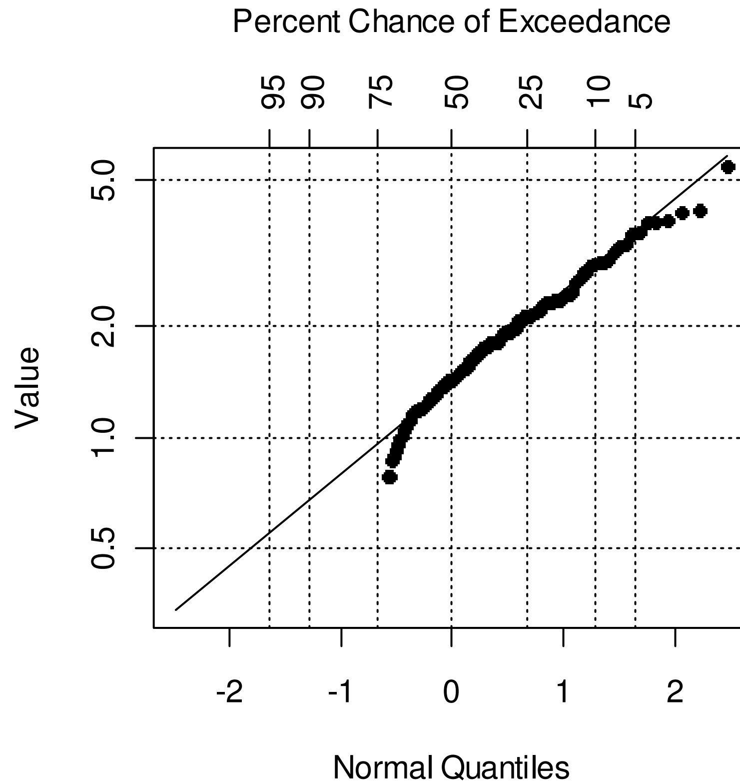
```
> achlor.ros<-ros(achlor.dat$conc,achlor.dat$nondetect)
> achlor.est<-data.frame(achlor.ros)
> achlor.est[1:10,]
    obs censored          pp   modeled
1  0.75     TRUE 0.006636801 0.3397521
2  0.75     TRUE 0.013273602 0.3939582
3  0.75     TRUE 0.019910403 0.4324218
4  0.75     TRUE 0.026547204 0.4635999
5  0.75     TRUE 0.033184005 0.4904518
6  0.75     TRUE 0.039820806 0.5143984
7  0.75     TRUE 0.046457607 0.5362435
8  0.75     TRUE 0.053094408 0.5564906
9  0.75     TRUE 0.059731210 0.5754783
10 0.75     TRUE 0.066368011 0.5934465
...
```

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] 1.404101
```

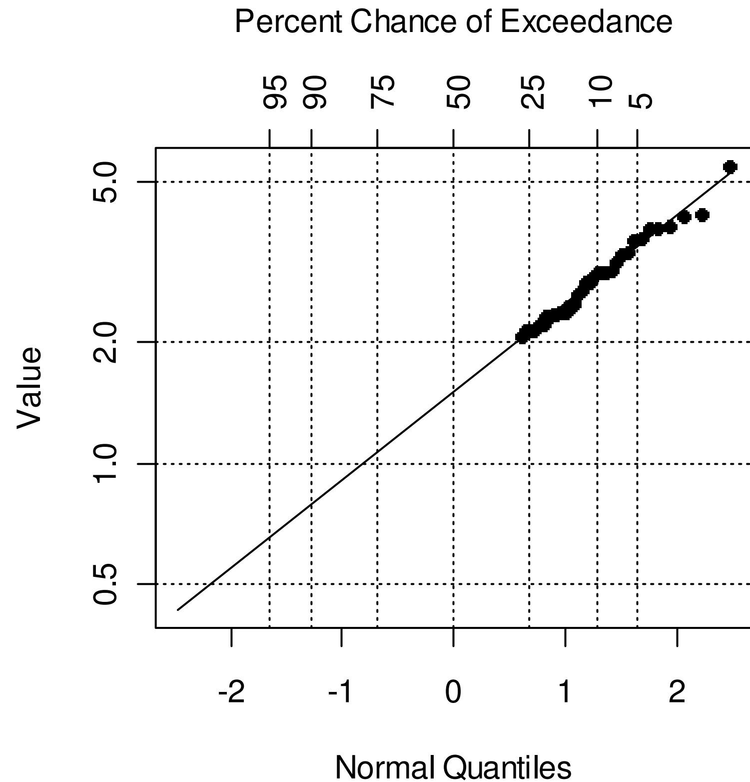
By applying the plot function to `ros` output, you can see the estimated distribution and get a visual representation of the `ros` process.

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



Let's repeat this, but use the quantification limit instead, which may be a better approach.

```
> achlor.dat<-data.frame(conc=eagles.dat$achlor,nondetect=F)
> achlor.dat$nondetect[achlor.dat$conc<2]<-T
> achlor.dat$conc[achlor.dat$conc<2]<-2
>
> ros(achlor.dat$conc,achlor.dat$nondetect)
         n      n.cen     median       mean         sd
147.000000 106.000000   1.503688   1.698922   0.870380
> achlor.ros<-ros(achlor.dat$conc,achlor.dat$nondetect)
> achlor.est<-data.frame(achlor.ros)
> 10^mean(log10(achlor.est$modeled))
[1] 1.506938

> plot(achlor.ros)
```



Notice the similarity in the results, even though we are now modeling more than two-thirds of the data.

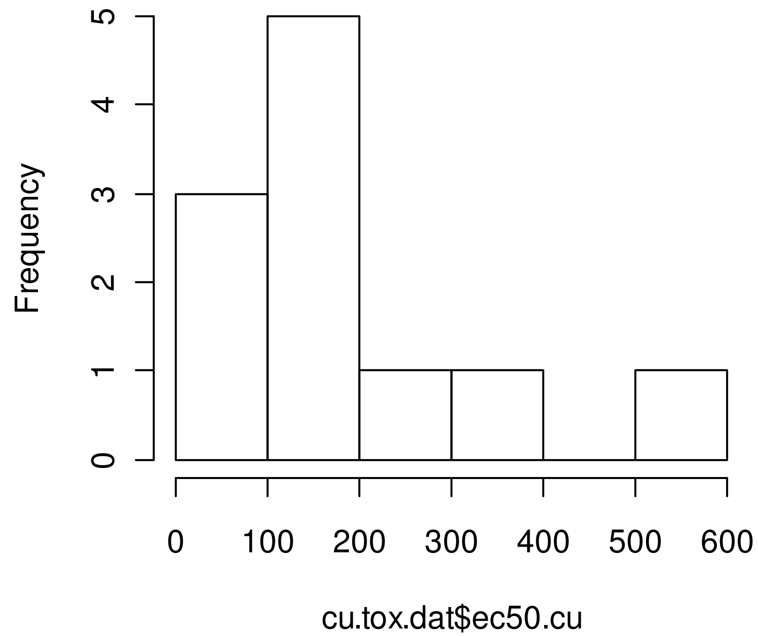
7.3. Histograms, box plots, and probability plots

R has functions that can be used to generate common types of plots. Some of these are demonstrated below using one of the soils data sets.

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

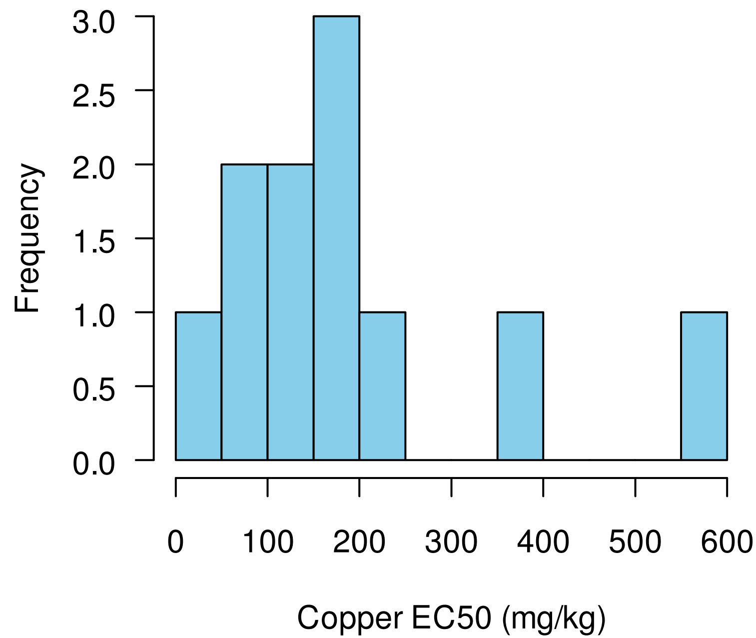
You can generate a histogram in R using the function `hist`.

```
> 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.

```
> hist(cu.tox.dat$ec50.cu,breaks=8,col="skyblue",xlab="Copper EC50
(mg/kg)",las=1,main="Copper toxicity to barley in soil")
```



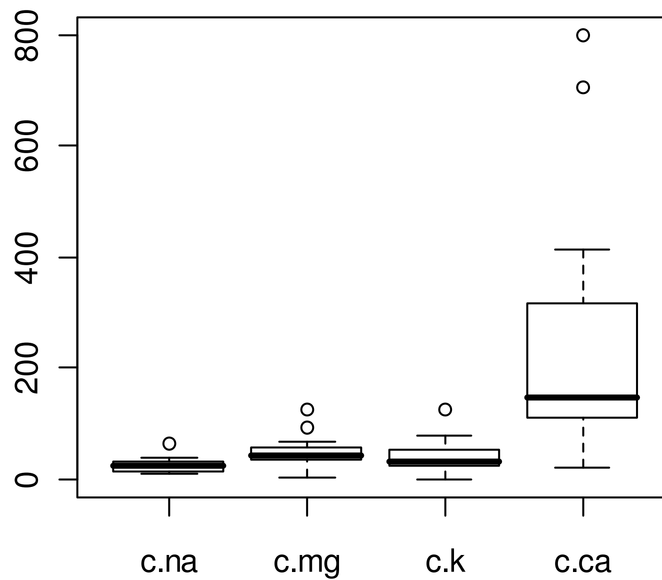
Boxplots are also a good way to summarize data. In the following example, a boxplot is used to compare concentrations of several cations in the soils.

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

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

And then make the plot.

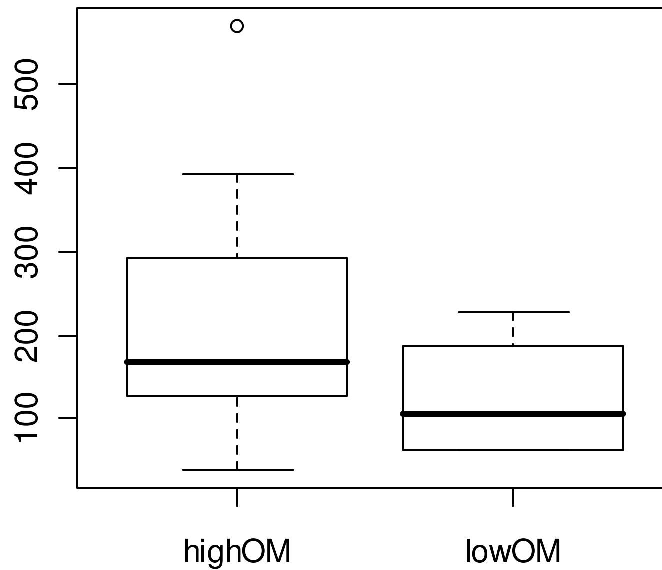
```
> boxplot(cu.tox.dat[,7:10])
```



The help file for `boxplot.stats` has a description of what the various symbols mean.

A more common approach would be to plot a single variable by some factor. For example, let's divide the soils data set into two groups: low and high organic matter:

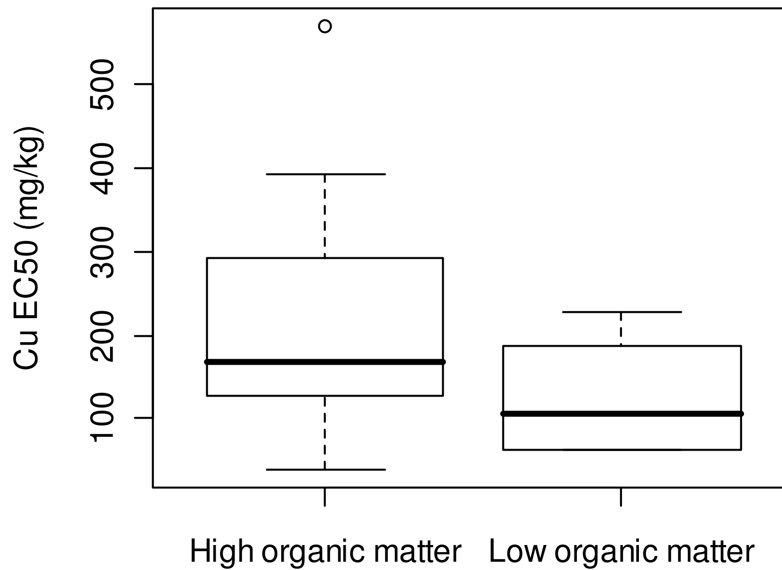
```
> cu.tox.dat$group[cu.tox.dat$oc<1]<-"lowOM"
> cu.tox.dat$group[cu.tox.dat$oc>=1]<-"highOM"
> boxplot(ec50.cu~group,data=cu.tox.dat)
```



Some other options are shown in the following command.

```
> boxplot(ec50.cu ~ group,data=cu.tox.dat,names=c("High organic
matter","Low organic matter"),ylab="Cu EC50 (mg/kg)")
```

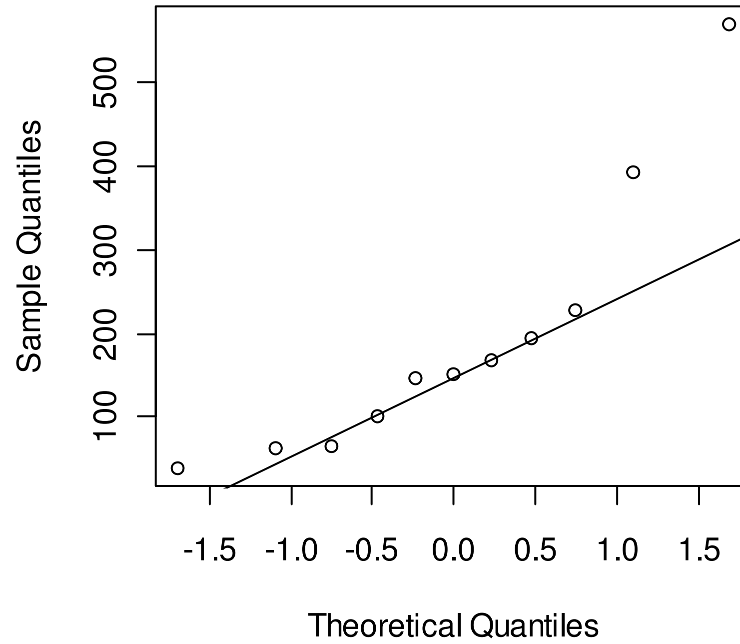
Note the use of the tilde symbol "~" in the above command. The code `ec50.cu ~ group` is considered a model formula in this case, and simply indicates that `ec50.cu` is described by `group` and should be split up based on the value of the variable `group`. We will see more of this character with the specification of statistical models.



7.4. Normal quantile plots and cumulative probability plots (Crawley 2007: 281)

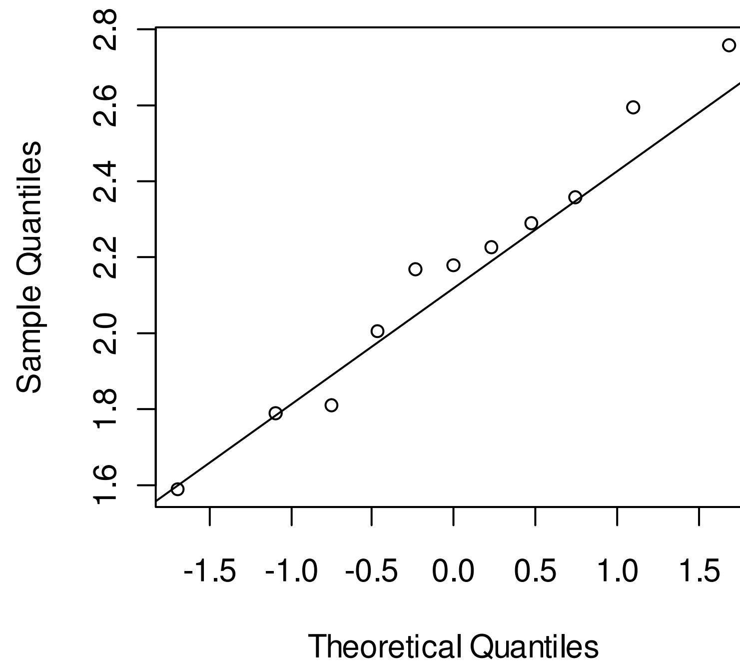
One way to assess the normality of the distribution of a given variable is with a quantile-quantile plot. This plot plots data values vs. quantiles based on a normal distribution (i.e. a z distribution).

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



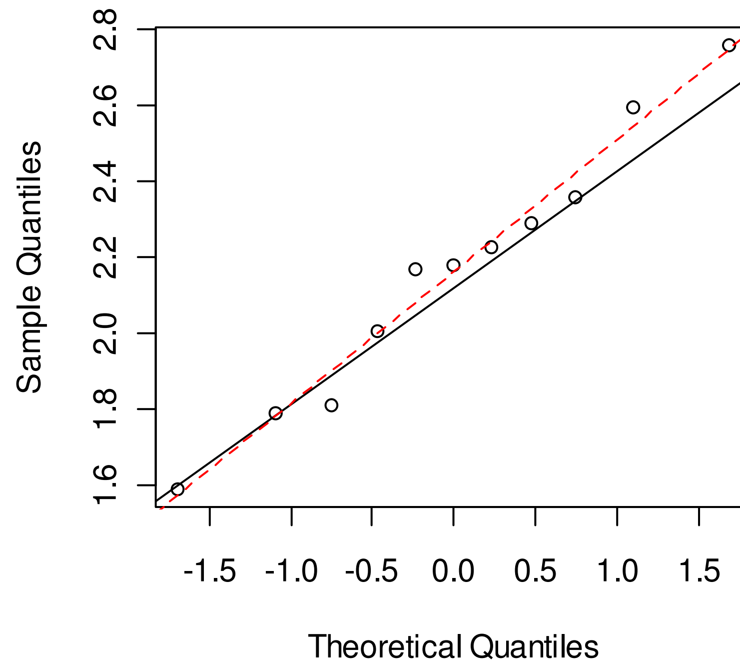
There definitely seems to be some deviation from normality here. A common distribution for toxicity data is log-normal.

```
> 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)
```



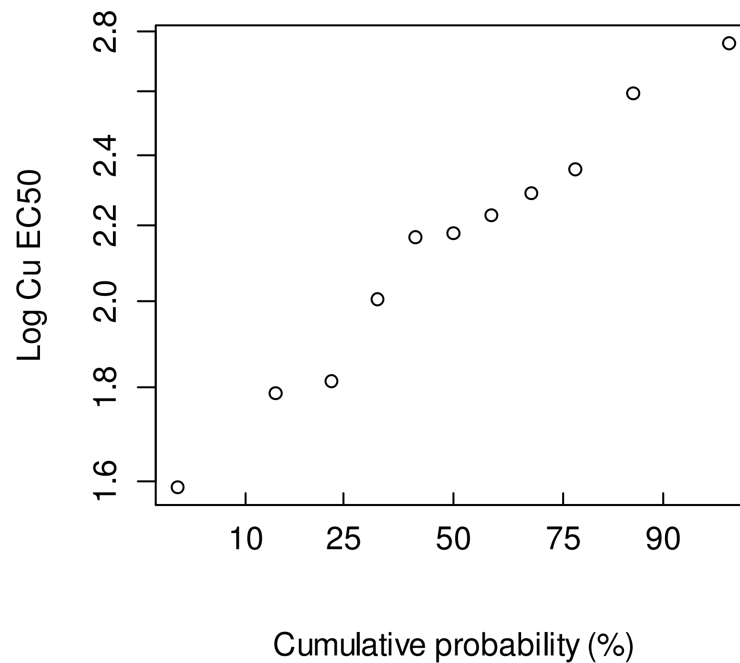
Normal quantile plots can be a bit tricky to understand. Here are some results that might make them more clear:

```
> x<-sort(cu.tox.dat$l.ec50)
> probs<-ppoints(x)
> probs
 [1] 0.04545455 0.13636364 0.22727273 0.31818182 0.40909091 0.50000000
 [7] 0.59090909 0.68181818 0.77272727 0.86363636 0.95454545
> qs<-qnorm(probs)
> qs
 [1] -1.6906216 -1.0968036 -0.7478586 -0.4727891 -0.2298841  0.0000000
 [7]  0.2298841  0.4727891  0.7478586  1.0968036  1.6906216
> slopes<-diff(x)/diff(qs)
> slopes
 [1] 0.33141770 0.07113896 0.70695940 0.66760581 0.04060199 0.21678724
 [7] 0.24877745 0.25219275 0.67728540 0.27407058
> sd(x)
[1] 0.3482742
> qqnorm(x)
> qqline(x)
> abline(median(x),sd(x),col="red",lty=2)
```



The cumulative probability plot (as least the one described below) is really a type of normal quantile plot. 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(0.01,0.1,0.25,0.5,0.75,0.9,0.99)),labels=c(1,10,25,50,75,90
,99),las=1,tcl=0.3,mgp=c(0,0.2,0))
```



R will return the quantiles of a given data set via 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

> mean(cu.tox.dat$l.ec50)
[1] 2.160197

> median(cu.tox.dat$l.ec50)
[1] 2.177536
```

Exercises

1. Install and load the `ISwR` package. Check out the help file for the `InsectSprays` data frame. 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

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.

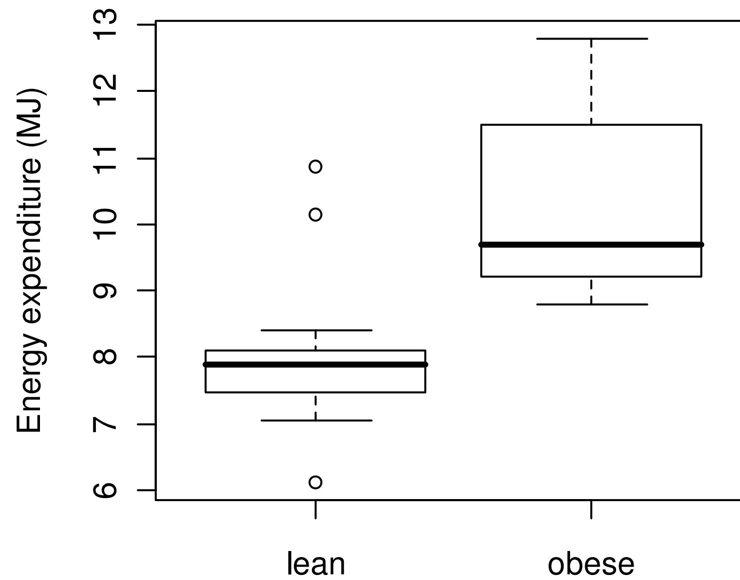
```
> DO.dat<-read.table("DO_methods_1.txt",header=TRUE)

> 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, ylab="Energy expenditure
(MJ)")
```




```
> 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"
"null.value"
[7] "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

9.1. The lm function, model formulas, and more on 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 main structure of the statistical model. Venables et al. (2006: 50-51) has a useful list of example formulae. Some examples are repeated below. 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 + C + A:B + A:C + B:C`	Two-factor ANOVA with secondary interaction term
`y ~ (A + B + C)^2`	Two-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.

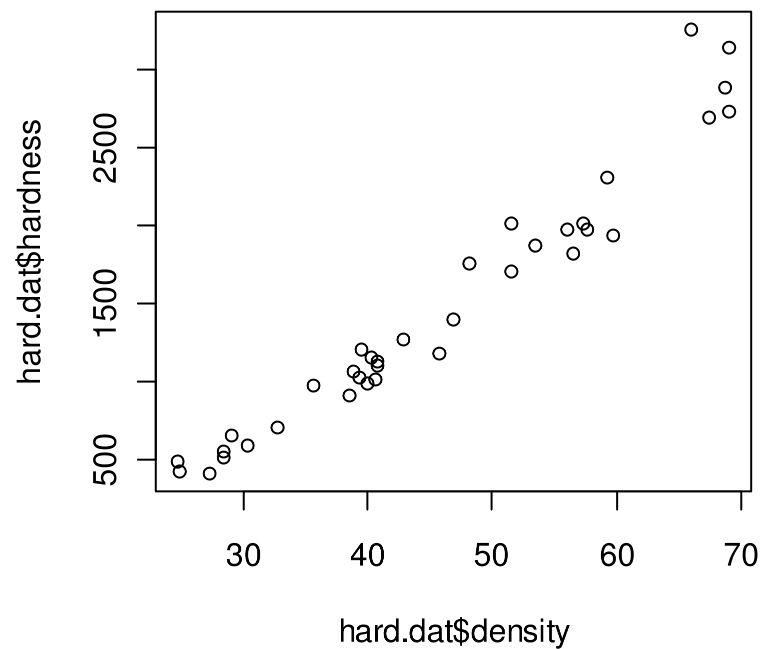
9.2. Linear regression

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)

> names(hard.dat)
[1] "density"  "hardness"

> 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 goal here is to see whether density is a good predictor of hardness, and if so, to develop a predictive model for hardness. Let's start out by seeing what the data look like.

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



This looks like a pretty clear relationship. To fit a linear model, 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
```

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. Let's take a look at some of the options R has for dealing with linear model output.

R has several extractor functions for pulling data out of linear models. 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
          6           7           8           9          10
 140.8061355   5.0474583 -15.9685611  92.2620821 -139.5072748
         11          12          13          14          15
  -0.7592772 -79.5126146 104.7367180 -145.0166194   2.9807107
         16          17          18          19          20
-164.2712918 -80.0219592 -50.0219592 -36.5366437 -293.3060005
         21          22          23          24          25
-136.5633428 148.6779800 -91.0940467 208.9059533 -30.3567287
         26          27          28          29          30
 -79.8740831 -268.6274205 -114.6327603 -171.8847628  66.1045576
```

```
          31           32           33           34           35
-338.3994472  625.0591692  -15.4501754   94.0404799  -73.2115225
          36
 326.7884775
```

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"
```

To get at these results, you can simply index the `lm` object, i.e. call up part of the list.

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

> mod1$coef
(Intercept)     density
-1160.49970    57.50667
```

This makes for some redundancy in R. For many model statistics, there are three ways to get your data: an extractor function, indexing the `lm` object, and the `summary` function. Here is a list of what's available from the `summary` function for this model:

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

> 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

> summary(mod.1)[[4]][1,1]
[1] -1160.500

> summary(mod.1)[["coefficients"]][1,1]
[1] -1160.500
```

In general, indexing the `lm` object is the simplest way to extract model data. However, not all options that you might want are available this way.

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 a 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 sums of squares (SS). (Note that this is different from the default approach used in other statistical software. Search the R mailing lists for type III sum of squares or something similar for a lot of discussion on this topic. More discussion follows in the section on ANOVA below.)

```
> 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
 ...
       29        30        31        32        33        34        35
2151.8848 2243.8954 2278.3994 2634.9408 2715.4502 2795.9595 2813.2115
       36
2813.2115
```

This function works for a whole range of statistical models in R—not just `lm` objects. We can do anything with these predictions—add them to the above plot, put them back in the original data frame. This function can also give confidence and prediction intervals.

```
> predict(mod.1,int="conf")
         fit       lwr       upr
1   259.9152  144.4580  375.3724
```

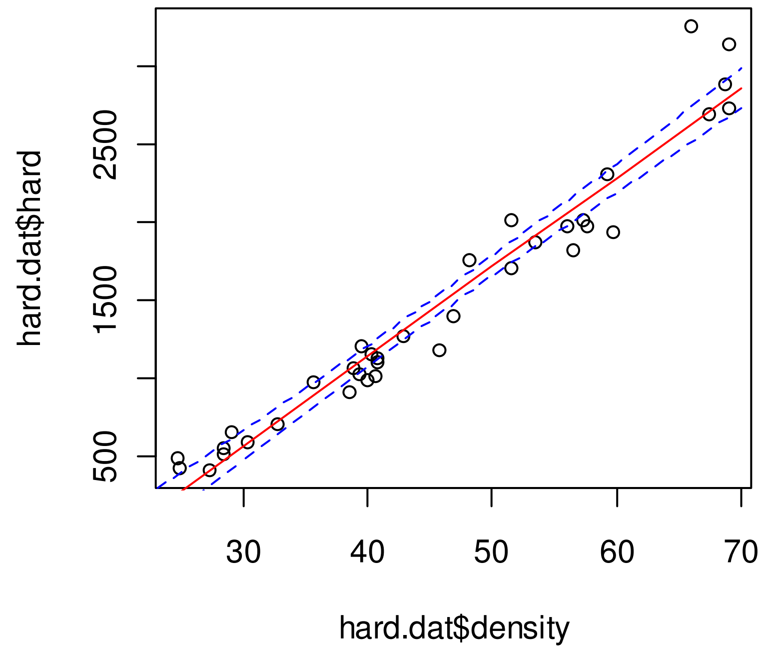
```
2   265.6658  150.5990  380.7327
3   409.4325  303.9330  514.9320
4   472.6899  371.2673  574.1125
...
```

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
...
```

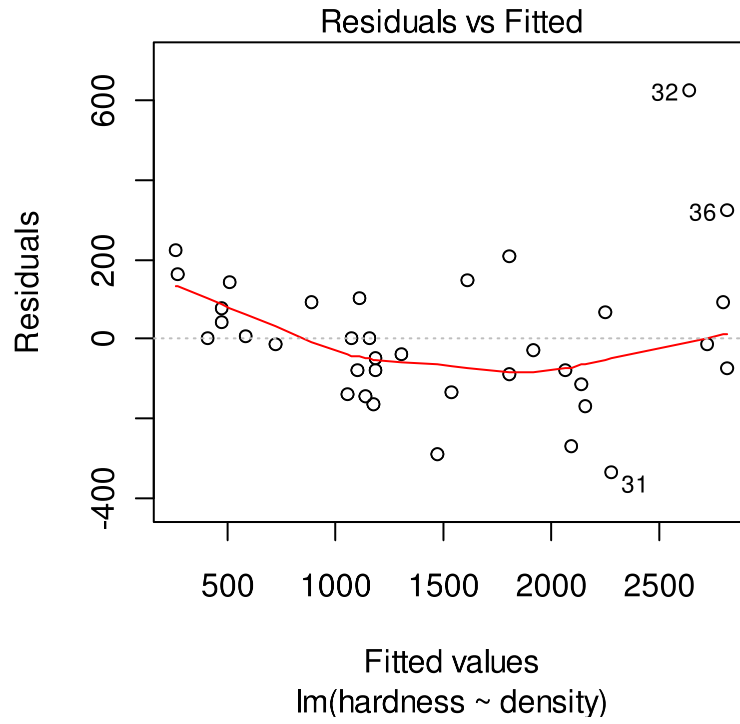
OK, so what is the best way to add these predictions? The function `matlines` will plot the columns in a matrix as a function of a single vector. This works well for plotting confidence intervals and predictions with a single line of code. (Note that the below code does not use the data frame `hard.pred.dat`,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

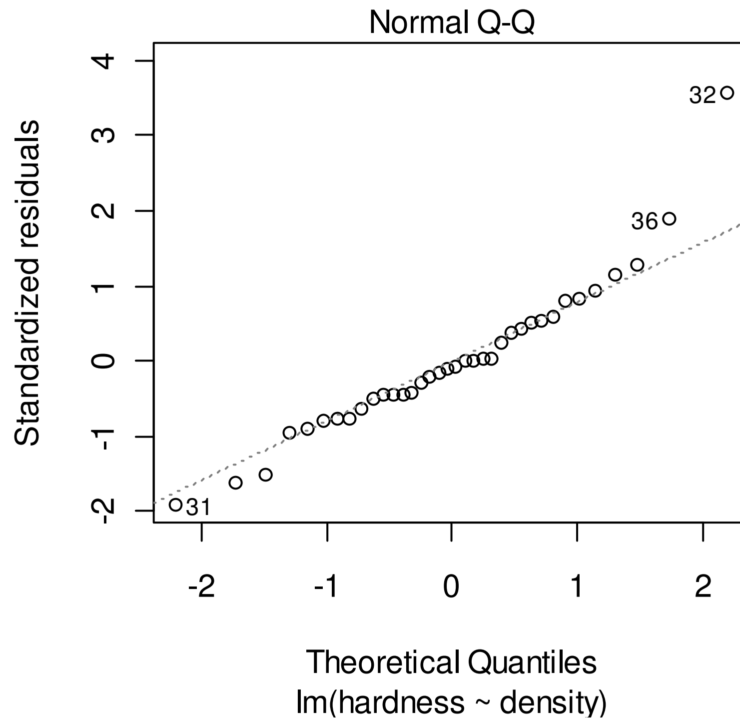
```
> plot(hard.dat$density,hard.dat$hard)
> matlines(density,conf.int,lty=c(1,2,2), col=c("red","blue","blue"))
```

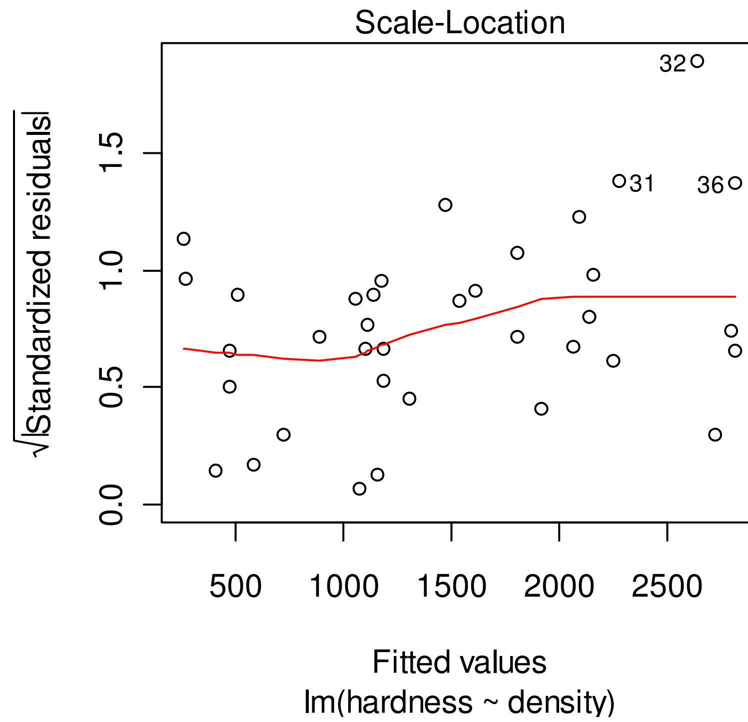


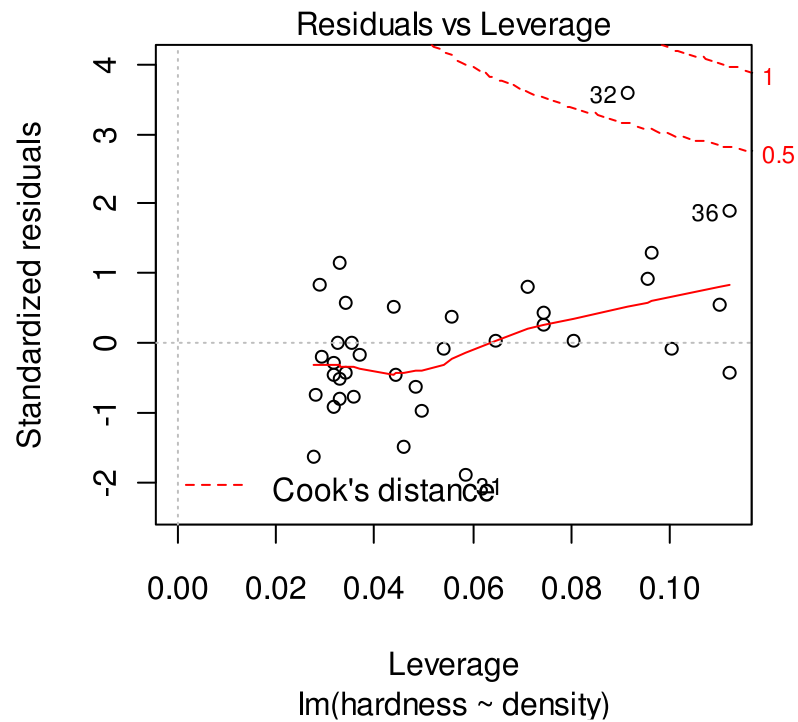
If we wanted to store the predictions in a data frame, we could have done that in one step.

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 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|
Fitted values
lm(hardness ~ density)
32
31
36
0.0
0.5
1.0
1.5
500
1000
1500
2000
2500


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

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

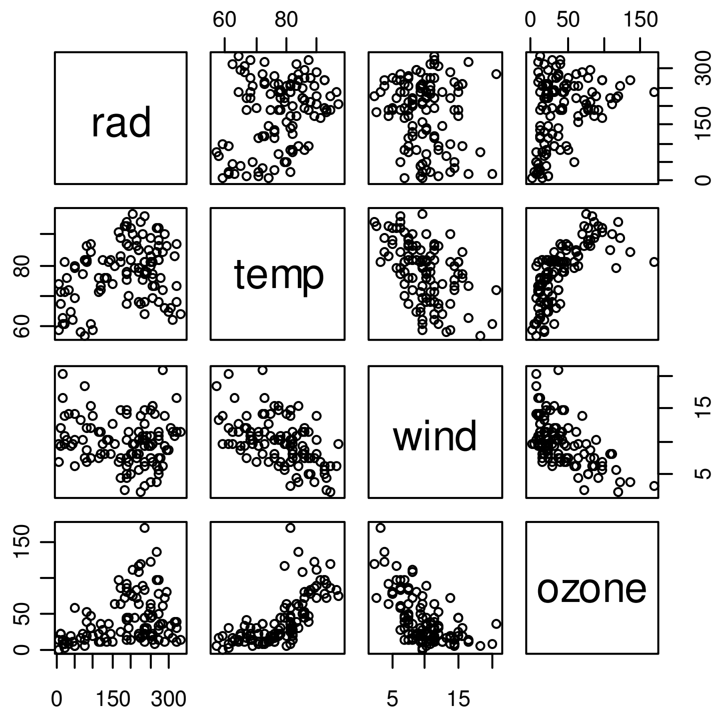
```
> ozone.dat<-read.table("Ozone.txt",header=T)

> 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 at relationships between variables 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)
```



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).

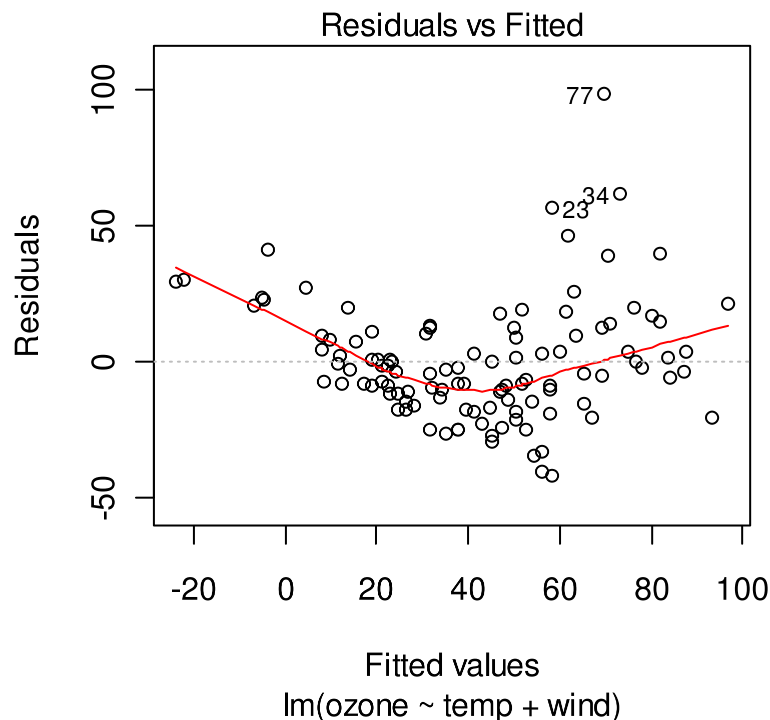
```
> mod.2<-lm(ozone~temp + wind, data = ozone.dat)
> summary(mod.2)

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
```

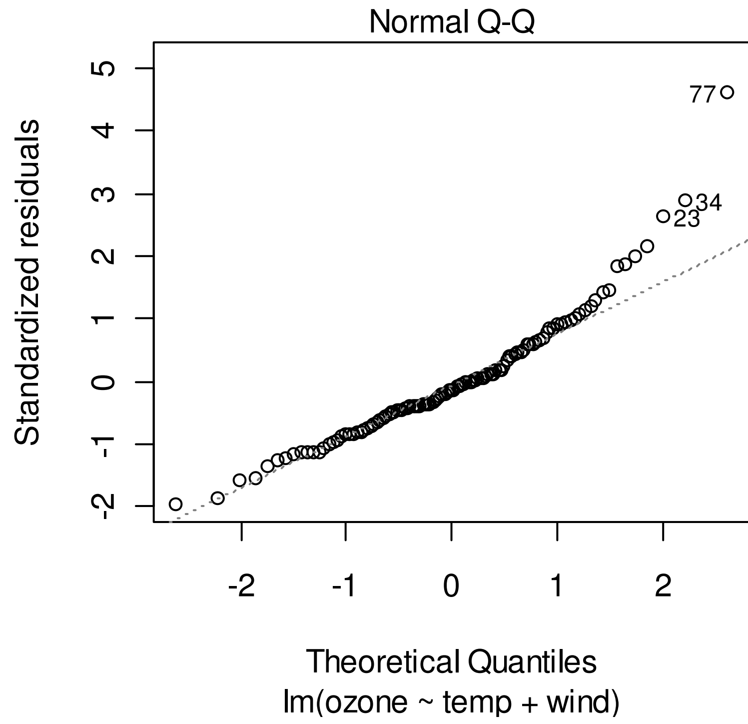
We could also use the `update` function for model 2—this is especially handy for dealing with large model formulas.

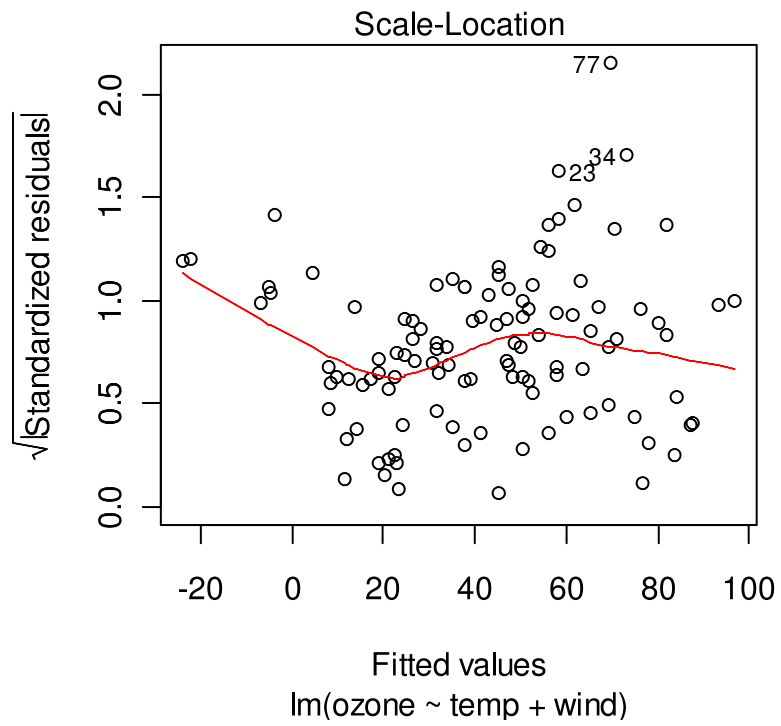
```
> mod.2<-update(mod.1, ~. -rad)
> summary(mod.2)

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
```

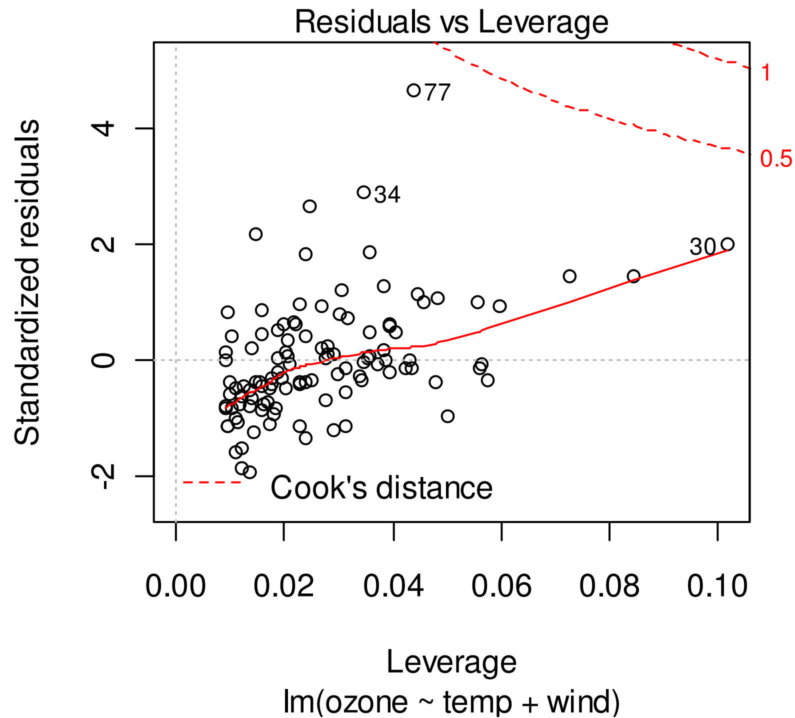
Let's take a look at the results.

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




Normal Q-Q
Standardized residuals
Theoretical Quantiles
lm(ozone ~ temp + wind)
77
34
23

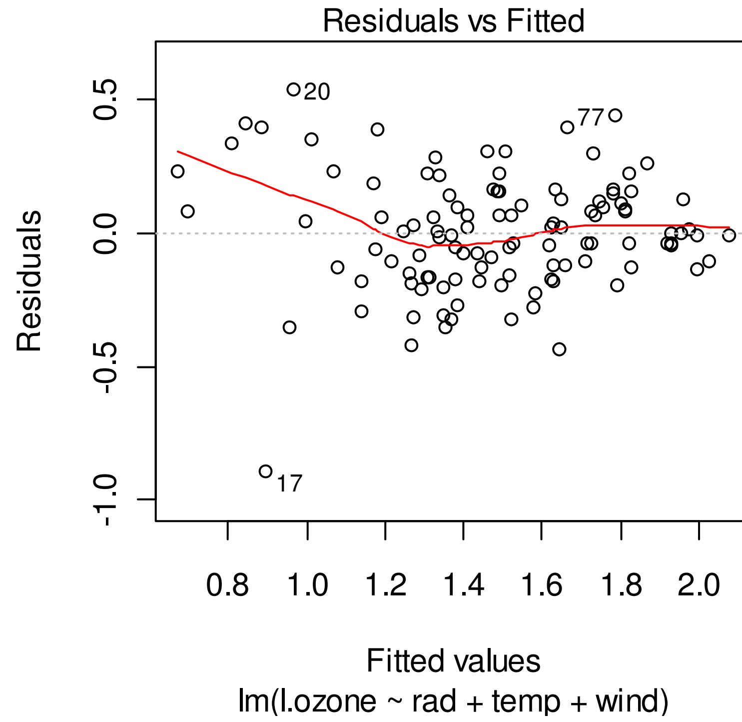

Scale-Location
√|Standardized residuals|
Fitted values
lm(ozone ~ temp + wind)
77
34
23

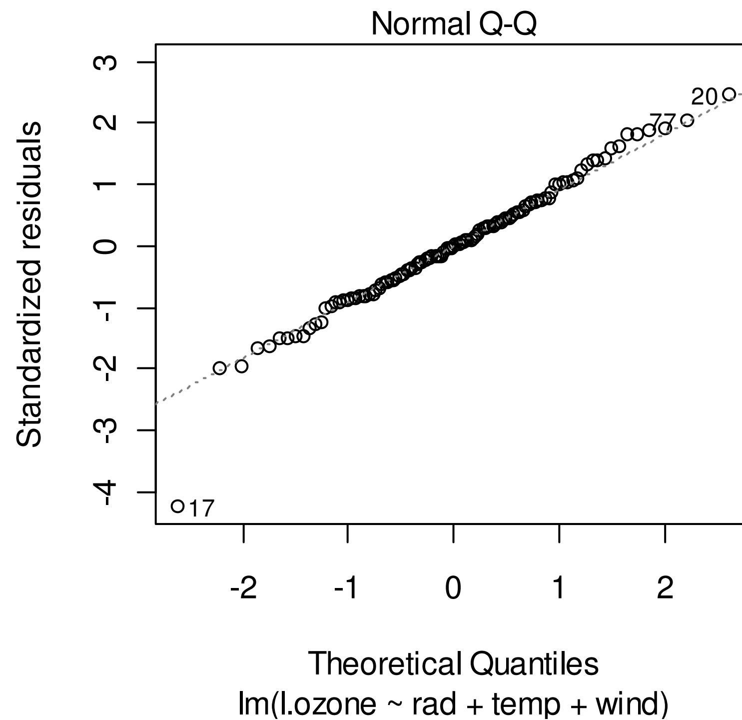


It looks like the residuals are not normally distributed, and there seems to be a relationship to the fiitted values. Since we are attempting to model concentration data, we might consider log-transforming the values.

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

Call:
lm(formula = l.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 ***
---
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.

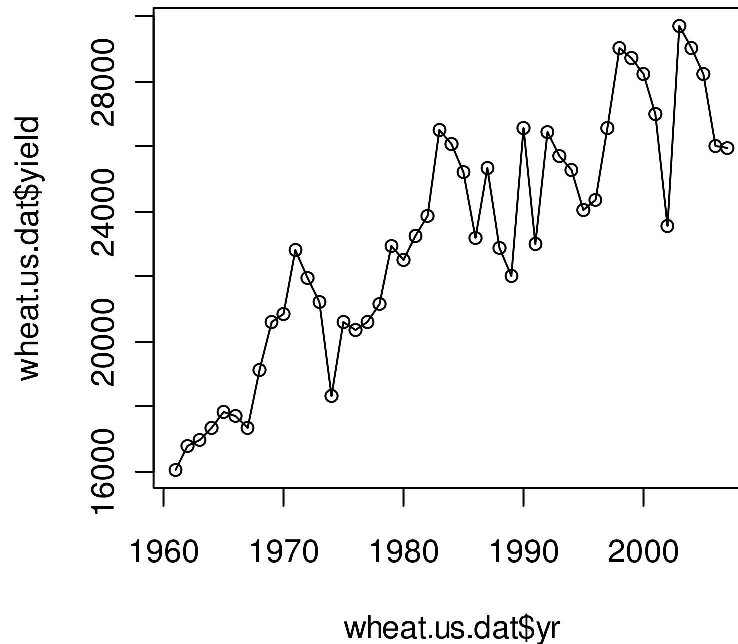




Our residuals look better now.

Polynomial regression can be carried out in R using the `lm` function.

```
> 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")
```

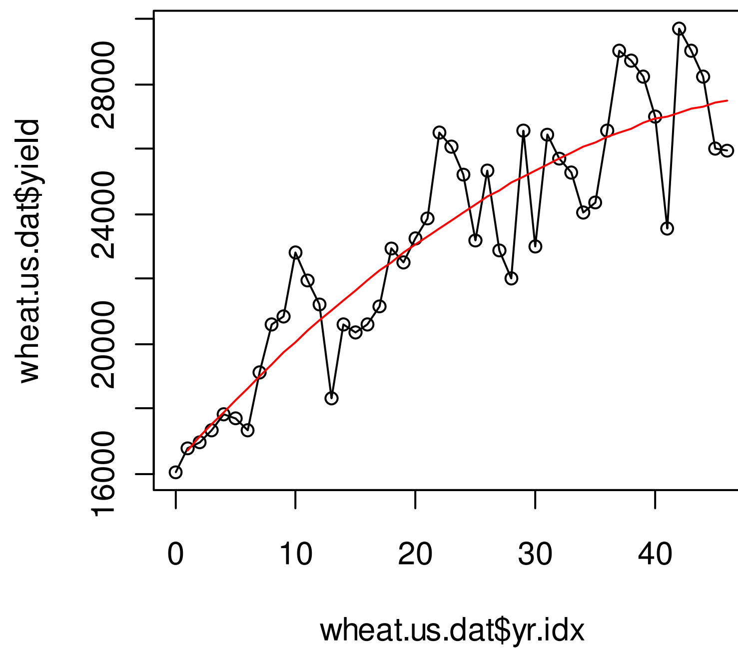



```
> mod.1<-lm(yield~yr.idx + I(yr.idx^2), data=wheat.us.dat)
> summary(mod.1)

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=1:46)
> wheat.pred.dat$yield.pred<-predict(mod.2,newdata=wheat.pred.dat)
> 
points(wheat.pred.dat$yr.idx,wheat.pred.dat$yield.pred,type="l",col="red")
```



9.3. ANOVA and pairwise comparisons

To demonstrate ANOVA in R, we will start out 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 of course, not normally distributed. To make these data approximate a normal distribution, we can use a square root transformation (Zar 1999).

```
> 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)
> mod.2

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

Coefficients:
(Intercept)       sprayB       sprayC       sprayD       sprayE
sprayF
     3.8115       0.1143      -2.3549      -1.5587      -1.8937
0.2550

> 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 ***
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

> 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
```

To demonstrate the extraction of some results from a statistical test:

```
> mod.2$coefficients
(Intercept)      sprayB      sprayC      sprayD      sprayE      sprayF
  3.8114910   0.1143102  -2.3549121  -1.5587119  -1.8937416   0.2549576
```

```
> mod.2$residuals
          1           2           3           4           5           6
-0.59046633 -1.09579589  0.70237651 -0.02005329 -0.02005329 -0.29367919
...
```

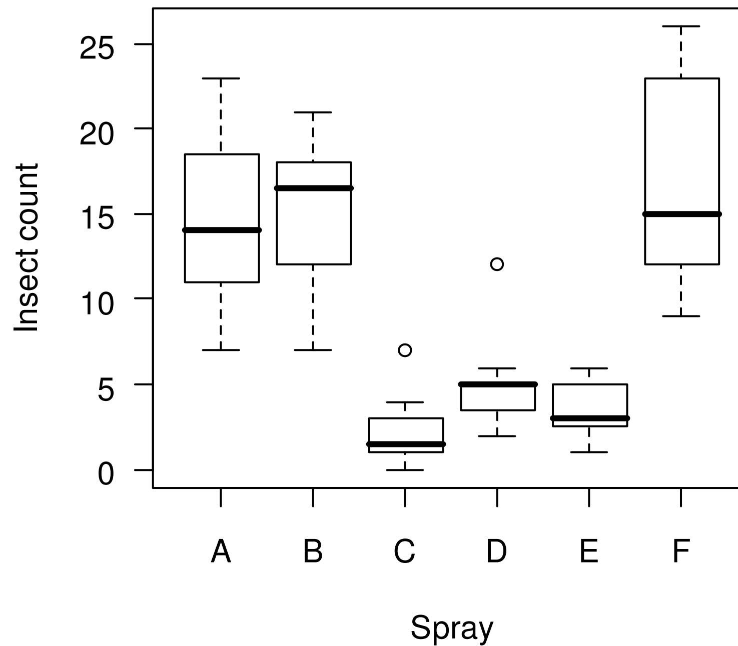
R has many multiple range tests available, including Tukey's HSD test in the base package, and many others in the `multcomp` package. The Tukey test is applied using the `TukeyHSD` function. Note that this function seems to require `aov` output (`lm` will not work).

```
> 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)
```



We will demonstrate barplots in a later section.

A two-factor or multi-factor ANOVA is carried out in a similar way. Let's use some data from Zar (199) on the respiration rate of 3 species of crabs in response to temperature..

```
> 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      sp:sex    temp:sex
         24          24          36           8          12          12
sp:temp:sex
          4
```

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

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)
            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.

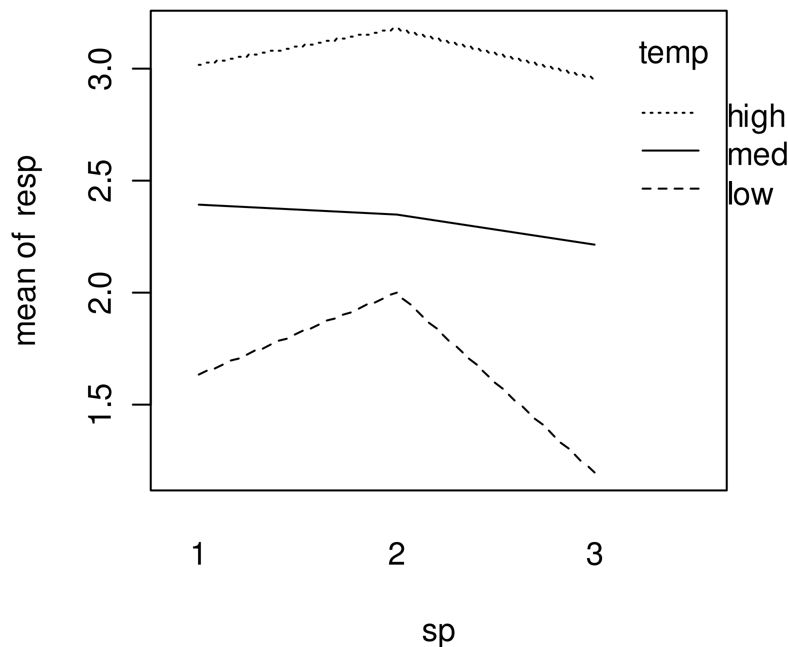
```
> mod.3<-aov(resp~(sp+temp+sex)^2,data=crabs.dat)
> summary(mod.3)
            Df  Sum Sq Mean Sq  F value    Pr(>F)
sp           2  1.8175  0.9088  23.6829 3.028e-08 ***
temp         2 24.6558 12.3279 321.2767 < 2.2e-16 ***
sex          1  0.0089  0.0089   0.2317   0.63211
sp:temp      4  1.1017  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
```

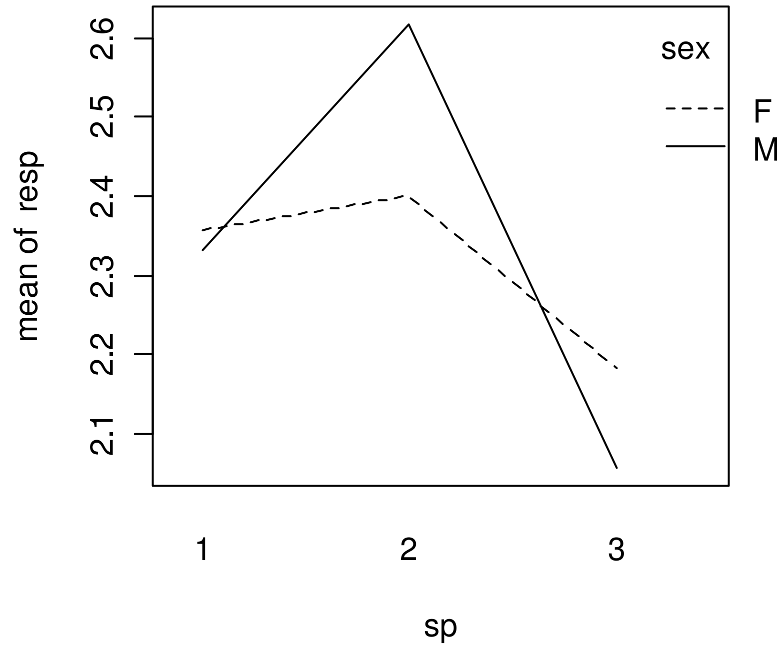
Don't forget that the ANOVA table that you get with `lm` objects in R returns tests based on Type I SS. If you have an unbalanced design, there are other functions available for carrying out ANOVA with Type II SS, e.g `Anova` in the `car` package. Alternatively, you can use the `anova` function with nested models for specific deletion tests.

So, our analysis shows that respiration differs with species and temperature, and that the effect of temperature is depdendent 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?

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

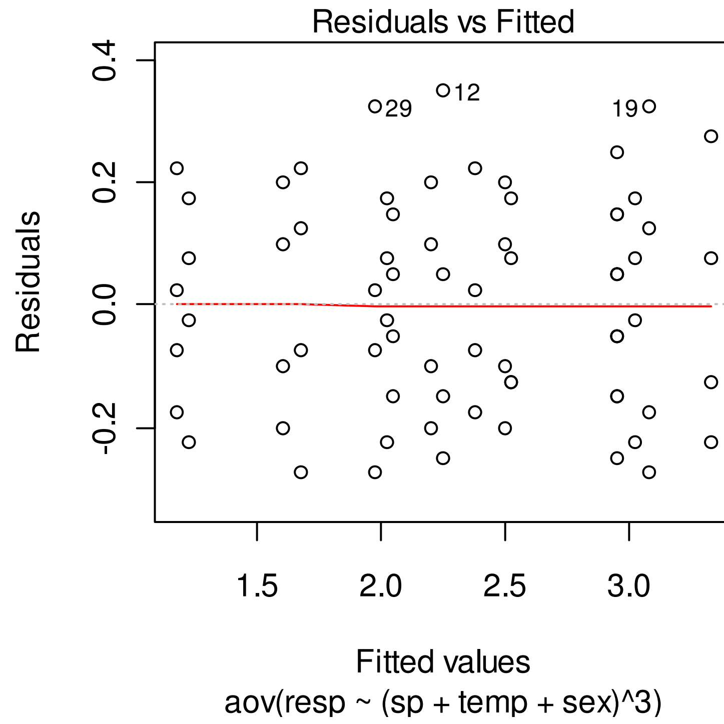


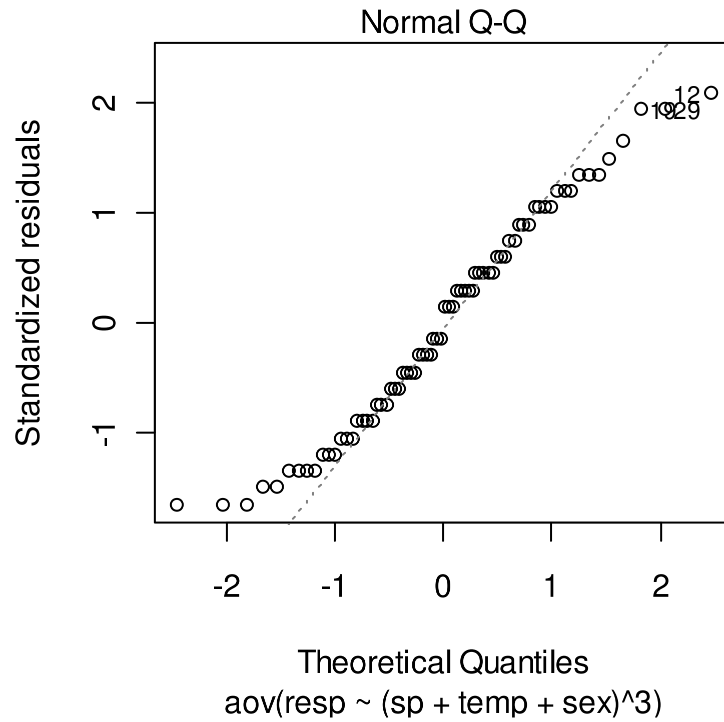
```
> with(crabs.dat,interaction.plot(sp,sex,resp))
```

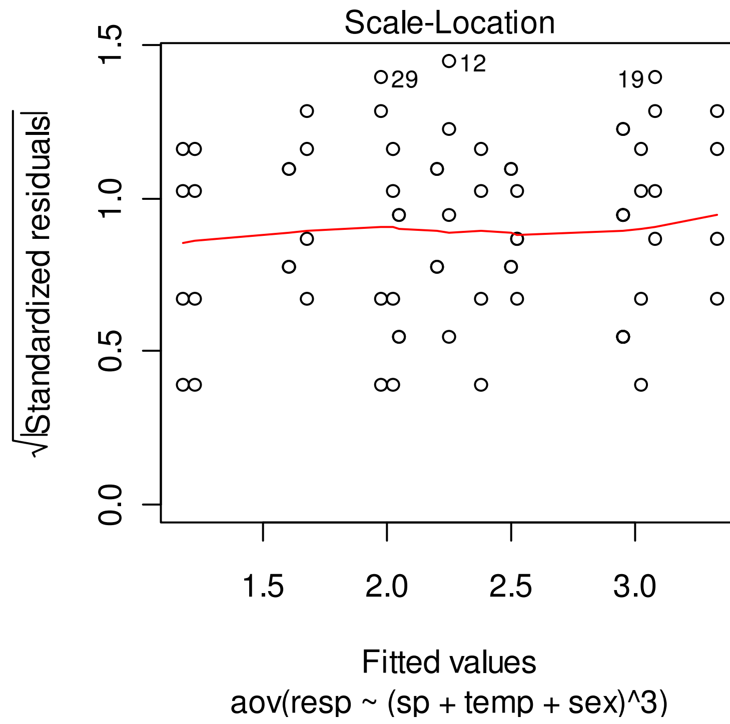


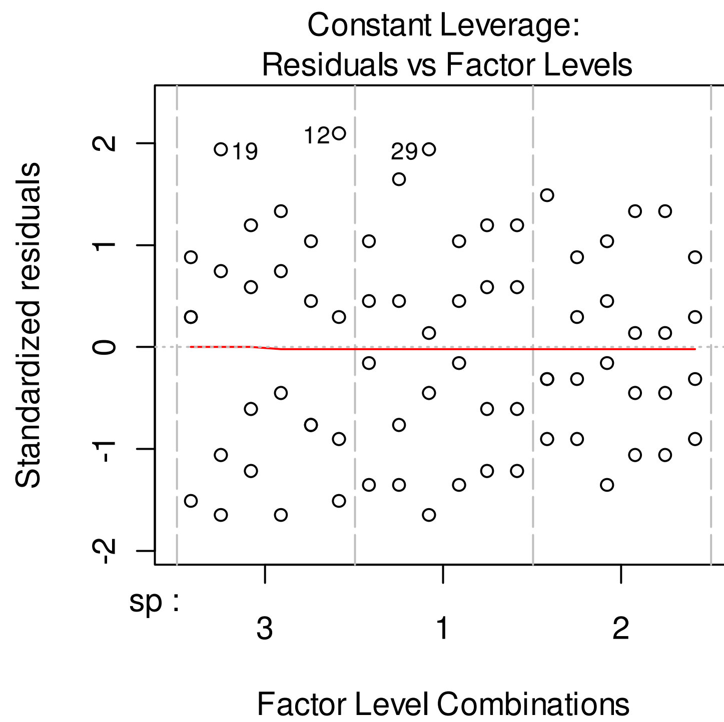
This second interaction plot shows why we don't see a significant main effect for sex (although this test is irrelevant, considering that the interaction is significant). We can also apply the `plot` function to the model output.

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


Residuals vs Fitted
Residuals
0.4
0.2
0.0
-0.2
29
12
19
1.5
2.0
2.5
3.0
Fitted values
aov(resp ~ (sp + temp + sex)^3)


Normal Q-Q
Standardized residuals
2
1
0
-1
12
1929
-2
-1
0
1
2
Theoretical Quantiles
aov(resp ~ (sp + temp + sex)^3)


Scale-Location
√|Standardized residuals|
Fitted values
aov(resp ~ (sp + temp + sex)^3)
29
12
19
1.5
1.0
0.5
0.0
1.5
2.0
2.5
3.0


Constant Leverage:
Residuals vs Factor Levels
Standardized residuals
Factor Level Combinations
sp :
3
1
2
19
12
29
2
1
0
-1
-2

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 a linear function of 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)
> 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
```

Note that `mod.2$coef` would have worked as well. Of course, you can extract specific coefficients as well:

```
> coef(mod.2)["ph"]
       ph
0.6920473

> coef(mod.2)[9]
       ph
```

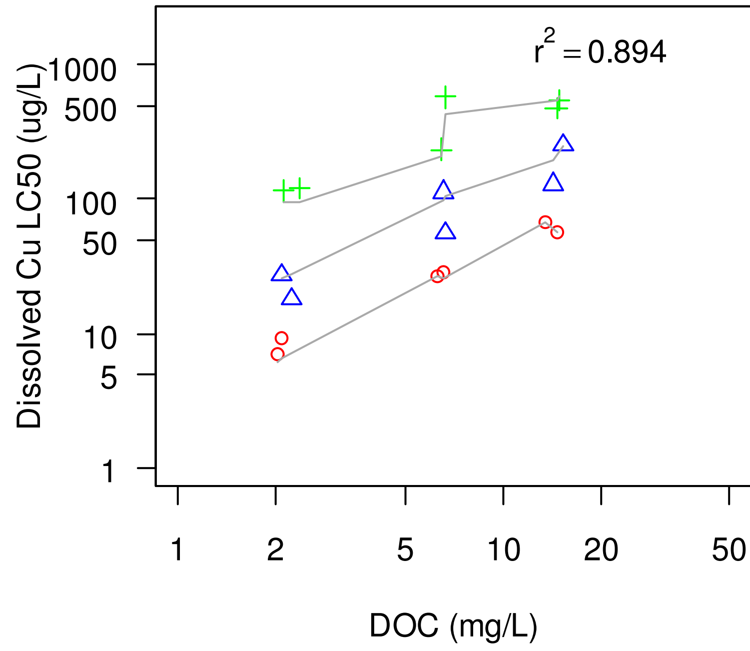
```
0.6920473
```

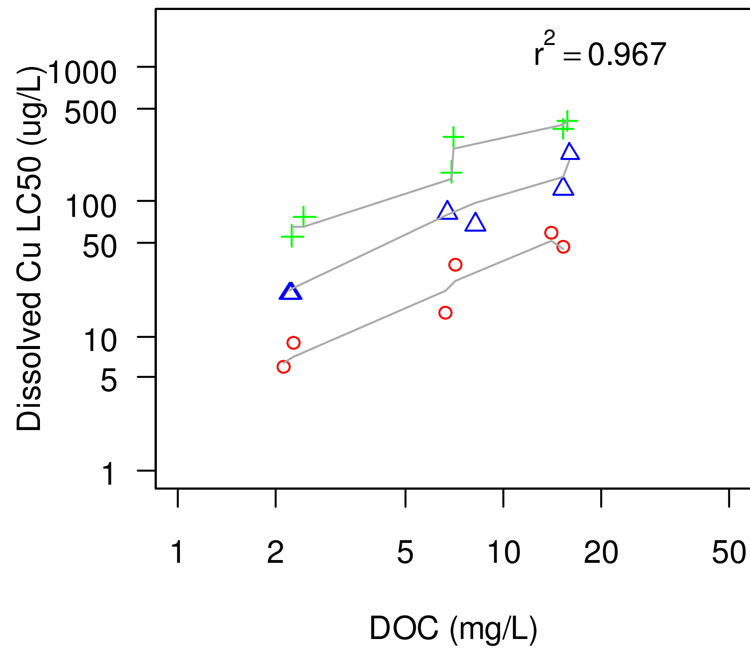
Using R's `plot` function along with some others, it is possible to very clearly display factorial data and model fit. Simply applying the `plot` function to ANOVA or ANCOVA output will give you some useful plots as we have seen with other `lm` objects.

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

...
```

Here is some more advanced code for plotting these results. Two 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$n.ph)
n.colors<-c("red","blue","green")
par(ask=T)
for(i in levels(lc50.dat$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="Dissolved Cu LC50 (ug/L)",main=paste("DOC source =",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=="Og1") 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),b
ty="n")
	text(20,1500,substitute(r^2==x,list(x=signif(cor(sub.2$lc50.pred,sub.2$lc50)^2,
3))))
}
```


DOC source = 1
$r^2 = 0.894$
Dissolved Cu LC50 (ug/L)
1000
500
100
50
10
5
1
1
2
5
10
20
50
DOC (mg/L)


DOC source = 5
$r^2 = 0.967$
Dissolved Cu LC50 (ug/L)
1000
500
100
50
10
5
1
1
2
5
10
20
50
DOC (mg/L)

Exercises

1. Read in the data in the file Daphnids.txt, which contains LC50s for Daphnia under various water chemistry conditions (Ryan in press). Analyze these data by multiple linear regression to determine which variables appear to have an effect on LC50. Plot predicted vs. observed LC50s and add confidence intervals.

2. 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 analogs 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)
```

Excercises

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

11. Graphics II

Murrell 2006, Crawley 2007, Dalgaard 2008

11.1 Arranging multiple plots per page

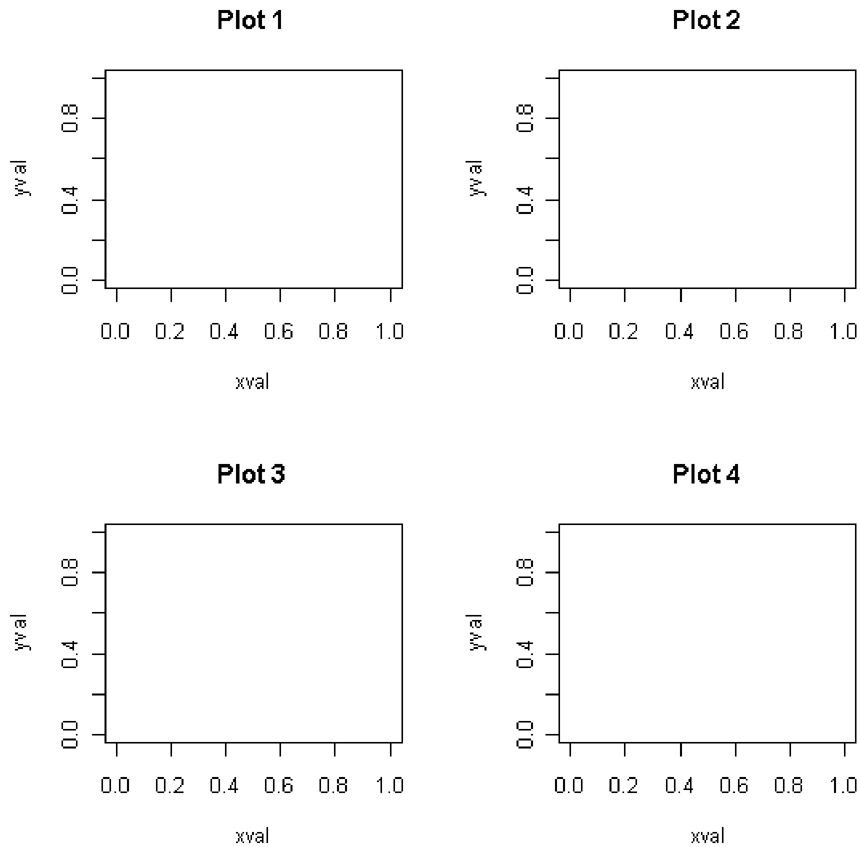
It is convenient to discuss the topic of arranging multiple plots per page here, because we will be demonstrating various arguments to the plot function—to keep track of changes, it is helpful to see plots side-by-side. There are two common ways of putting multiple plots on one page. One way is to modify the graphic state setting `mfrow` (or equivalently, `mfcol`), with the `par` function. This is a very convenient, and easy to understand 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. (You can find more flexibility still in the `lattice` package, which is not covered in this workshop.) Before we start describing each of these methods, it is useful to view the current graphic state setting, and since we have not yet modified them, these are the defaults. Query the settings with `par()`. A long list of settings is the result—only a few are shown below.

```
> par()
$xlog
[1] FALSE

$ylog
[1] FALSE

...

$mfcol
[1] 1 1

$mfg
[1] 1 1 1 1

$mfrow
[1] 1 1

...

$yaxt
[1] "s"
```

Note that the default settings for `mfcol` and `mfrow` are `1,1`. This means that plots are arranged on a page in a one row-by-one column fashion. In other words, only one plot is shown on a page. To create a plot layout that consists of four equal sized plot regions, `mfrow` (or `mfcol`) would be set to `2,2` (e.g. `mfrow=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. This is best demonstrated with an example:

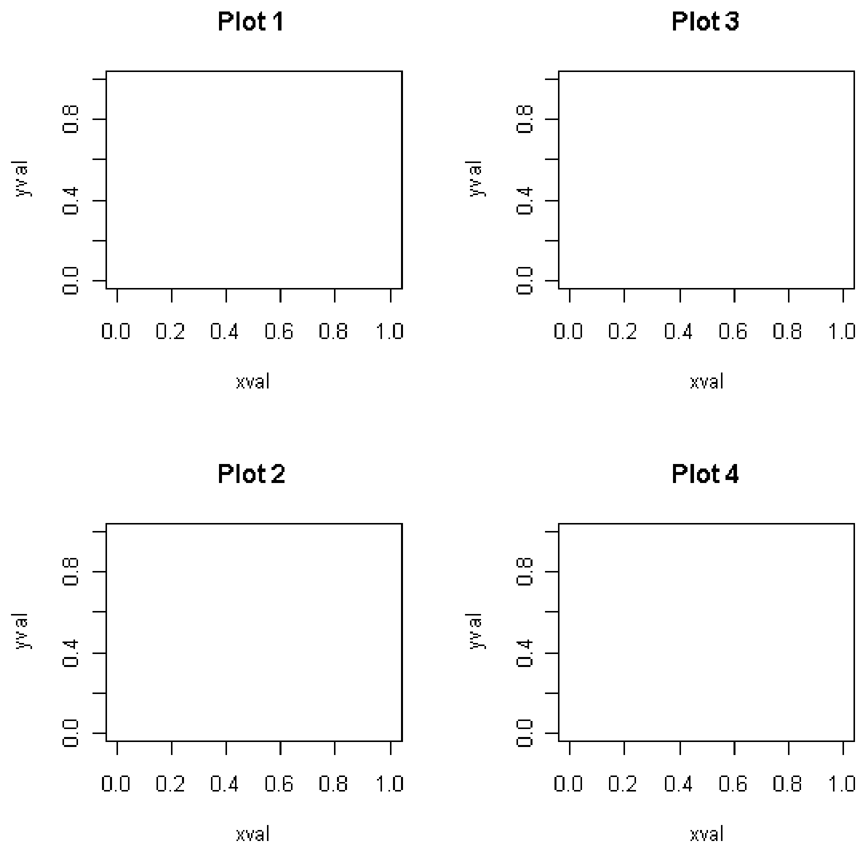
```
> xval=c(0,1)
> yval=c(0,1)
```

```
> par(mfrow=c(2,2))
> plot(x=xval,y=yval,type="n",main="Plot 1")
> plot(x=xval,y=yval,type="n",main="Plot 2")
> plot(x=xval,y=yval,type="n",main="Plot 3")
> plot(x=xval,y=yval,type="n",main="Plot 4")
```



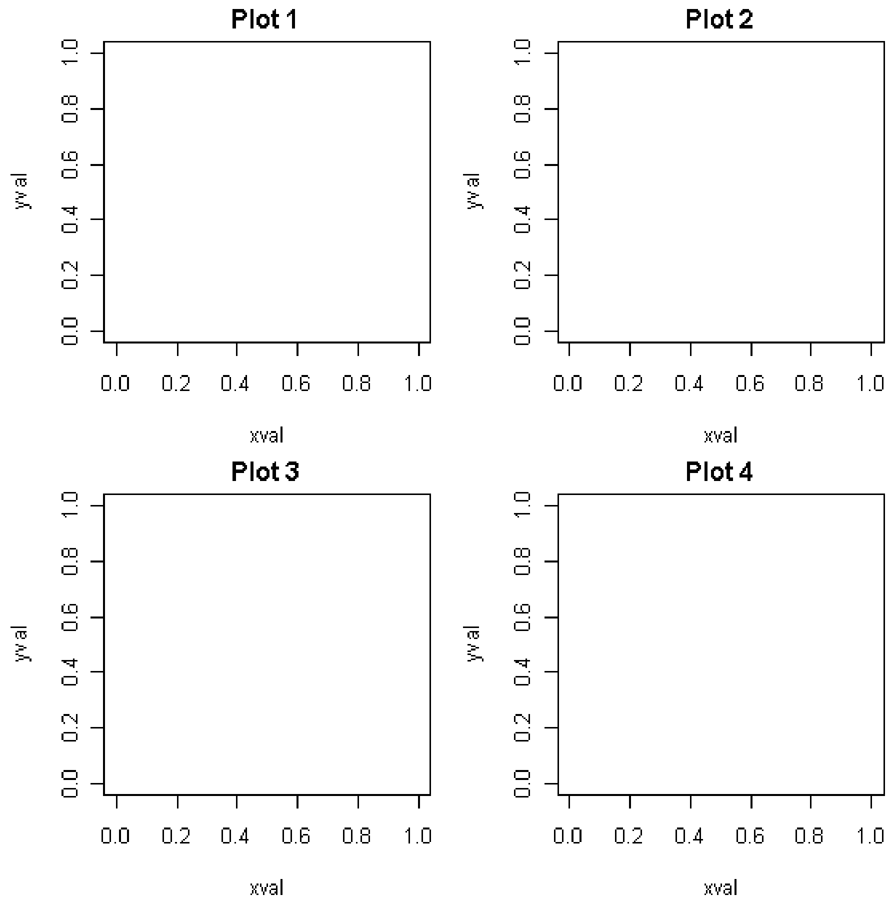
Notice the order of plots as they are arranged on the page. The plotting starts with the upper left region of the page, fills out the top row, continues to the second row, and eventually ends in the bottom right region of the page. If another plot had been specified, it would have been placed in the upper left region of a new page. With `mfcol`, the plots are arranged differently.

```
> par(mfcol=c(2,2))
> plot(x=xval,y=yval,type="n",main="Plot 1")
> plot(x=xval,y=yval,type="n",main="Plot 2")
> plot(x=xval,y=yval,type="n",main="Plot 3")
> plot(x=xval,y=yval,type="n",main="Plot 4")
```



If 4 plots per page were desired, but the size shown above is not pleasing, you can change the settings of the plot margin with `mar`. The default for `mar` is `mar=c(5.1,4.1,4.1,2.1)`. Each of the elements in the `mar` vector represents the number of lines from one side of the plot region to the outer margin—the reading order here is bottom, left, top, right. The outer margins can be set in a similar fashion with `oma`, but that will not be covered here. Below is an example of decreasing the figure margins to make the plot regions bigger.

```
> par(mfrow=c(2,2),mar=c(4,4,2,1))
> plot(x=xval,y=yval,type="n",main="Plot 1")
> plot(x=xval,y=yval,type="n",main="Plot 2")
> plot(x=xval,y=yval,type="n",main="Plot 3")
> plot(x=xval,y=yval,type="n",main="Plot 4")
```



More flexibility is allowed with the `layout` function because plot regions with different sizes can be specified. Here we will first show how to create the example above, and then we will show how to create a 3-panel plot, with one panel twice the width of the other two. First, reset the graphical output to show a singe plot, and prove that only one plot is displayed by typing a plot command:

```
> par(mfrow=c(1,1))
> plot(x=xval,y=yval,type="n",main="Plot 4")
```

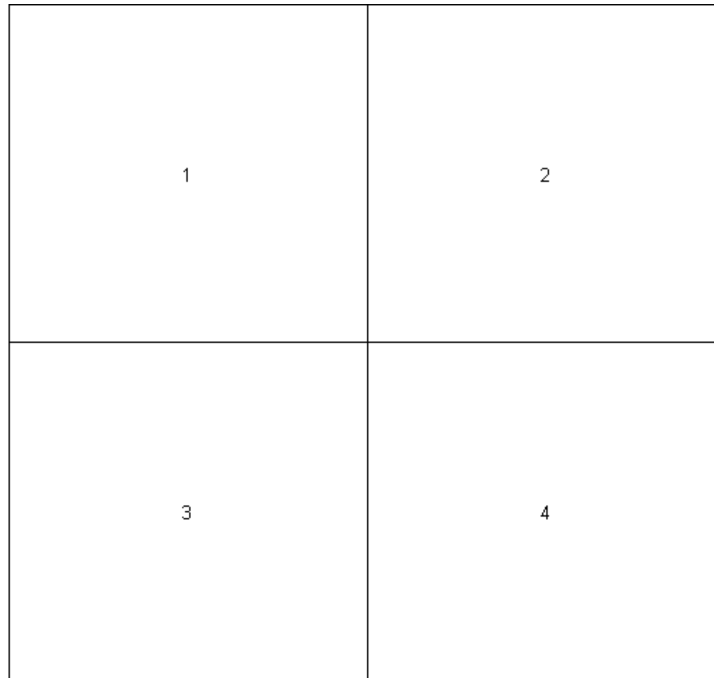
Note, we did not reset the margins, so they are still set as `mar=c(4,4,2,1)`. The only required argument to the `layout` function is a matrix, but arguments can be specified within `matrix` to designate the number of rows and columns. For example in the code below, a matrix with 4 elements is specified, and the additional arguments tell R to output the figures by row, and that 2 columns of figures should be displayed. This specifies that the first plot is in the upper left position, the second plot is in the upper right position, the third plot is in the lower left, and the fourth plot is in the lower right.

```
> layout(matrix(c(1,2,3,4),byrow=TRUE,ncol=2))
> plot(x=xval,y=yval,type="n",main="Plot 1")
> plot(x=xval,y=yval,type="n",main="Plot 2")
> plot(x=xval,y=yval,type="n",main="Plot 3")
```

```
> plot(x=xval,y=yval,type="n",main="Plot 4")
```

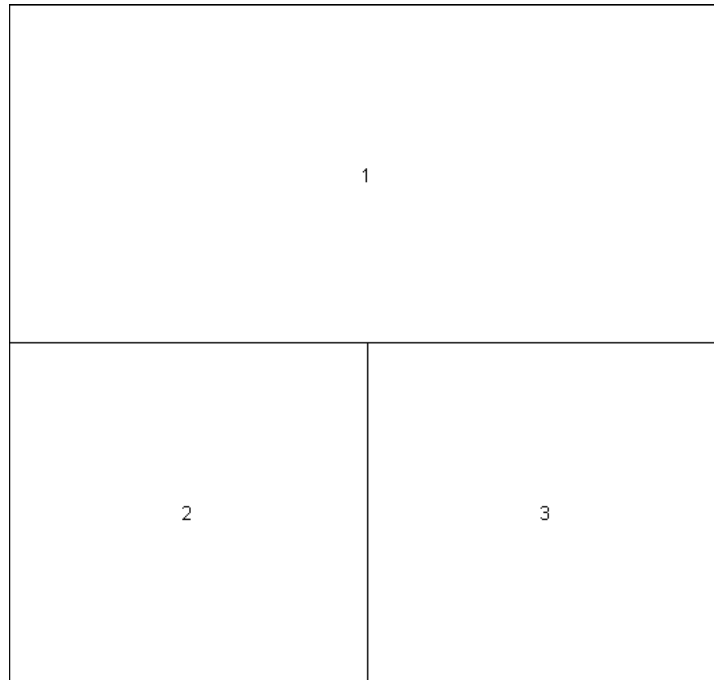
The results are identical to the plot shown above. Instead of actually plotting 4 figures, `layout.show` can be used, with the only required argument being the number of plots. This is a very handy way to check your plot layout:

```
> layout.show(4)
```



This tells you exactly how your plots will be arranged on the page. Let's demonstrate how to create a 3-panel plot with plot regions that have unequal sizes. The code below indicates that 3 plots should be created, with the first plot occupying the first two regions of the matrix, and the second and third plots occupying the third and fourth regions of the matrix, respectively.

```
> layout(matrix(c(1,1,2,3),byrow=TRUE,ncol=2))
> layout.show(3)
```



More creative layouts can also be specified, with some areas containing no plots, and plots with different heights and widths. Try the following layouts:

```
> layout(matrix(c(1,3,0,0,2,2),byrow=TRUE,ncol=2))
> layout.show(3)

> layout(matrix(c(1,3,0,0,2,2),byrow=TRUE,ncol=2),heights=c(2,0.5,1))
> layout.show(3)

> layout(matrix(c(1,3,0,0,2,2),byrow=TRUE,ncol=2),heights=c(2,0.5,1),
  respect=TRUE)
> layout.show(3)
```

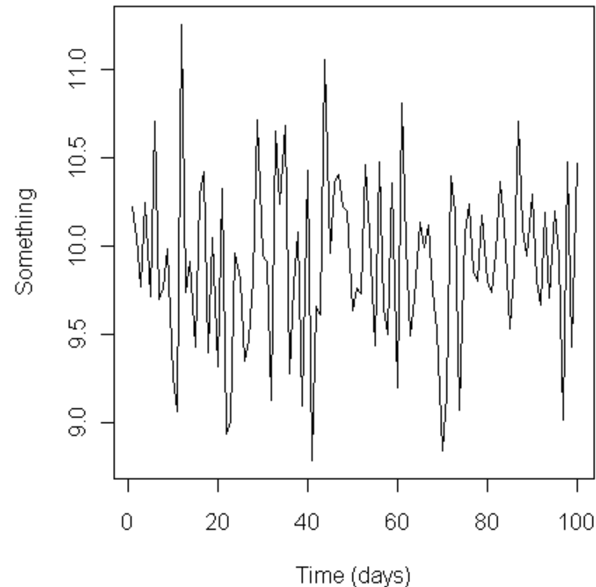
Once you get the hang of using layout, it is very easy to manipulate the number, arrangement, and size of plots in R graphical output. However, if multiple plots of equal size are to be created, specifying `mfrow` is an easier option.

11.2 More on the plot function

While the default `plot` command will produce a high quality plot, R allows the user to have fine control over essentially all aspects of the graphical output. Here we will build upon the basics that were discussed in section 5, beginning with some of the other arguments to the `plot` function. Since it is not important which data we plot to demonstrate these arguments, we will simply generate a dataset using `rnorm`.

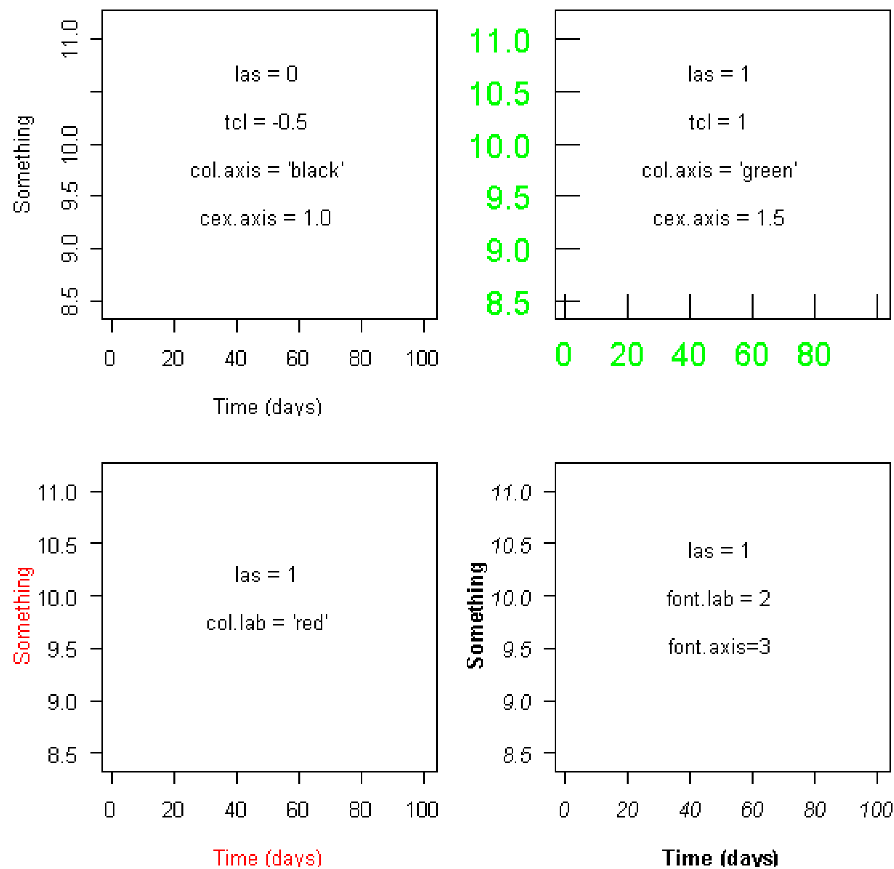
```
> xval<-1:100
> yval<-rnorm(100,mean=10,sd=0.5)
```

```
> plot(xval,yval,type="l",xlab="Time (days)",ylab="Something")
```



Now we will remove the data series and start making changes to the plot. The easiest way to compare plots with different formats is to plot several plots on one page. For this, we will display 4 panels per page, and each plot will be annotated with text describing various arguments:

```
> par(mfrow=c(2,2))
> plot(xval,yval,type="n",xlab="Time (days)",ylab="Something")
> plot(xval,yval,type="n",xlab="",ylab="",las=1,tcl=1,col.axis="green",
     cex.axis=1.5)
> plot(xval,yval,type="n",xlab="Time (days)",ylab="Something",las=1,
     col.lab="red")
> plot(xval,yval,type="n",xlab="Time(days)",ylab="Something",las=1,
     font.lab=2,font.axis=3)
```



Notice that for these plots, `type="n",` which tells R to create a plot with no data shown. The first plot is essentially a default R plot, with axis labels specified. The default values for several arguments are shown in the plot region for the first figure. Remember, that plot parameters can be checked by typing `par()`. If you have not changed the value of the plot parameters, they will indicate the default settings of the various plot arguments. We have already changed `mfrow` and `mar`, so these will correspond to the user-defined settings.

In the first plot, the default values for several arguments (i.e `las`, `tcl`, `col.axis`, and `cex.axis`) are specified. These arguments can be used to change the way that the axes or their features are displayed. There are 4 different settings for `las`, and they are used to set the orientation of the tick mark labels. For example, `las=0` sets the tick mark labels parallel to their respective axis (this is the default) `las=1` sets the tick mark labels horizontal so that they are parallel to the reading direction (i.e. shown in the figure above), `las=2` sets the tick mark labels perpendicular to their respective axes, and `las=3` sets the tick mark labels perpendicular to the reading direction. The argument `tcl` is used to specify the tick mark length. The value tells R how long to make the ticks relative to the text size. Alternatively, `tck` can be used to set the tick mark length relative to the plot size. A setting that we typically use is tcl=0.5 for major tick marks and tcl=0.3 for minor tick marks, but as with many graphical settings, the values used can be modified to suit the user (i.e. you). The argument `col.axis` is used to set the color of the tick mark labels; as mentioned above, there are 657 different colors to choose from. The size of

the characters used for the tick mark labels can be set with the argument `cex.axis`, which is simply the character expansion factor to use for the tick mark labels.

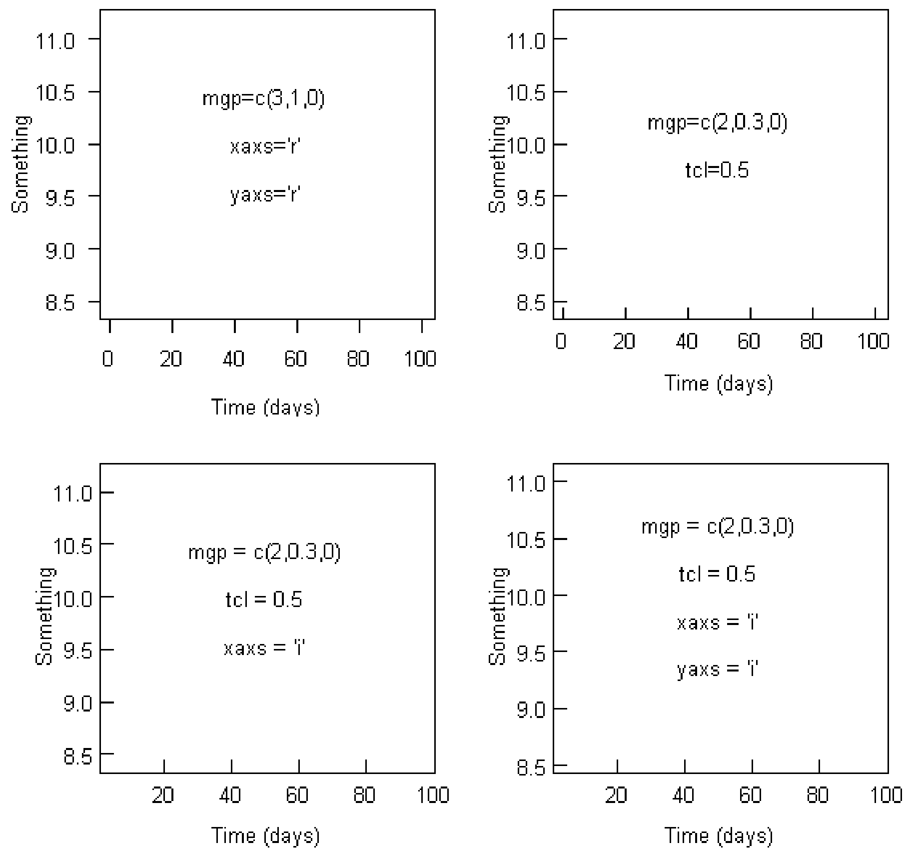
The second plot shows an example of how the plot can be modified by changing the values for the 4 arguments listed in the first plot. Notice the difference in the orientation of the tick mark labels, the difference in tick mark length, and the color and size of the tick mark labels. This plot also shows no axis labels, because we have specified them to be `""`.

The third and fourth plots show further modifications to the axis labels and tick mark labels. Several fonts can be specified, including `1` for Roman or upright, `2` for bold, `3` for slanted or italic, `4` for bold and slanted, and `5` for symbol.

Some other useful arguments include `mgp`, `xaxs`, `yaxs`. The argument `mgp` requires a vector of three values to be specified in a certain order, i.e. the default for `mgp` is: `mgp=c(3,1,0)`, which specifies that the overall axis label is 3 lines away from the edge of the plot region, the tick mark labels are 1 line of text away from the edge of the plot region and the tick marks are on the edge of the plot region, i.e. 0 lines away. The following code provides several examples of how changing the values of the plot arguments can influence the way in which figures are generated.

```
> plot(xval,yval,type="n",xlab="Time(days)",ylab="Something",las=1,
     mgp=c(3,1,0),xaxs="r",yaxs="r")
> plot(xval,yval,type="n",xlab="Time(days)",ylab="Something",las=1,
     mgp=c(2,0.3,0),tcl=0.5)
> plot(xval,yval,type="n",xlab="Time(days)",ylab="Something",las=1,
     mgp=c(2,0.3,0),tcl=0.5,xaxs="i")
> plot(xval,yval,type="n",xlab="Time(days)",ylab="Something",las=1,
     mgp=c(2,0.3,0),tcl=0.5,xaxs="i",yaxs="i")
```

Notice how the axes are modified in the following plots, corresponding to the settings for the various arguments that can be set in the plot command.

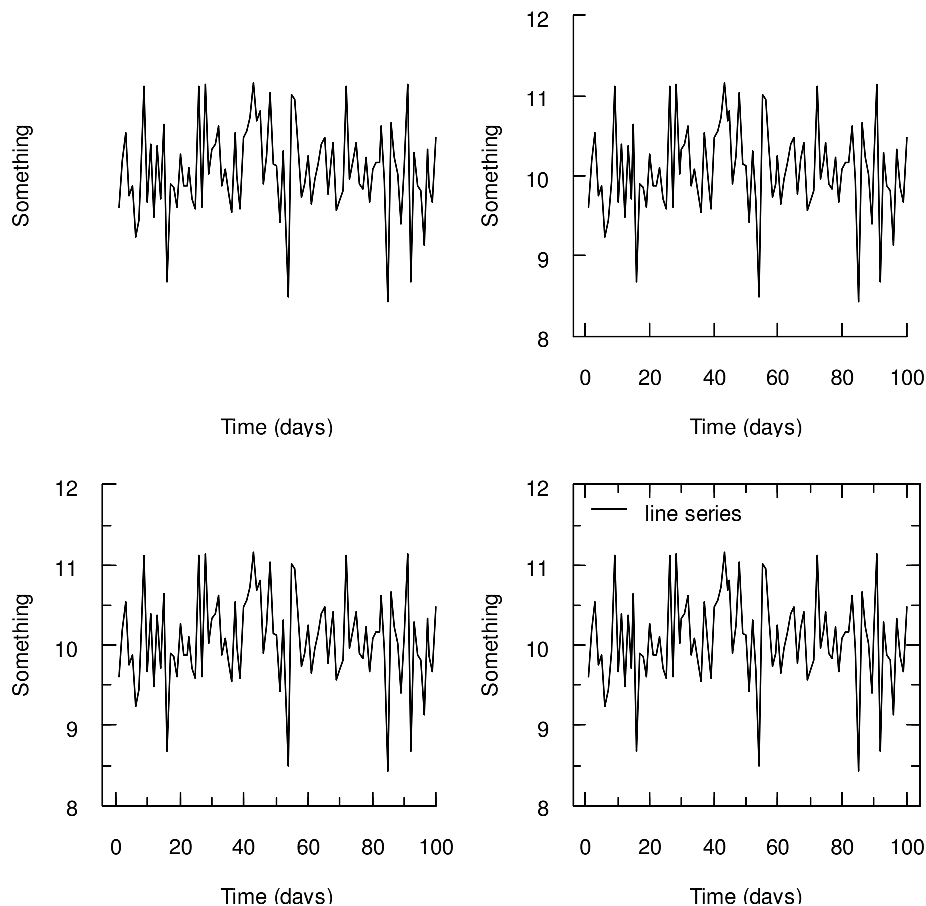


A complete description of the various plot parameter settings that can be specified as arguments to plot can be viewed by typing `?par`. This will provide a list of the plot parameter settings that can either be set by specifying values in a `par` command, or by specifying values as arguments through 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.

Plots can be further customized by adding axes manually, or by annotating the plot by adding lines, segments, text, expressions, legends or other options. Here we describe the creation of user-defined axes, with customized tick marks, on all 4 sides of the plot region.

```
> plot(xval,yval,type="l",xlab="Time(days)",ylab="Something",
     ylim=c(8,12),xlim=c(0,100),yaxs="i",axes=FALSE)

> plot(xval,yval,type="l",xlab="Time(days)",ylab="Something",
     ylim=c(8,12),xlim=c(0,100),yaxs="i",axes=FALSE)
> axis(1,at=seq(0,100,20),labels=c("0","20","40","60","80","100"),tcl=0.5)
> axis(2,at=c(8:12),labels=c("8","9","10","11","12"),tcl=0.5,las=1)

> plot(xval,yval,type="l",xlab="Time(days)",ylab="Something",
     ylim=c(8,12),xlim=c(0,100),yaxs="i",axes=FALSE)
```

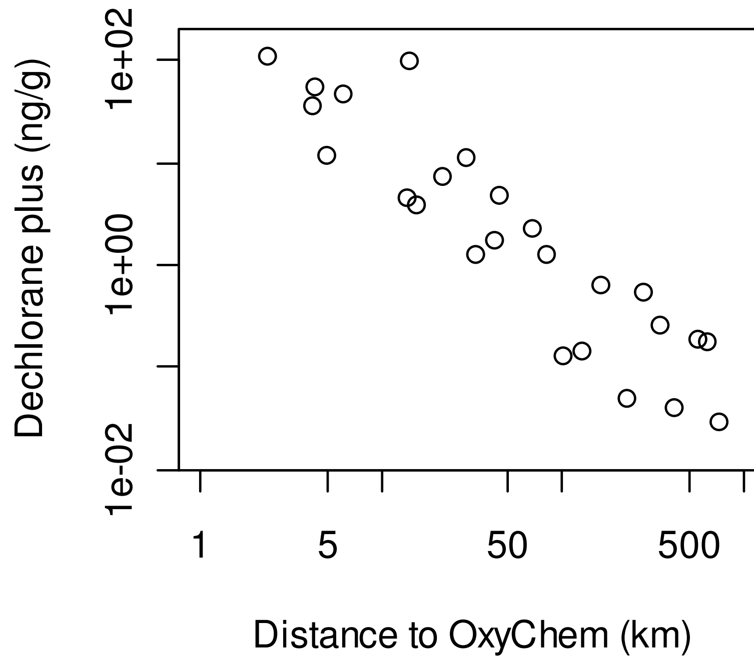
```
> axis(1,at=seq(0,100,20),labels=c("0","20","40","60","80","100"),tcl=0.5)
> axis(1,at=seq(0,100,10),labels=FALSE,tcl=0.3)
> axis(2,at=c(8:12),labels=c("8","9","10","11","12"),tcl=0.5,las=1)
> axis(2,at=seq(8,12,0.5),labels=FALSE,tcl=0.3)

> plot(xval,yval,type="l",xlab="Time(days)",ylab="Something",
    ylim=c(8,12),xlim=c(0,100),yaxs="i",axes=FALSE)
> axis(1,at=seq(0,100,20),labels=c("0","20","40","60","80","100"),tcl=0.5)
> axis(1,at=seq(0,100,10),labels=FALSE,tcl=0.3)
> axis(2,at=c(8:12),labels=c("8","9","10","11","12"),tcl=0.5,las=1)
> axis(2,at=seq(8,12,0.5),labels=FALSE,tcl=0.3)
> axis(3,at=seq(0,100,20),labels=FALSE,tcl=0.5)
> axis(3,at=seq(0,100,10),labels=FALSE,tcl=0.3)
> axis(4,at=c(8:12),labels=FALSE,tcl=0.5,las=1)
> axis(4,at=seq(8,12,0.5),labels=FALSE,tcl=0.3)
> legend("topleft",c("line series"),lty=1,bty="n")
```



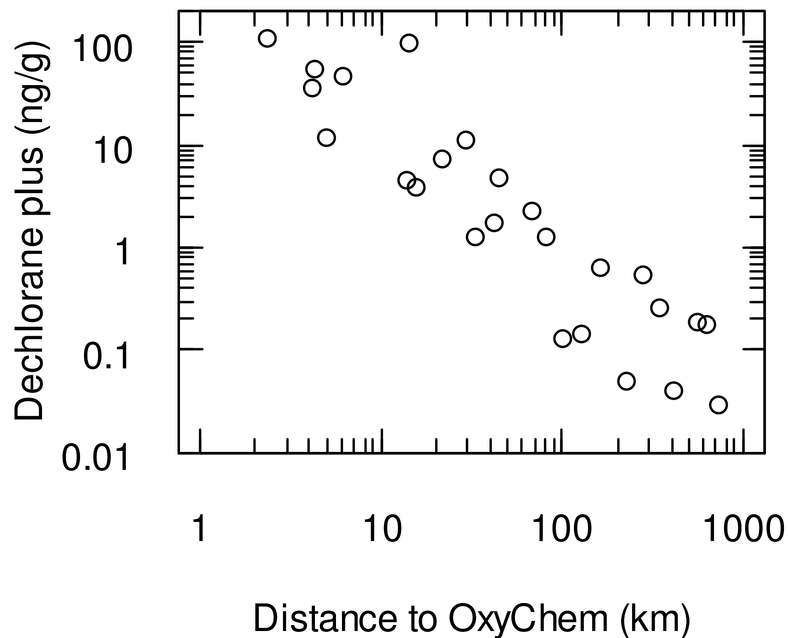
Suppose you wanted the x and y scales to be logarithmic, rather than arithmetic. As and example, we will use the OxyChem.txt dataset and plot concentration of dechlorane plus in pine tree bark vs. distance from OxyChem.

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

> plot(oxy_chem.dat$dist,oxy_chem.dat$dechlor,type="p",log="xy",
xlab="Distance to OxyChem (km)",ylab="Dechlorane plus(ng/g)",
ylim=c(0.01,200),xlim=c(1,1000),yaxs="i")
```

The resulting plot looks like:



Suppose you want the axes to look like this:

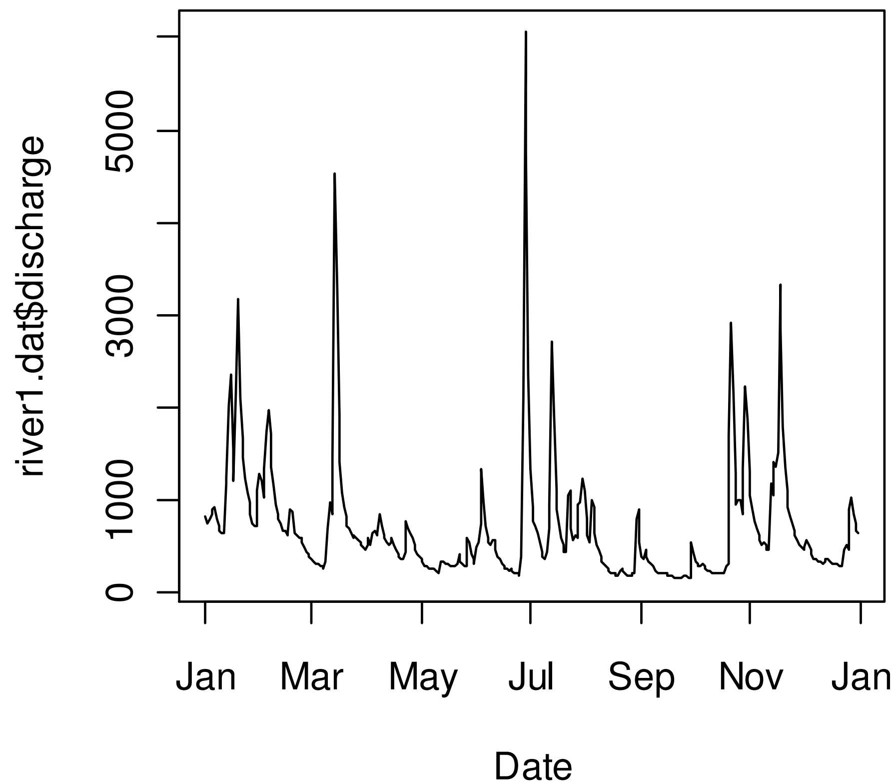


Think about how you might make these axes look like this, given the examples above.

The `eaxis` function in the `sfsmisc` package can be used to create "pretty" log axes automatically, but is not covered in this workshop.

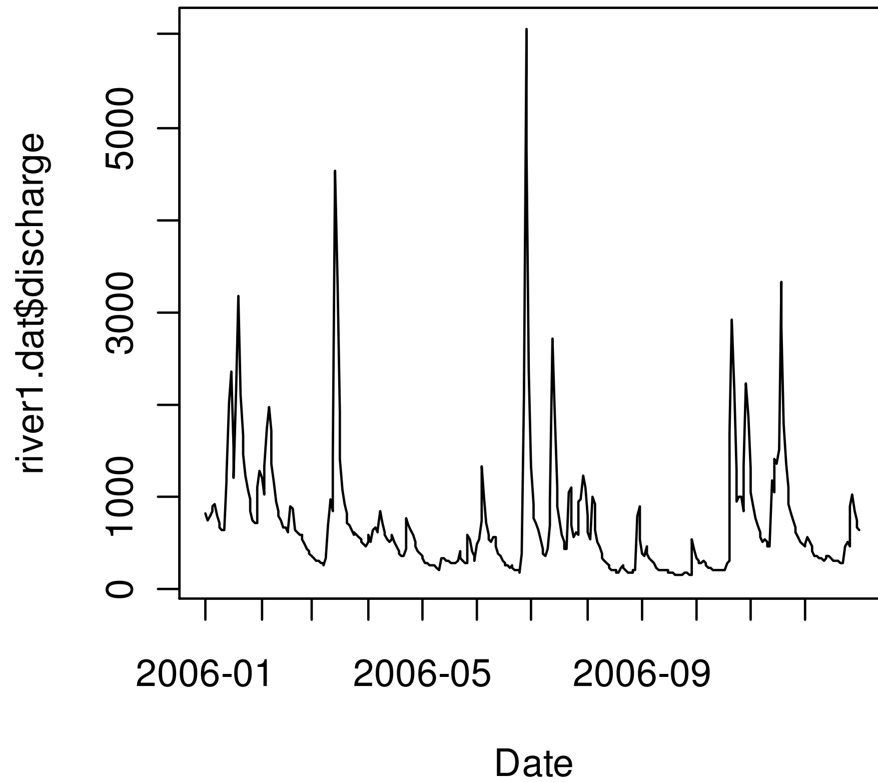
R has nice capabilities for plotting time series data. We will demonstrate some of this functionality with the River_Flow.txt dataset. This represents another way that axes can be modified by the user to suit their needs. The code immediately below will produce R's default time series axis.

```
> rivers.dat<-read.table("River_flow.txt",header=T)
>
> river1.dat<-na.omit(subset(rivers.dat,site==1509000))

> plot(as.Date(river1.dat$date),river1.dat$discharge,xlab="Date",type="l")
```

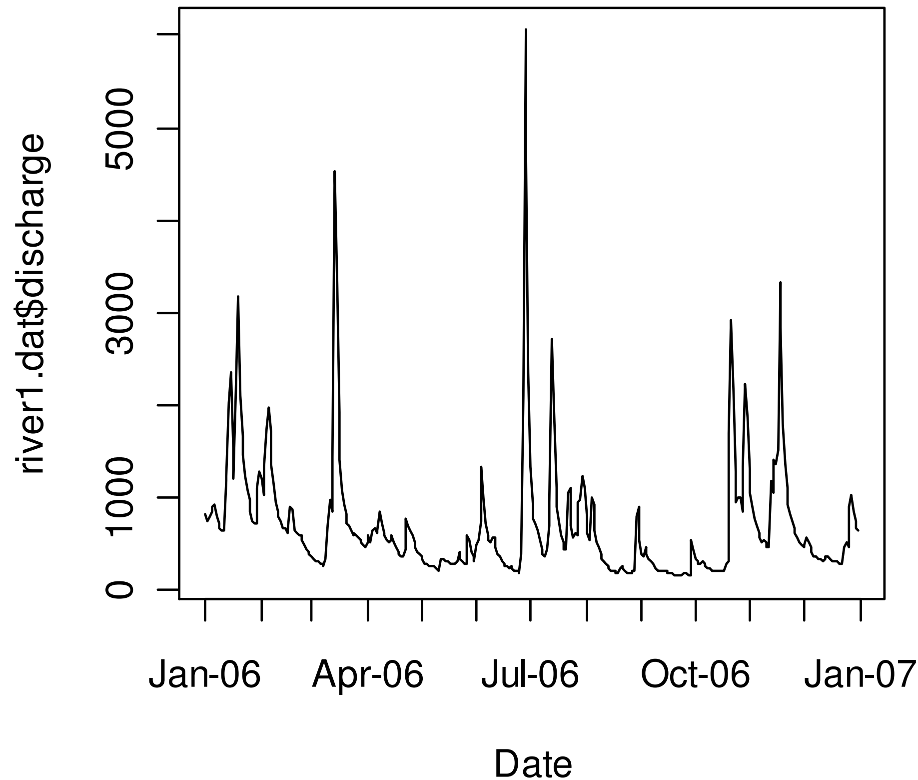


This plot may be fine for some situations, but if you need more information in the x-axis labels, you will have to modify the code. There are several ways that this can be done, and there are several options for specifying dates.

```
> plot(as.Date(river1.dat$date),river1.dat$discharge,xlab="Date",type="l",
xaxt="n",xlim=c(min(as.Date(river1.dat$date)),
max((as.Date(river1.dat$date)))))
> axis.Date(1,as.Date(river1.dat$date),
at=seq(min(as.Date(river1.dat$date)),max(as.Date(river1.dat$date)),
"1 months"), format="%Y-%m",cex.axis=1,labels=TRUE)
```




```
> plot(as.Date(river1.dat$date),river1.dat$discharge,xlab="Date",
type="l",xaxt="n",xlim=c(min(as.Date(river1.dat$date)),
max((as.Date(river1.dat$date)))))

> axis.Date(1,as.Date(river1.dat$date),
at=seq(as.Date("2006-01-01"),as.Date("2007-01-01"),"1 months"),
format="%b-%y",cex.axis=1,labels=TRUE)
```



There are many formats that you can use for display of dates on the axis. To see some of the options, type `?strptime`.

It may be preferable to perform a few operations on the data before the plot is created. This may make it easier to keep track of your code. For example, to create the plot shown above, you can enter the following code instead:

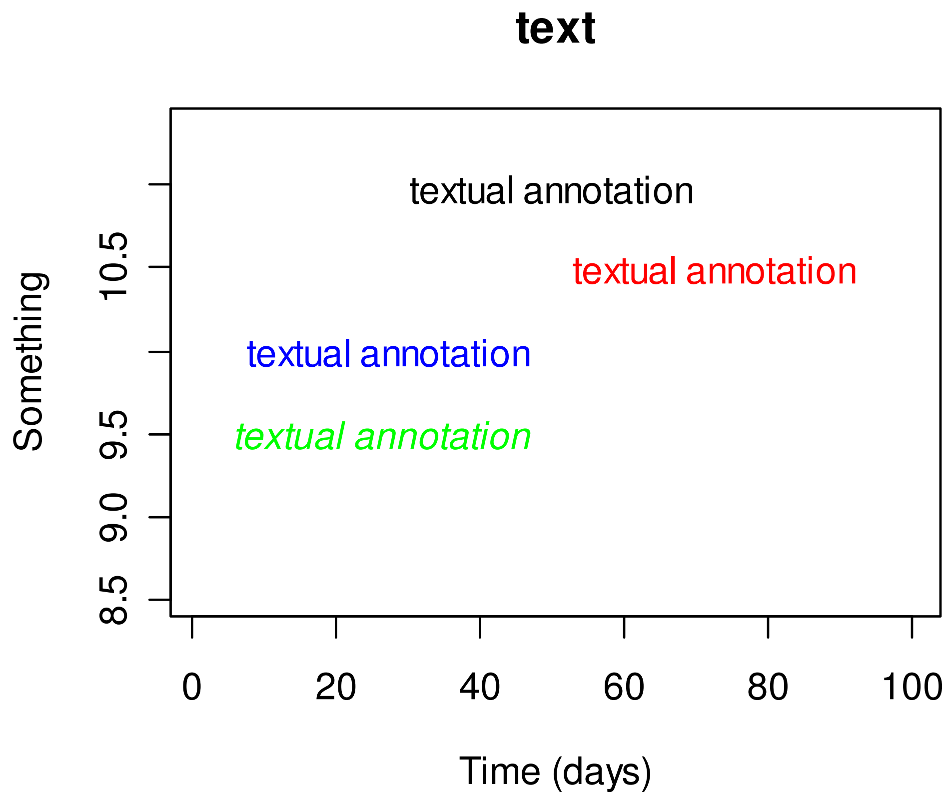
```
> plot(as.Date(river1.dat$date),river1.dat$discharge,xlab="Date",
type="l",xaxt="n",xlim=c(min(as.Date(river1.dat$date)),
max((as.Date(river1.dat$date)))))

> date_axis<-as.Date(river1.dat$date)
> min_date<-min(as.Date(river1.dat$date))
> max_date<-max(as.Date(river1.dat$date))

> axis.Date(1,date_axis,at=seq(min_date,max_date+1,"1 months"),
format="%b-%y",cex.axis=1,labels=TRUE)
```

11.3 Annotating plots

Now we will show some very simple ways to annotate plots, including adding text, lines, arrows, legends, segments, and expressions. We will build upon the last plot that we created, but we will plot as `type="n"`, so that the data are not shown. Where possible, the different lines of code will be associated with a different colored object in the associated figure.

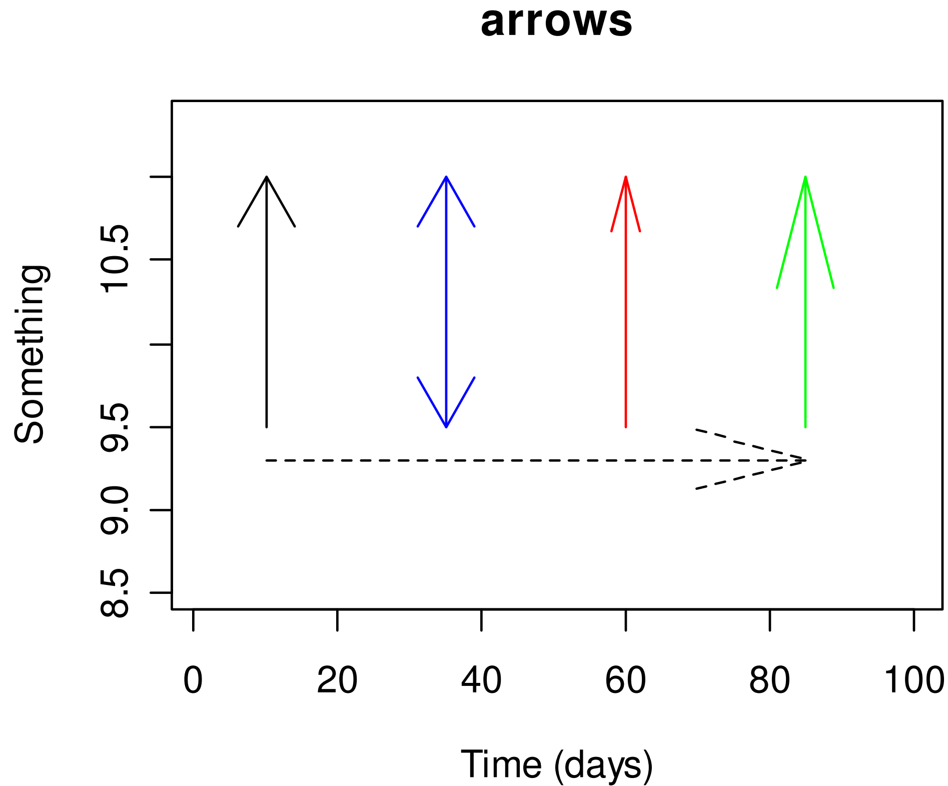
The following shows that text can be placed any where in the figure, with different characteristics, including alignment, color, and font type. There are also methods for changing the orientation of text, but that will not be covered here.

```
> xval<-1:100
> yval<-rnorm(100,mean=10,sd=0.5)
> plot(xval,yval,type="n",xlab="Time (days)",ylab="Something",main="text")
> text(50,11,"textual annotation")
> text(50,10.5,"textual annotation",pos=4,col="red")
> text(50,10,"textual annotation",pos=2,col="blue")
> text(50,9.5,"textual annotation",pos=2,col="green",font=3)
```



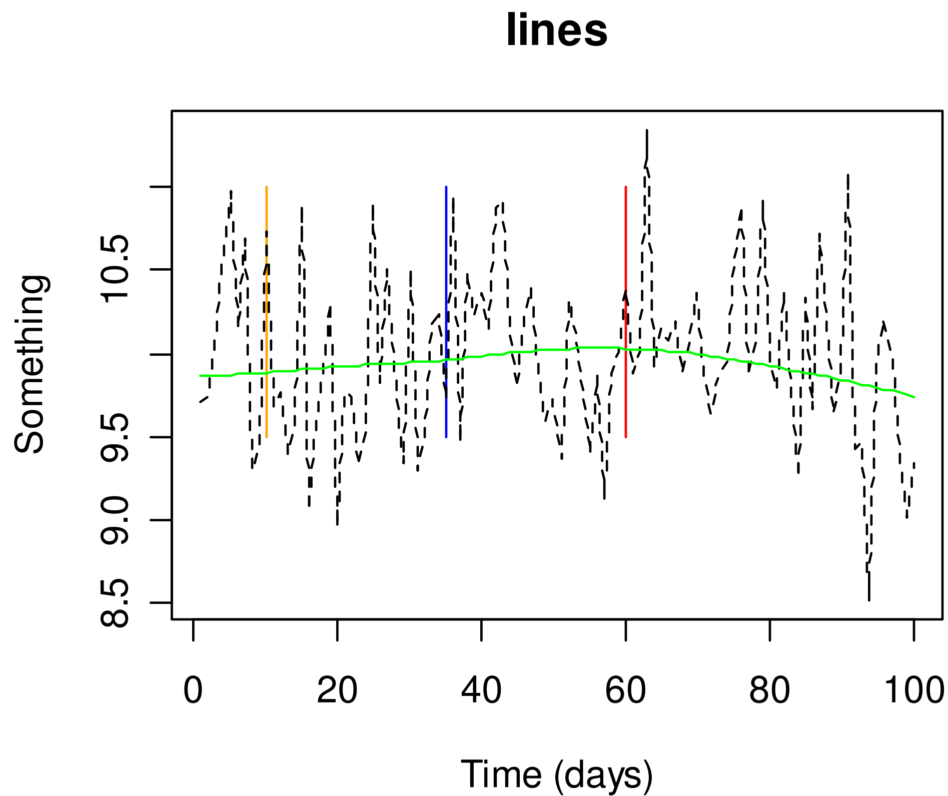
With `arrows`, it is easy to create arrows with different colors, different arrow lengths, different angles, different colors, and different line types. Note that line width can also be adjusted.

```
> plot(xval,yval,type="n",xlab="Time
(days)",ylab="Something",main="arrows")
> arrows(10,9.5,10,11)
> arrows(35,9.5,35,11,code=3,col="blue")
> arrows(60,9.5,60,11,code=2,col="red",angle=15)
> arrows(85,9.5,85,11,code=2,col="green",angle=15,length=0.5)
> arrows(10,9.3,85,9.3,code=2,col="black",angle=15,length=0.5,lty=2)
```



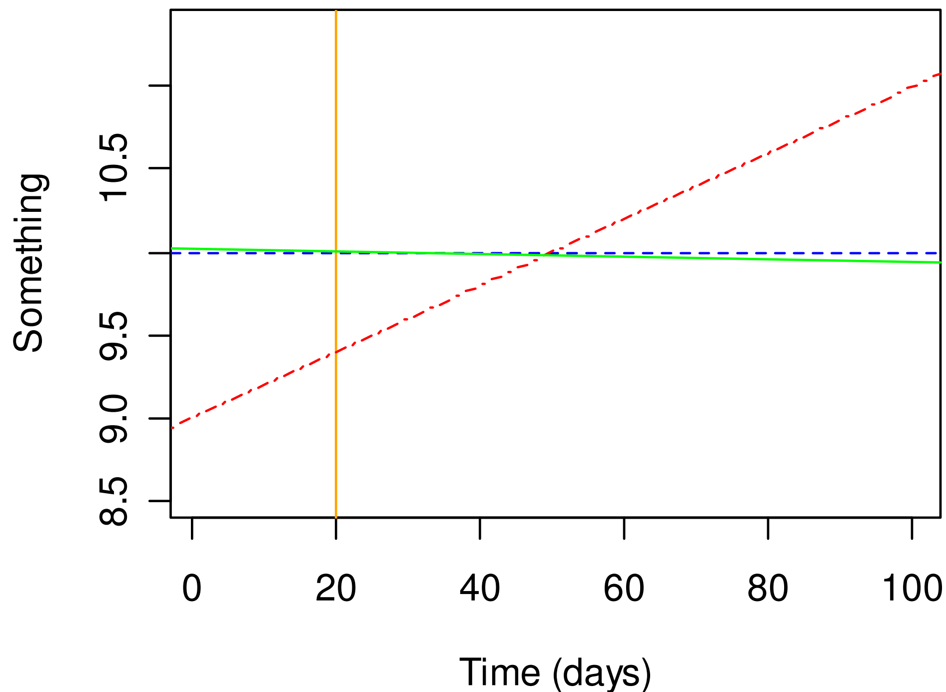
Here we demonstrate that different types of lines can be plotted. It is possible to specify lines as x and y coordinates, as the data series, or with different functions, including `lowess`. As with the functions above, different line types, colors, and widths can be specified.

```
> plot(xval,yval,type="n",xlab="Time
(days)",ylab="Something",main="lines")
> lines(x=c(10,10),y=c(9.5,11),col="orange")
> lines(x=c(35,35),y=c(9.5,11),col="blue")
> lines(x=c(60,60),y=c(9.5,11),col="red",lty=1)
> lines(yval,col="black",lty=2)
> lines(lowess(yval),col="green")
```



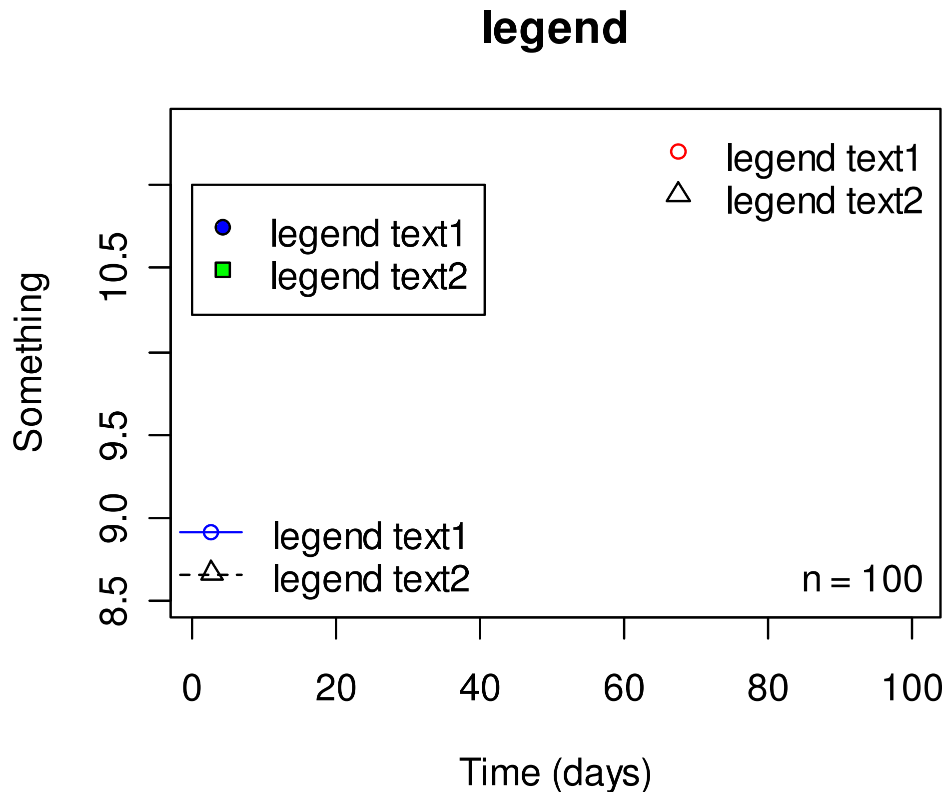
The function `abline` can be used to plot lines as vertical or horizontal lines, or as lines with a given intercept and slope. A linear model can also be specified.

```
> plot(xval,yval,type="n",xlab="Time
(days)",ylab="Something",main="abline")
> abline(v=20,col="orange")
> abline(h=10,col="blue",lty=2)
> abline(9,0.02,col="red",lty=4)
> abline(lm(yval~xval),col="green")
```



The `legend` function is very useful, since it can be used for more than just legends. The `legend` function can be used to place a legend in various places within a figure, with various attributes. For example, the legend can contain symbols, lines, and text. You can use the legend to simply annotate the corner of plots, since it is very easy to specify a location.

```
> plot(xval,yval,type="n",xlab="Time
(days)",ylab="Something",main="legend")

> legend(0,11,c("legend text1","legend
text2"),pch=c(21,22),pt.bg=c("blue","green"))

> legend("topright",c("legend text1","legend
text2"),pch=c(1,2),col=c("red","black"),bty="n")

> legend("bottomleft",c("legend text1","legend
text2"),pch=c(1,2),lty=c(1,2),col=c("blue","black"),bty="n")

> legend("bottomright",c("n = 100"),bty="n")
```

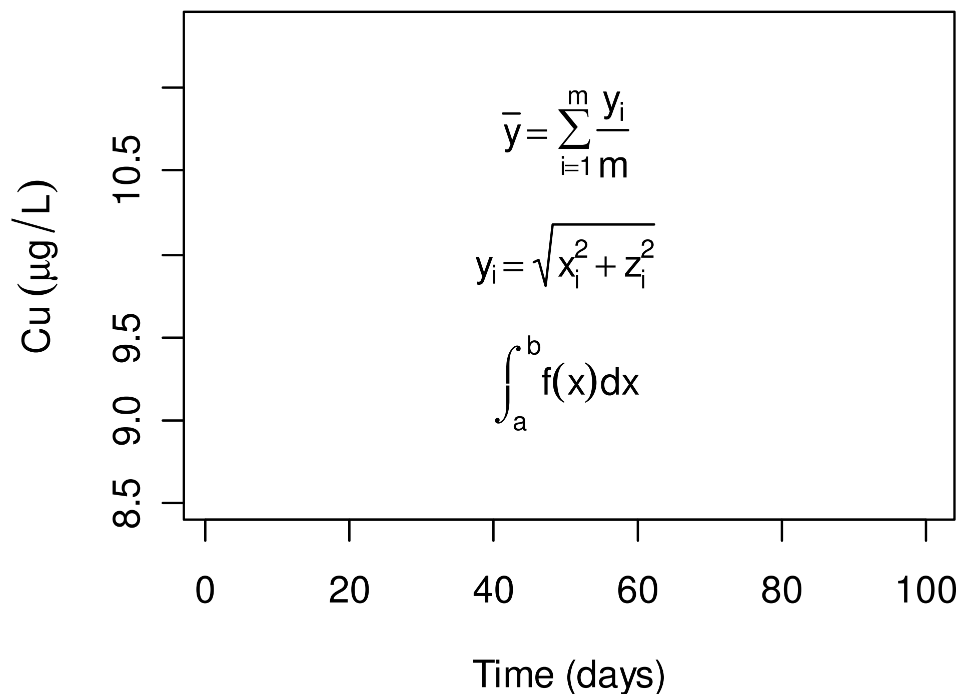


Equations and symbols can be added to figures by using the `expression` function within a `text` function call. The syntax for writing equations and symbols with function can be viewed by typing:

```
> demo(plotmath)
```

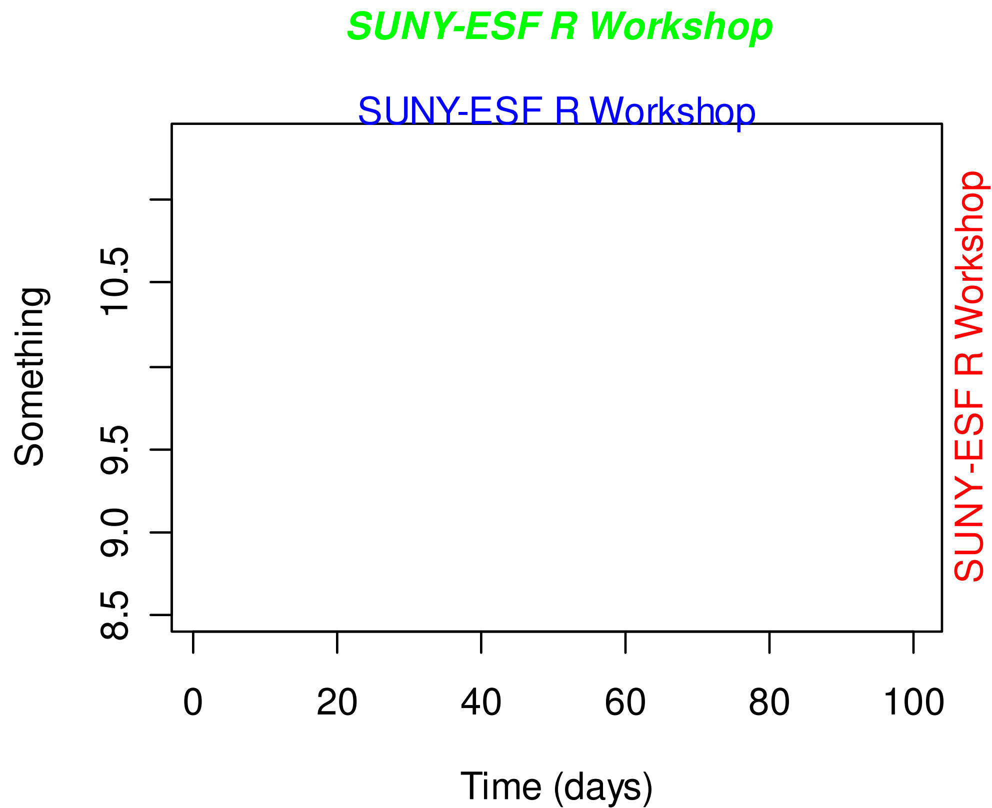
The code below shows just a few uses of expressions. Note the use of expression in the y-axis label.

```
> plot(xval,yval,type="n",xlab="Time (days)",ylab=expression("Cu"
~(mu*g/L)),main="expression")
> text(50,10.75,expression(bar(y)==sum(frac(y[i],m),i==1,m)))
> text(50,10,expression(y[i]==sqrt(x[i]^2+z[i]^2)))
> text(50,9.25,expression(integral(f(x)*dx,a,b)))
```



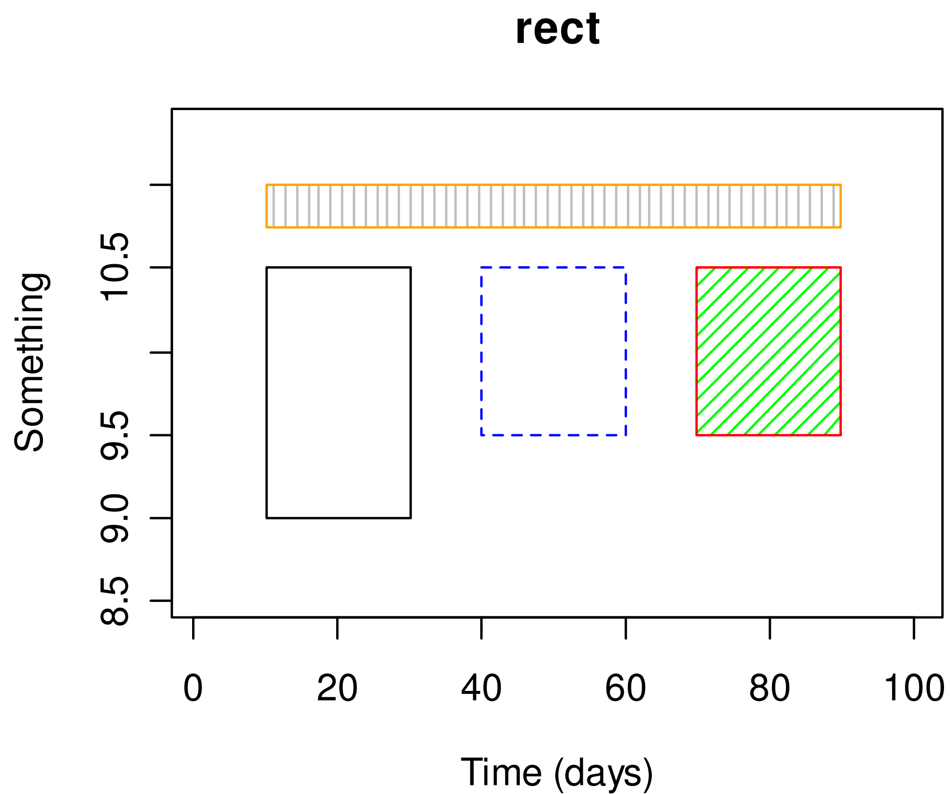
It is also possible to annotate the plot margins. The side of the plot needs to be specified, along with the number of lines away from the plotting region. Font and colors can also be modified, as with the functions described above.

```
> plot(xval,yval,type="n",xlab="Time (days)",ylab="Something")
> mtext("SUNY-ESF R Workshop",side=3,col="blue")
> mtext("SUNY-ESF R Workshop",side=4,col="red")
> mtext("SUNY-ESF R Workshop",side=3,col="green",line=2,font=4)
```



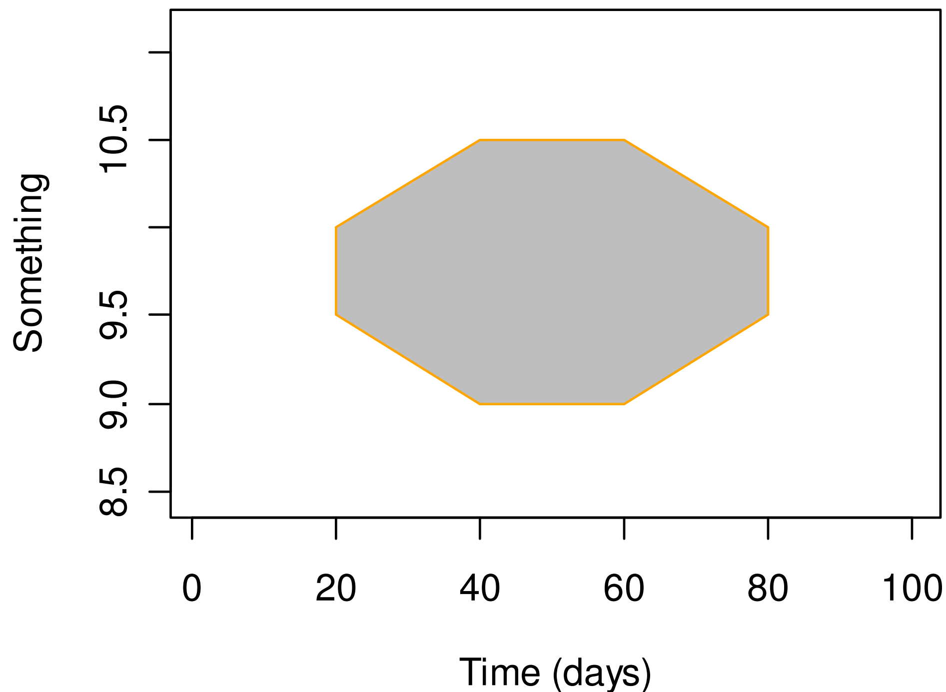
Some simple examples of how to add rectangles to plots with `rect` is shown below:

```
> plot(xval,yval,type="n",xlab="Time (days)",ylab="Something",main="rect")
> rect(10,9,30,10.5)
> rect(40,9.5,60,10.5,border="blue",lty=2)
> rect(70,9.5,90,10.5,border="red",density=20,col="green")
> rect(10,10.75,90,11,border="orange",density=20,col="grey",angle=90)
```



The code below shows a simple example of how to use `polygon`:

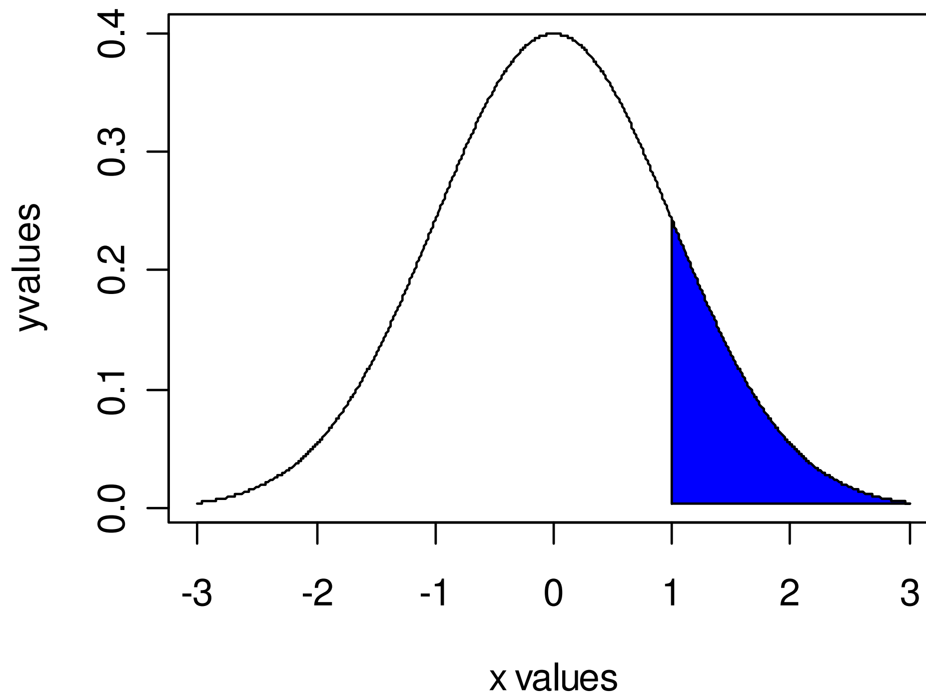
```
> plot(xval,yval,type="n",xlab="Time
(days)",ylab="Something",main="polygon")
> polygon(c(40,20,20,40,60,80,80,60),c(9,9.5,10,10.5,10.5,10,9.5,9),
lty=1,border="orange",col="grey",angle=45)
```



A more interesting example of using `polygon` showing the vectorized nature of the vertices follows:

```
> xval<-seq(-3,3,0.01)
> yval<-dnorm(xval)
> plot(xval,yval,xlab="x values",ylab="yvalues",type="l")
> polygon(c(1,xval[xval>=1]),c(yval[xval==3],yval[xval>=1]),col="blue")
```

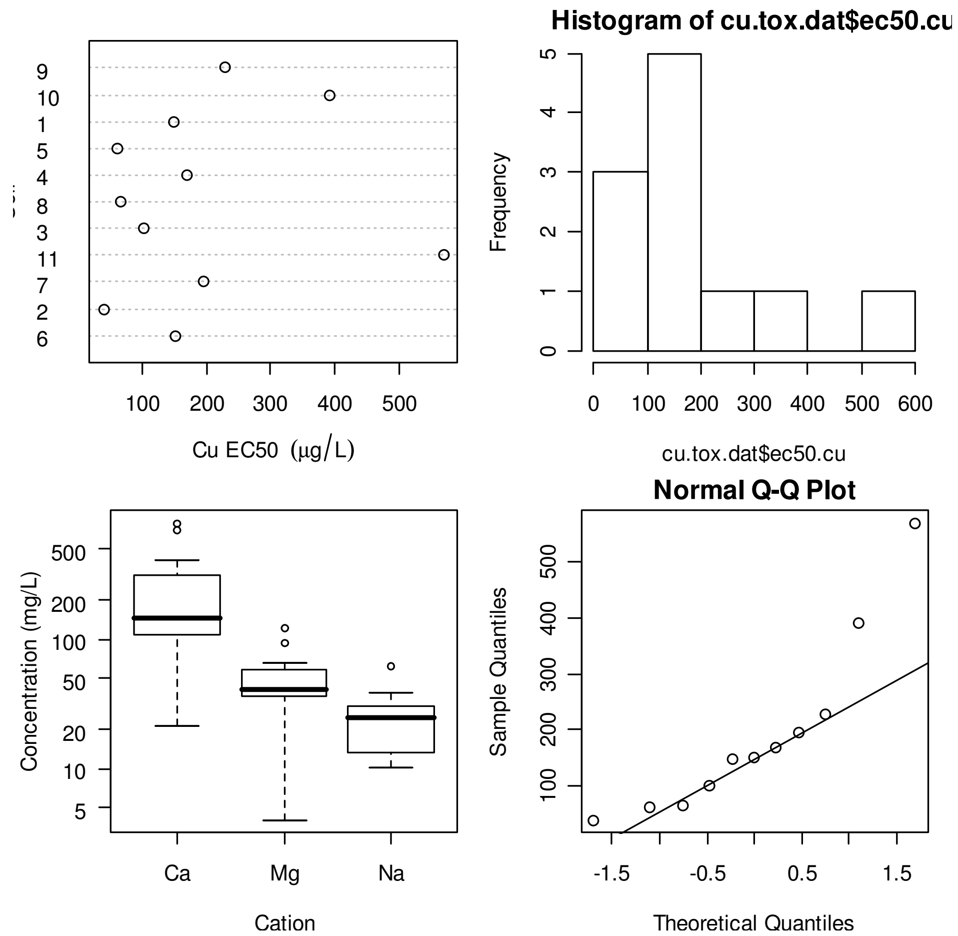
We will cover the `dnorm` function later, but as you might guess, it is associated with the normal distribution. This figure shows that you can specify a vector for the vertices, and the resulting polygon appears to have smooth edges.



11.4 Other high-level plotting functions

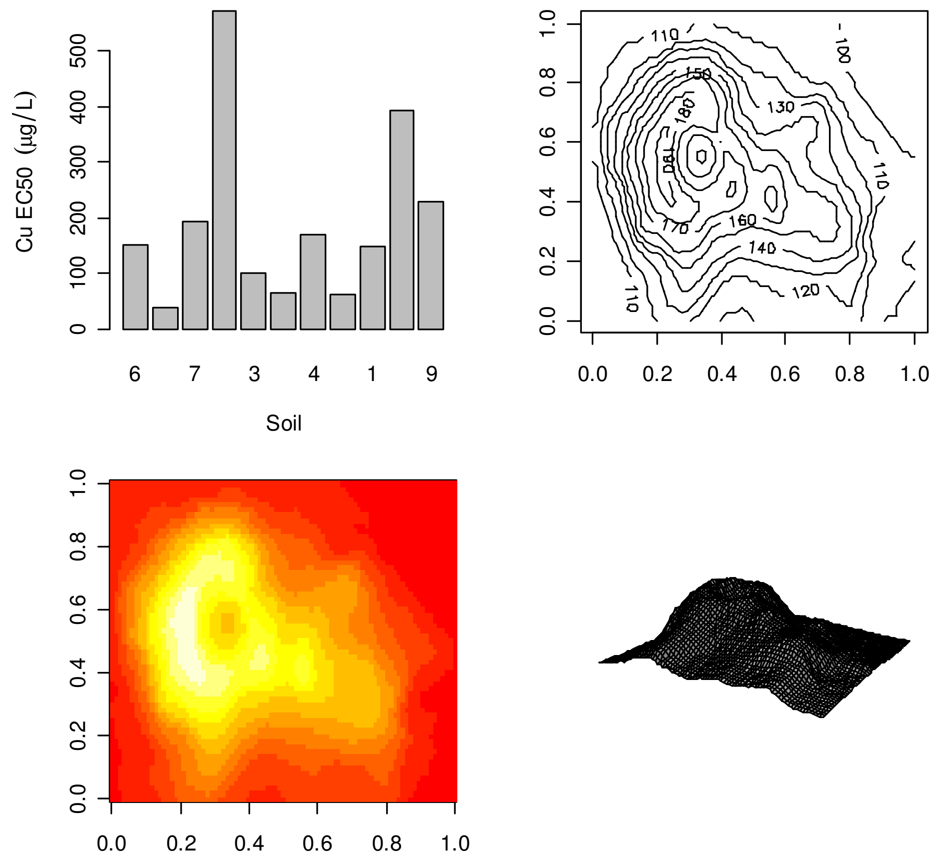
Many other high level plotting functions exist, and the code below demonstrates just a few examples. In the code below we show the use of `dotchart`, `hist`, `boxplot`, and `qqnorm`. These are all very useful plots, and as with many of R's graphics, they can be further modified by the user.

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

> dotchart(cu.tox.dat$ec50.cu, labels=cu.tox.dat$soil,
     xlab=expression("Cu EC50 "~(mu*g/L)),ylab="Soil")

> hist(cu.tox.dat$ec50.cu)

> cations<-list(cu.tox.dat$c.ca,cu.tox.dat$c.mg,cu.tox.dat$c.na)

> boxplot(cations,names=c("Ca","Mg","Na"),log="y",las=1,xlab="Cation",
     ylab="Concentration (mg/L)")

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



The plots shown below include the `barplot`, the `contour` plot, the `image` plot, and the `persp` plot. The arguments to these plot functions can also be modified by the user.

```
> barplot(cu.tox.dat$ec50.cu, names.arg=cu.tox.dat$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)
```



11.5 Graphics output

R graphics output can be produced in a wide 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 (i.e. a type of graphical output is specified), and then it must be closed to complete the output (i.e. the device is turned of with `dev.off()`).

For example, to create a png file containing the plots that we just created above:

```
> png(file="esf_1.png")
> par(mfrow=c(1,1))
> image(volcano)
> par(mfrow=c(2,2))
> barplot(cu.tox.dat$ec50.cu, names.arg=cu.tox.dat$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)
> dev.off()
windows
      2
```

```
> png(file="esf_2.png")
> par(mfrow=c(1,1))
> image(volcano)
> dev.off()
null device
          1
```

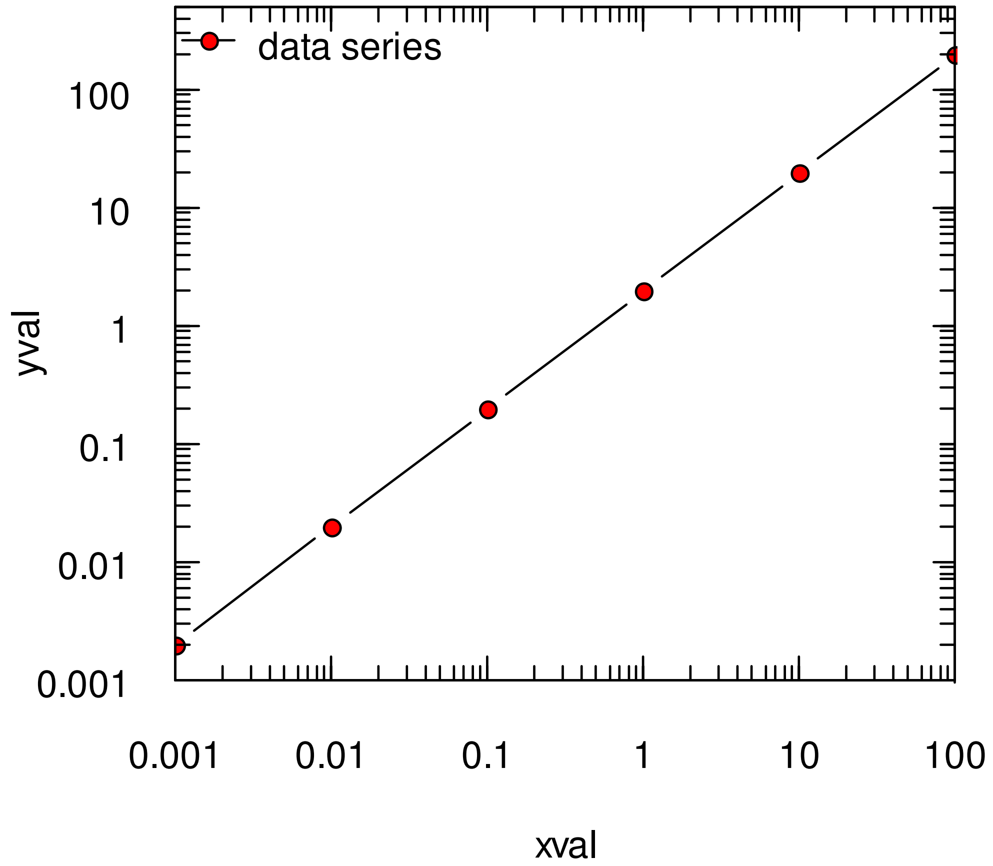
Navigate to your working directory and open the png files. A png file cannot be more than one page, so if multiple pages are desired, they must be in different files. This is not the case with pdf files.

```
> pdf(file="esf_1.pdf")
> par(mfrow=c(2,2))
> barplot(cu.tox.dat$ec50.cu, names.arg=cu.tox.dat$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)
> par(mfrow=c(1,1))
> image(volcano)
> dev.off()
null device
          1
```

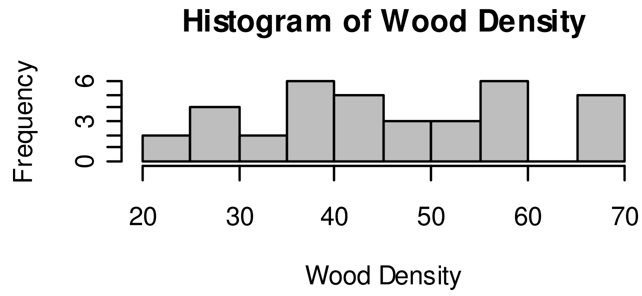
There are several optional arguments that can be specified for the various graphical outputs. For example, to see the arguments for the `pdf` function, type `?pdf`. Output file formats include png, jpeg, bmp, postscript, and others.

Exercises

1. Recreate the following plot. You will have to create your own axes using the `axis` command.

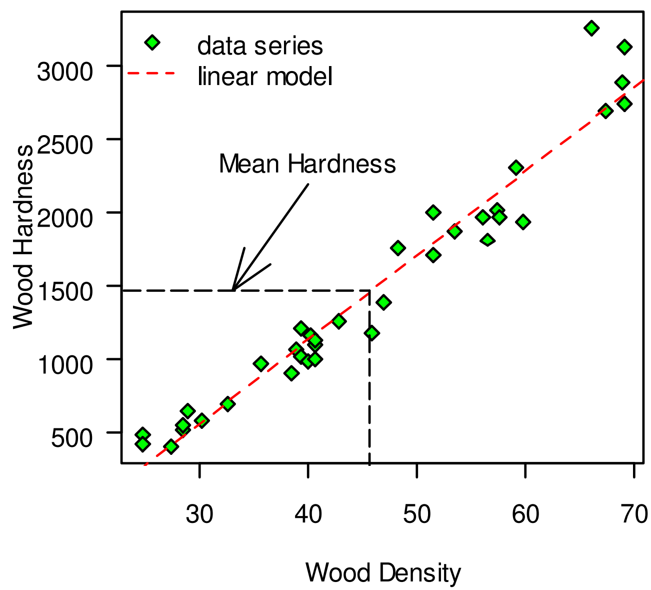


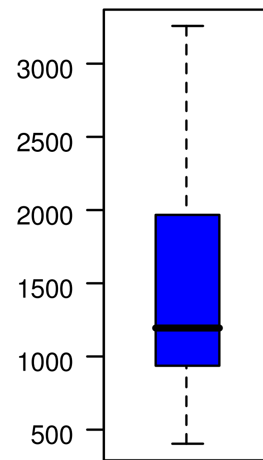
2. Using the Janka.txt dataset that was used earlier for the `lm` example, create the following summary plots. You will have to use `layout` to get plotting regions of different size. Write this file to a pdf named graphics2.pdf.



Mean hardness
= 1470

Mean density
= 45.7





12. Generalized linear models

Crawley 2007: Chapter 13, R-Intro: Section 11.6.2

12.1 The glm function

Generalized linear models are a very flexible class of statistical models that allow one to specify a response distribution and a link function. The form of a GLM is given below.

$$g\left(E\left(Y\right)\right)=\beta_0+\beta_1x_1+\beta_2x_2...+\beta_mx_m$$

In this equation, g is the link function, $E\left(Y\right)$ is the expected value of the response variable, x_1 through x_m are predictors, and β_0 through β_m are coefficients. In R, GLM models can be specified using the `glm` function. There are eght different error distributions available in `glm`, including binomial and poisson, each with a default link function. The help file for `glm` gives the following information.

```
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,
    ...)
```

When a response variable is a binomial (e.g. presence/absence, dead/alive) or a proportion (i.e. a value between zero and unity), it is not appropriate to apply linear regression. Instead, logistic regression, which utilizes a logit transformation, is appropriate. To carry out logistic regression in `glm`, we just need to specify a binomial distribution and a logistic link function. Note that 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
```

If you have a data frame with one row for each subject (e.g. success or failure) it is easy to specify the model formula for logistic regression: `response~predictor`. 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)
> mod.1

Call:  glm(formula = prop.dead ~ cu, family = binomial, data = tox.dat,
weights = tot)

Coefficients:
(Intercept)           cu
    -4.4162       0.1742

Degrees of Freedom: 6 Total (i.e. Null);  5 Residual
Null Deviance:      119.8
Residual Deviance: 7.361        AIC: 22.89

> 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 ***
```

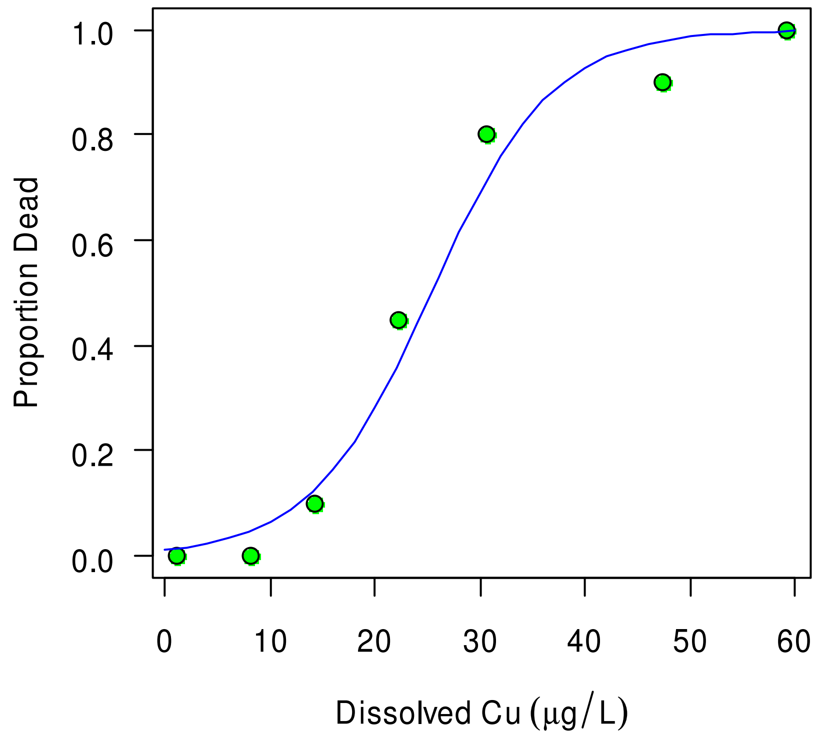
```
cu           0.17425    0.03067   5.681 1.34e-08 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(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
```

The other method is shown below.

```
> resp<-cbind(tox.dat$dead,tox.dat$alive)
> mod.2<-glm(resp~cu,binomial,data=tox.dat)
> mod.2

Call:  glm(formula = resp ~ cu, family = binomial, data = tox.dat)

Coefficients:
(Intercept)           cu
    -4.4162       0.1742

Degrees of Freedom: 6 Total (i.e. Null);  5 Residual
Null Deviance:      119.8
Residual Deviance: 7.361        AIC: 22.89
```

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)
> points(predict.dat$cu,predict.dat$prop.dead,type="l",col="blue")
```



GLM can be applied to a wide range of model and data types. 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.

```
> 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 let's 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)
```

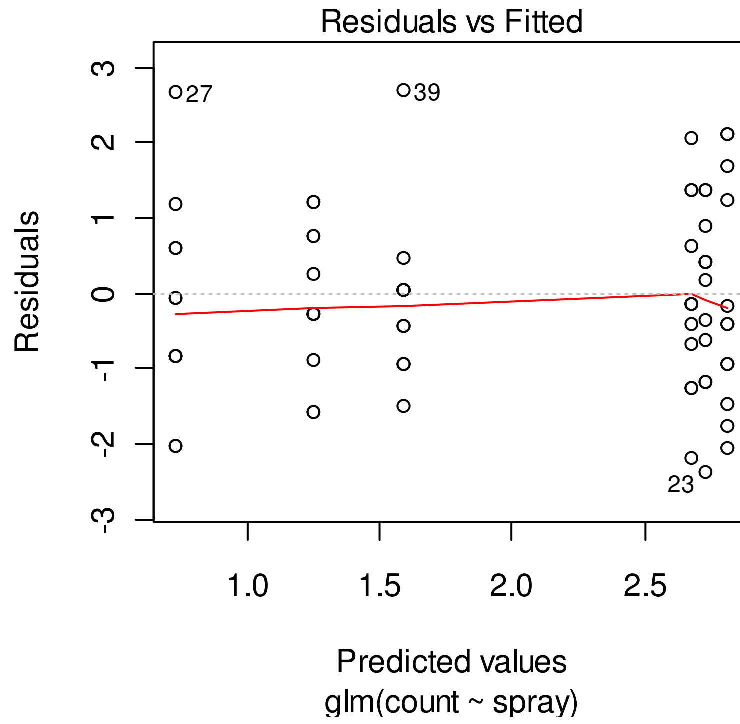
```
Call:
glm(formula = count ~ spray, family = "poisson", data = insects.dat)

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
```

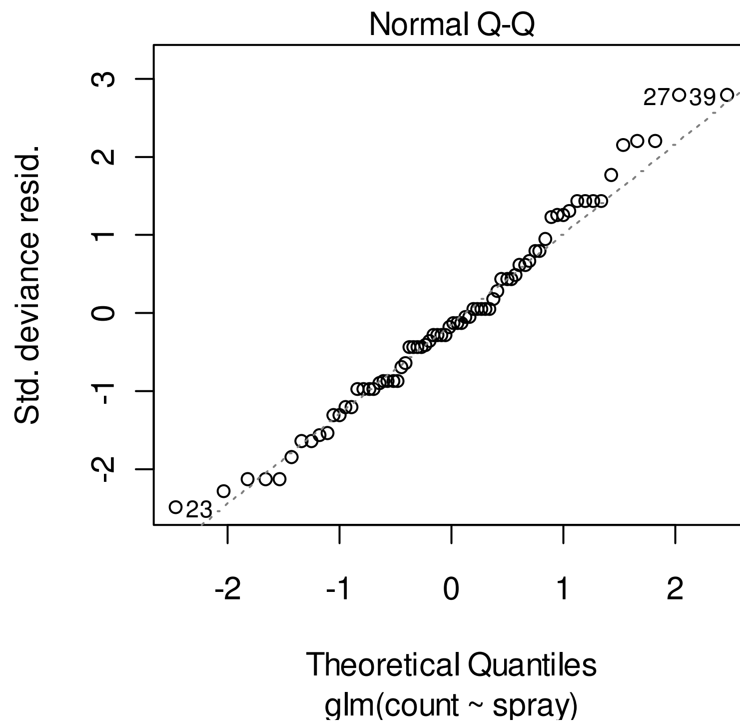
To get an ANOVA table, we can use the `anova` function.

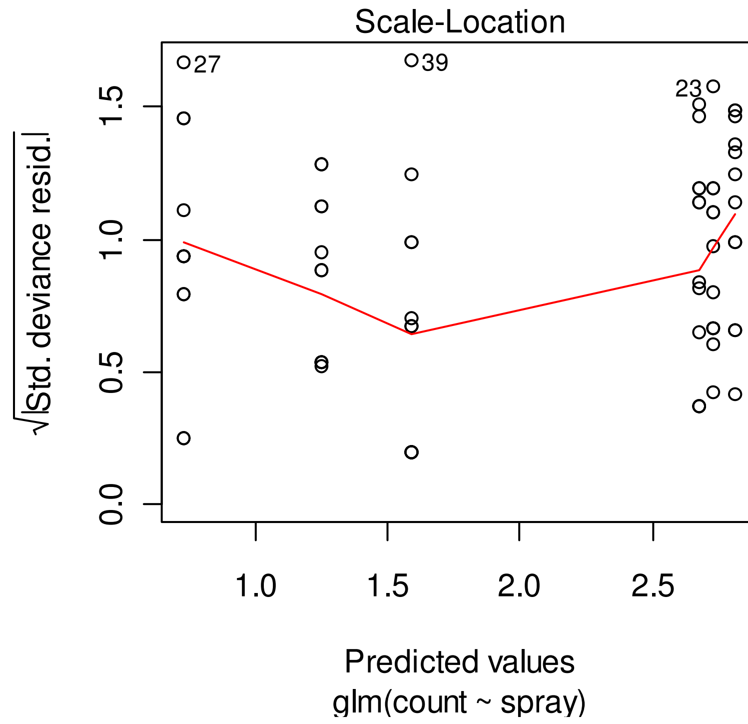
```
> anova(mod.1,test="Chisq")
Analysis of Deviance Table

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
```

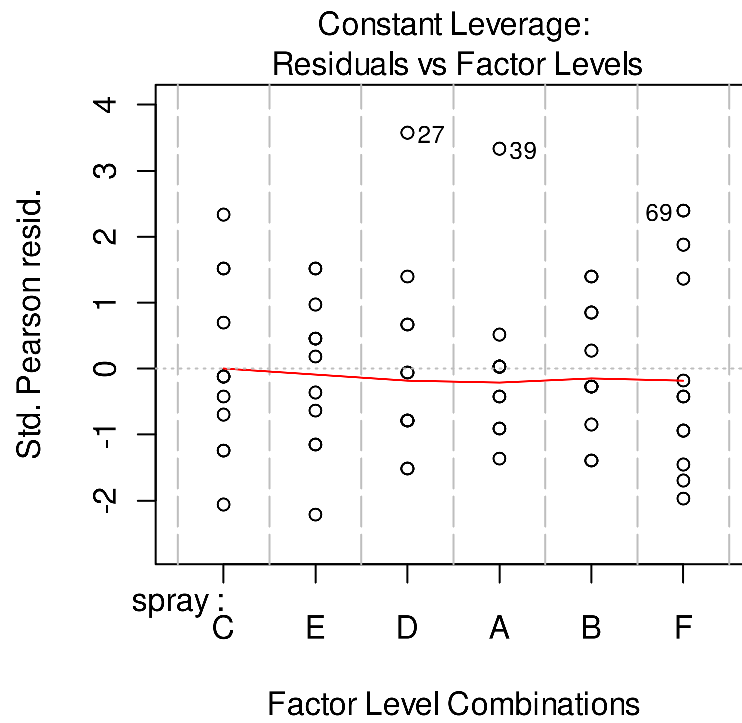
Let’s take a look at residuals.

```
> plot(mod.1)
```


Residuals vs Fitted
Residuals
Predicted values
glm(count ~ spray)
27
39
23
3
2
1
0
-1
-2
-3
1.0
1.5
2.0
2.5


Normal Q-Q
Std. deviance resid.
Theoretical Quantiles
glm(count ~ spray)
27
39
23
3
2
1
0
-1
-2
-2
-1
0
1
2


Scale-Location
√|Std. deviance resid.|
0.0
0.5
1.0
1.5
27
39
23
1.0
1.5
2.0
2.5
Predicted values
glm(count ~ spray)


Constant Leverage:
Residuals vs Factor Levels
Std. Pearson resid.
-2
-1
0
1
2
3
4
27
39
69
spray :
C
E
D
A
B
F
Factor Level Combinations

Exercises

1. Organism response to metal is often a function of the log 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 nearer to the city center. Make sure you take a look at the data to decide how to specify the model formula.

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.

13. Generalized additive models

Crawley 2008: Chapter 18

13.1. The gam function

Generalized additive models (GAMs) are a very flexible approach to 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.

$$g\left(E\left(Y\right)\right)=\beta_0+f_1\left(x_1\right)+f_2\left(x_2\right)...+f_m\left(x_m\right)$$

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.

Generalized additive models can be fit with functions from multiple packages. We use the `gam` function from the package `mgcv` package. As with `glm`, `gam` is a very flexible function, and the following treatment just scratches the surface.

We need to install and load the `mgcv` package.

```
> install.packages("mgcv")
--- Please select a CRAN mirror for use in this session ---
trying URL
'http://lib.stat.cmu.edu/R/CRAN/bin/windows/contrib/2.8/mgcv_1.4-1.1.zip'
Content type 'application/zip' length 1006854 bytes (983 Kb)
opened URL
downloaded 983 Kb

package 'mgcv' successfully unpacked and MD5 sums checked

The downloaded packages are in
        C:\Documents and Settings\Sasha\Local
Settings\Temp\Rtmp9psnyE\downloaded_packages
updating HTML package descriptions

> library(mgcv)
This is mgcv 1.4-1.1
```

Also install the package `gamair`, which contains the data sets used in Wood (2006).

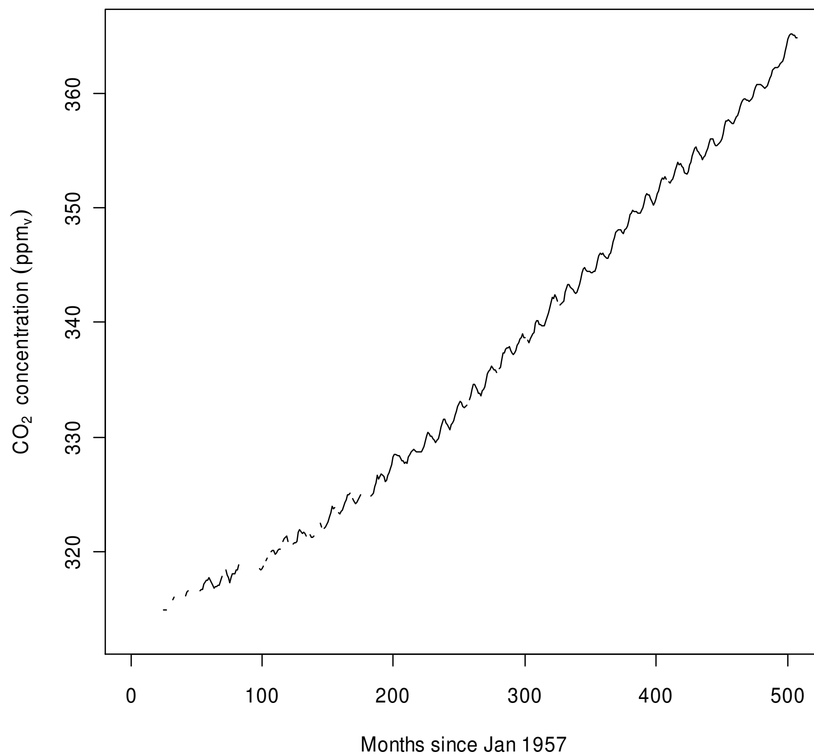
```
> install.packages("gamair")
> library(gamair)
```

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

> with(co2s, plot(co2 ~ c.month,type="l",xlab="Months since Jan
1957",ylab=expression(CO[2]~~"concentration"~~(ppm[v]))))
```



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.

```
> mod.1

Family: gaussian
Link function: identity

Formula:
co2 ~ c.month + s(month)

Estimated degrees of freedom:
 3.341923   total =  5.341923

GCV score:  3.033957

> 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 ***
---
```

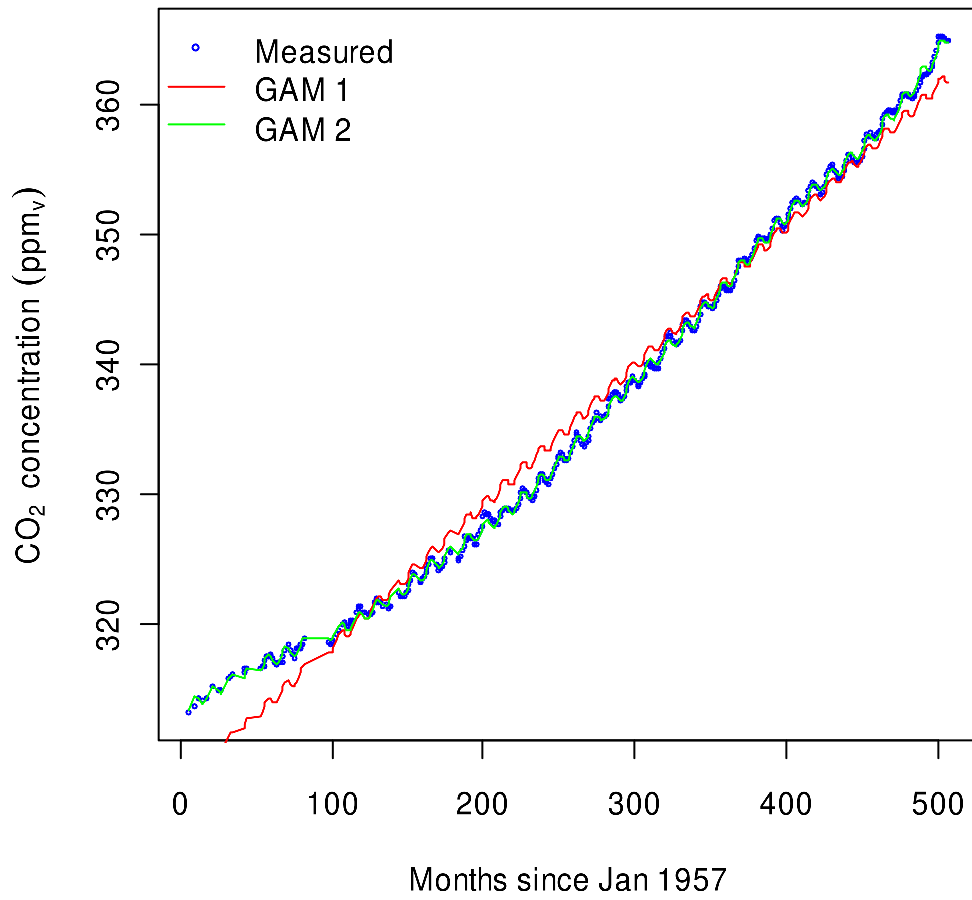
```
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

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.

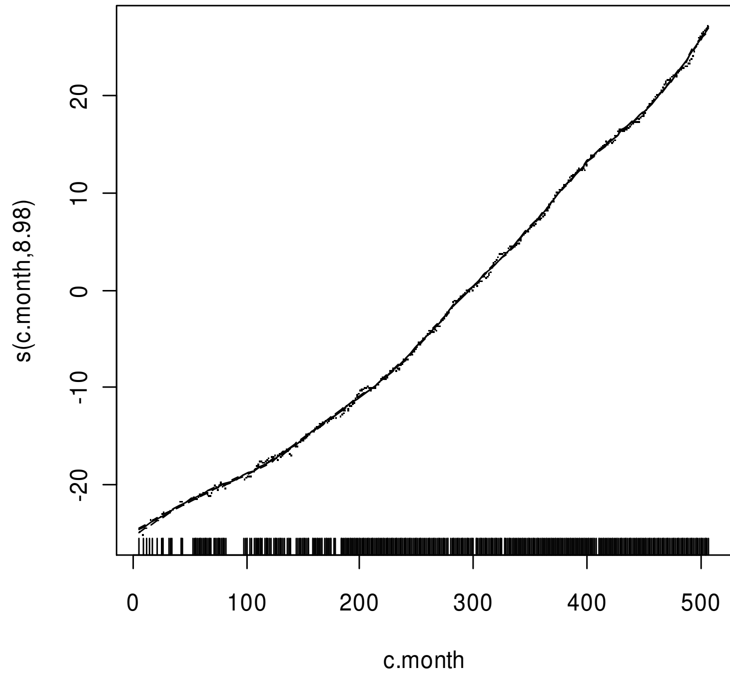
```
> mod.2<-gam(co2 ~ s(c.month) + s(month), data = co2.dat)
> summary(mod.2)

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
```

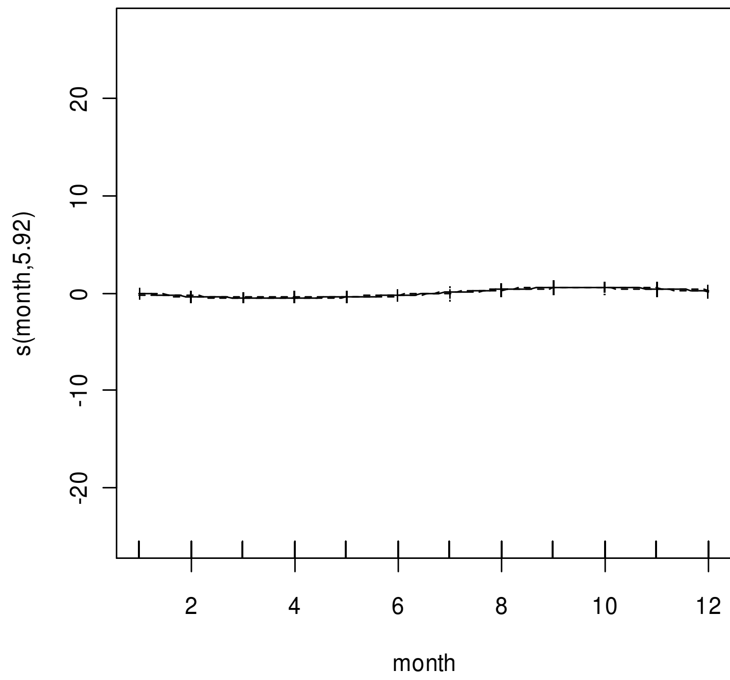
How do the models look?

```
> with(co2.dat, plot(co2 ~ c.month,pch=1,cex=0.4,col='blue',xlab="Months
since Jan 1957",ylab=expression(CO[2]~~"concentration"~~(ppm[v]))))
> points(co2.dat$c.month,predict(mod.1),type="l",col='red')
> points(co2.dat$c.month,predict(mod.2),type="l",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.

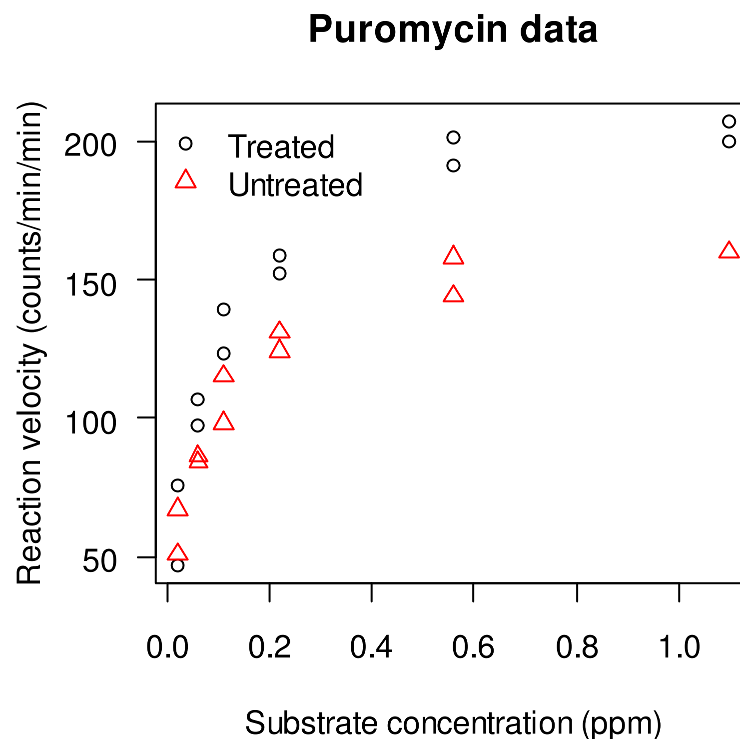
14. Nonlinear regression

Dalgaard 2008: Chapter 16

14.1 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 frame describes the reaction velocity of an enzymatic reaction with and without treatment with the antibiotic puromycin.

```
> attach(Puromycin)

> plot(rate ~ conc, las = 1, xlab = "Substrate concentration (ppm)", ylab
= "Reaction velocity (counts/min/min)", pch = as.integer(Puromycin$state),
col = as.integer(Puromycin$state), main = "Puromycin data")
> legend("topleft",c("Treated","Untreated"),pch=c(1,2),
col=c("black","red"),bty="n")
```



To model these data, let's use the Michaelis-Menten equation, which has an upper asymptote. The equation can be described as follows.

$$y = \frac{V_{\max} x}{1 + K_m x},$$

The asymptote $V_{\max}$ is the maximum reaction velocity, and K_m is the Michaelis-Menten constant. First we fit a model to the "treated" group—this is specified in the `subset` argument in the `nls` function call.

```
> mod.1<-nls(rate ~ Vm*conc/(K + conc), data = Puromycin,subset = state ==
"treated",start = c(Vm = 200, K = 0.05), trace = TRUE)
1636.586 :  2e+02 5e-02
1205.620 :  211.15721948   0.06162713
1195.573 :  212.51134162   0.06384178
1195.45 :  212.66623435   0.06409387
1195.449 :  212.68204569   0.06411864
1195.449 :  212.68357944   0.06412103
```

These second model uses different initial guesses of parameter values.

```
> mod.2 <- nls(rate ~ Vm * conc/(K + conc), data = Puromycin,
            subset = state == "untreated",
            start = c(Vm = 160, K = 0.05), trace = TRUE)
883.5846 :  160.00   0.05
859.7071 :  160.47706337   0.04798192
859.6064 :  160.30560335   0.04774760
859.6043 :  160.28376043   0.04771392
859.6043 :  160.28058775   0.04770902
859.6043 :  160.28012517   0.04770831
```

Let's look at a summary of the two different models.

```
> 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

> summary(mod.2)

Formula: rate ~ Vm * conc/(K + conc)

Parameters:
    Estimate Std. Error t value Pr(>|t|)
```

```
Vm 1.603e+02  6.480e+00  24.734 1.38e-09 ***
K  4.771e-02  7.782e-03   6.131 0.000173 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 9.773 on 9 degrees of freedom

Number of iterations to convergence: 5
Achieved convergence tolerance: 4.47e-06
```

We can extract parameter estimates from an `nls` object using the `coef` function.

```
> coef(mod.1)
         Vm            K
212.68357944   0.06412103

> coef(mod.1)[1]
      Vm
212.6836

> coef(mod.1)["Vm"]
      Vm
212.6836
```

Sometimes problems arise if the initial guesses for the parameter values are not reasonable—this is probably one of the most common reasons for failure of the `nls` procedure to converge. For some common models, 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:

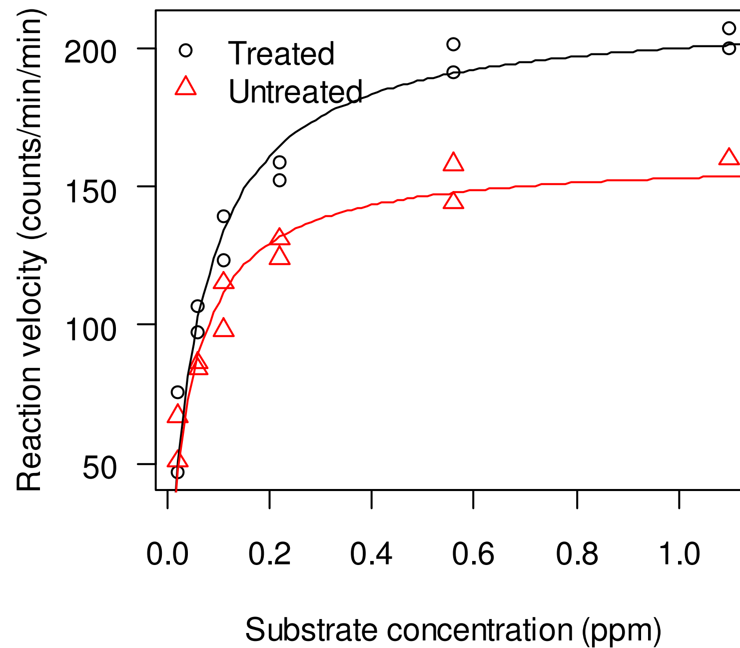
```
> mod.3<-nls(rate ~ SSmicmen(conc, Vm, K), data = Puromycin,subset = state
== "treated")

> mod.4<-nls(rate ~ SSmicmen(conc, Vm, K), data = Puromycin,subset = state
== "untreated")

> summary(mod.3)

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
```

```
> summary(mod.4)

Formula: rate ~ SSmicmen(conc, Vm, K)

Parameters:
    Estimate Std. Error t value Pr(>|t|)
Vm 1.603e+02  6.480e+00  24.734 1.38e-09 ***
K  4.771e-02  7.782e-03   6.131 0.000173 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 9.773 on 9 degrees of freedom

Number of iterations to convergence: 5
Achieved convergence tolerance: 3.942e-06
```

Comparing the results of the model fitting exercises, we see that the self-starting functions provided the same parameter estimates as the `nls` in which we had to specify initial guesses.

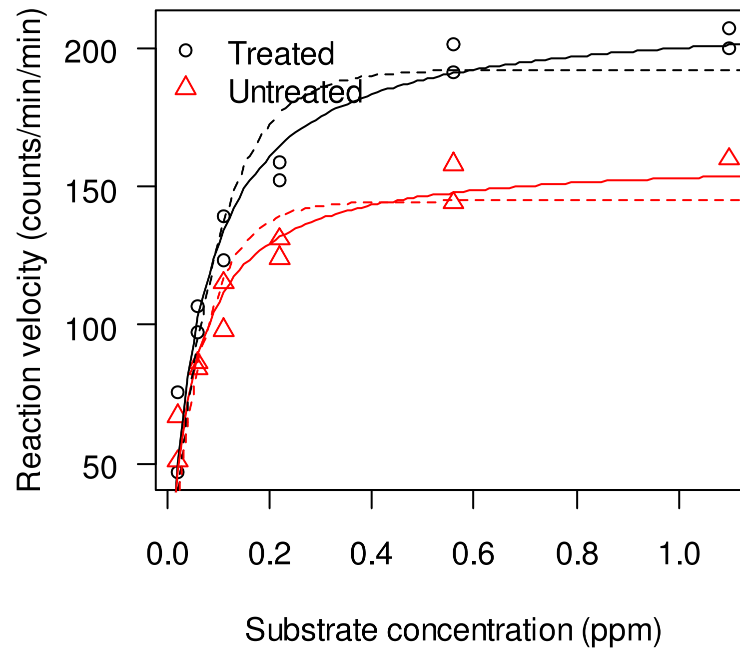
Now let's make some model predictions and show them on a new figure. We will use the same procedure for creating a new vector of x-values (substrate concentration) to use as the basis for our y-value predictions (reaction velocity).

```
> xv=seq(0,1.2,0.01)
> mod.3.yv=predict(mod.3,data.frame(conc=xv))
> mod.4.yv=predict(mod.4,data.frame(conc=xv))
> plot(rate ~ conc, las = 1, xlab = "Substrate concentration (ppm)", ylab
= "Reaction velocity (counts/min/min)", pch = as.integer(Puromycin$state),
col = as.integer(Puromycin$state), main = "Puromycin data")
> legend("topleft",c("Treated","Untreated"),pch=c(1,2),
col=c("black","red"),bty="n")
> lines(xv,mod.3.yv)
> lines(xv,mod.4.yv,col="red")
```



On a second plot, the results of another nonlinear model fit are shown, this time using the self-starting function for the 2-parameter asymptotic exponential model. This model does not appear to fit the data quite as well as the previous models. Here is the code for the fitting, and the addition to the existing plot:

```
> mod.5<-nls(rate ~ SSasympOrig(conc, Asym, lrc), data = Puromycin, subset
= state == "treated")
> mod.6<-nls(rate ~ SSasympOrig(conc, Asym, lrc), data = Puromycin, subset
= state == "untreated")

> mod.5.yv=predict(mod.5,list(conc=xv))
> mod.6.yv=predict(mod.6,list(conc=xv))
> lines(xv,mod.5.yv,lty=2)
> lines(xv,mod.6.yv,col="red",lty=2)
```



There are several common non-linear models that are useful, any of which can be applied using `nls` (Crawley 2007: Table 20.1, 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

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 λ is the decay constant. Calculate the half-life (ln(2)/c, but note that in R, natural log is called with `log`). Be sure to use the `coef` function. Plot the data and the model predictions.

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).

15. Grouping, loops, 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. However, interactive R sessions and simple scripts can also make use of these features.

15.1. Loops and grouping

Loops are a common feature in most programming languages—they allow you to repeat a command or a set of commands many times with a small amount of code. In R it is possible to avoid using loops for many procedures where they would be required in other languages via vectorized operations. Vectorized operations are more efficient than loops from both the standpoint of both writing and executing code. However, in some cases, loops are 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 (i in some.vector) print(i)
[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 that grouping is useful.

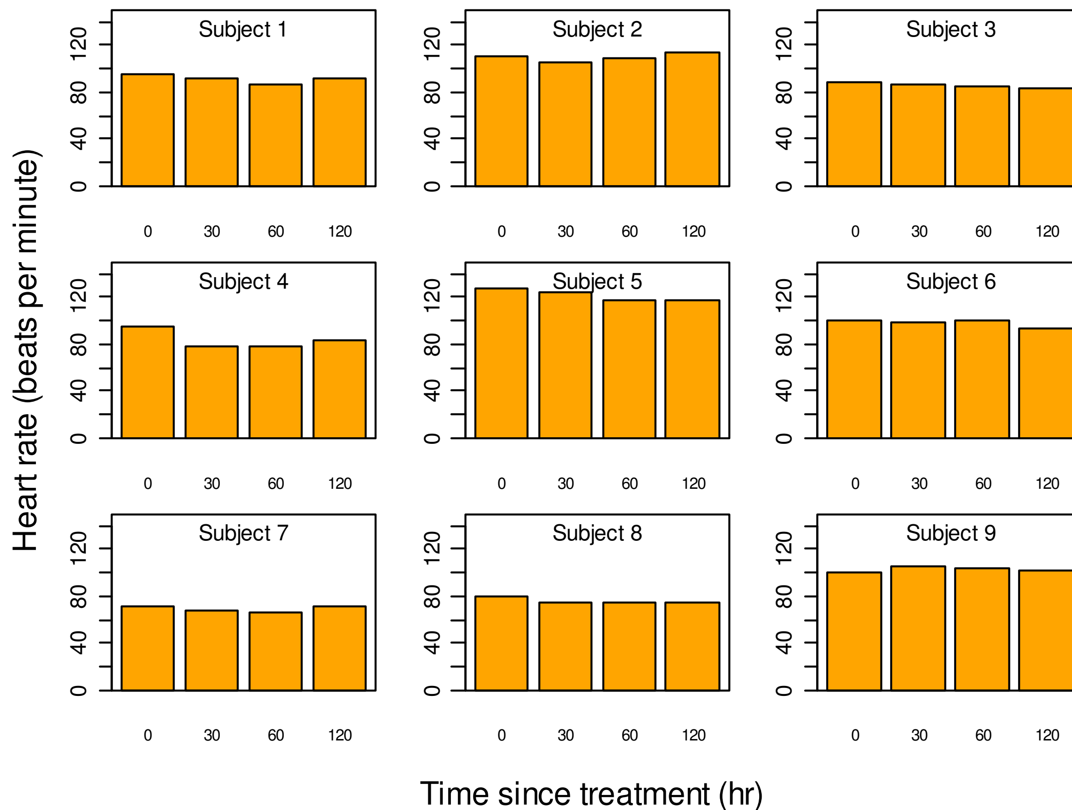
```
> orgs<-c("Tree","Grass","Snail")
> j<-0
> for (i in orgs) {
+   j<-j + 1
+   z<-j^2
+   print(c(j,z,i))
+ }
[1] "1"    "1"    "Tree"
[1] "2"     "4"     "Grass"
[1] "3"     "9"     "Snail"
```

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. This type of loop can often make use of the `level` function, which will return the levels of a factor.

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$subj)) {
       sub.dat<-subset(hr.dat, subj==i)
       barplot(sub.dat$hr,names.arg=sub.dat$time, xlab="",ylab="",
       ylim=c(0,150),cex.names=0.7,col='orange')
       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.splt.dat<-split(hr.dat,hr.dat$subj)

> for (i in levels(hr.dat$subj)) {
+ sub.dat<-hr.splt.dat[[i]]
+ barplot(sub.dat$hr,names.arg=sub.dat$time, xlab="",ylab="",
+ ylim=c(0,150),cex.names=0.7,col='orange')
+ 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)
```

This code should give you plots that are identical to those shown above.

You can nest loops within loops, as demonstrated below with the toxicity data used in the ANCOVA analysis above.

```
> lc50.dat<-read.table("Ogeechee_tox_summary.txt",header=TRUE)
> lc50.dat$dom.source<-factor(lc50.dat$dom.source)
> lc50.dat$n.ph<-factor(lc50.dat$n.ph)
```

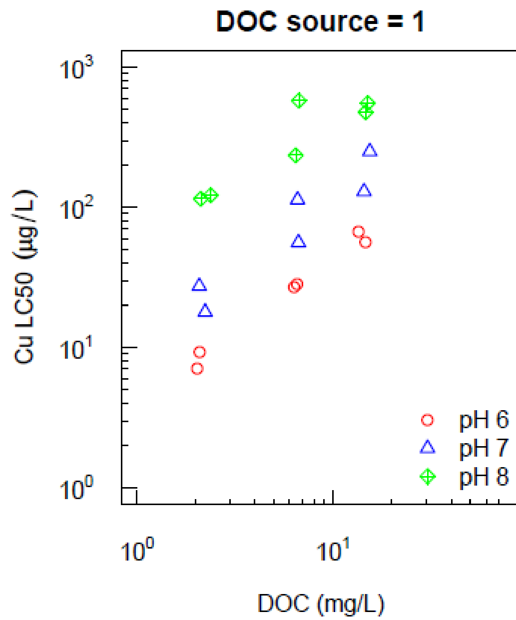
In this example, we use the levels of both `dom.source` and `n.ph` as counter variables via the `levels` function.

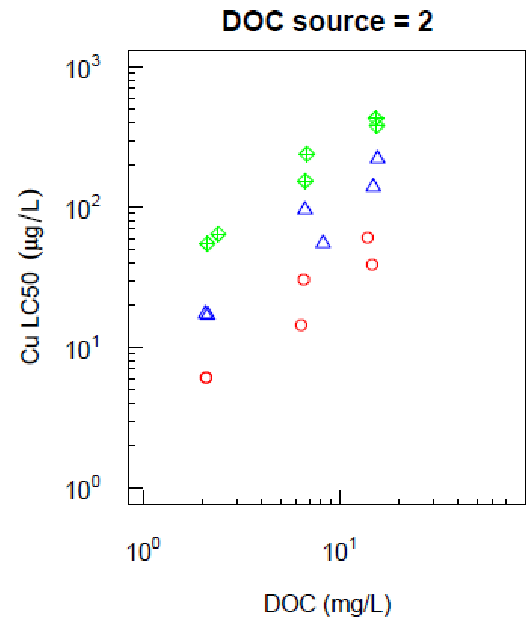
```
> levels(lc50.dat$dom.source)
[1] "1" "2" "3" "4" "5" "6" "7"
```

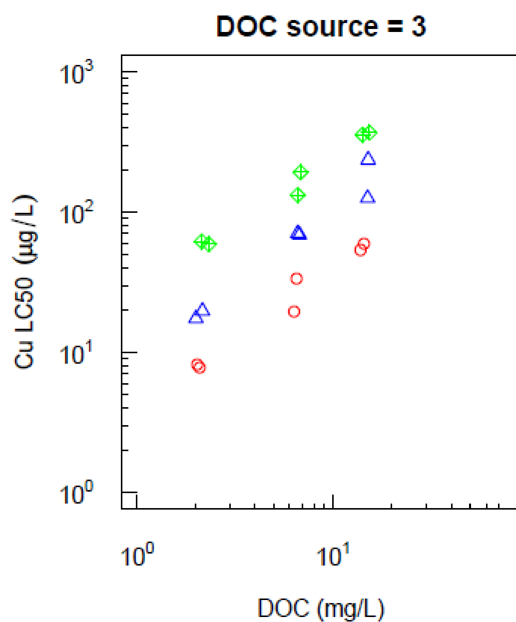
Let's set up a 4 panel plot in a pdf file.

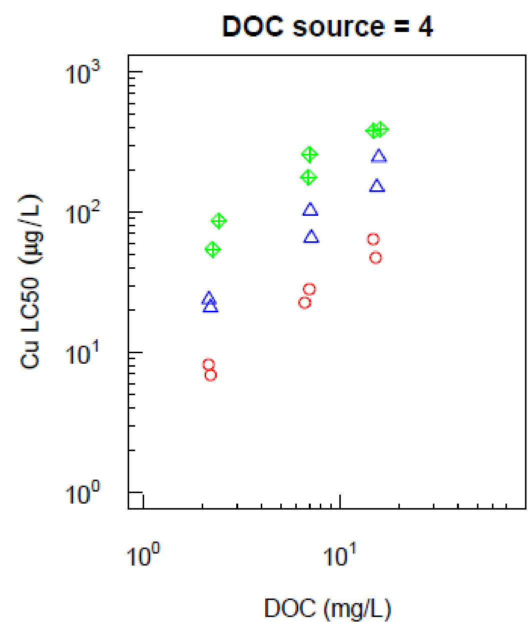
```
lc50.dat<-read.table("Ogeechee_tox_summary.txt",header=TRUE)
lc50.dat$dom.source<-factor(lc50.dat$dom.source)
lc50.dat$n.ph<-factor(lc50.dat$n.ph)

source("Functions.R")

pdf("Daphnia_plots.pdf",width=8.5,height=11)

       par(mfrow=c(2,2),mar=c(6,4,2,6),oma=c(8,3,10,0))

       p.cols<-c("red","blue","green")
       p.pch<-c(1,2,9)

       for(i in levels(lc50.dat$dom.source)[1:4]) {

              sub1.dat<-subset(lc50.dat,dom.source==i)

              plot(sub1.dat$doc,sub1.dat$diss.lc50,type="n",xlim=c(1,75),
                     ylim=c(1,1000),las=1,axes=F,      xlab="DOC (mg/L)",
                     log="xy",ylab=expression("Cu LC50"~~(mu*g/L)),
                     main=paste("DOC source =",i))
              logaxis(1,1,75);logaxis(2,1,1000)

              if(i==1) legend("bottomright",c("pH 6","pH 7","pH 8"),
                                    pch=p.pch,col=p.cols,bty="n")

              k<-0
              for (j in levels(sub1.dat$n.ph)) {
                     k<-k+1
                     sub2.dat<-subset(sub1.dat,n.ph==j)
                     points(sub2.dat$doc,sub2.dat$lc50,pch=p.pch[k],
                                   col=p.cols[k])
              }
       }
dev.off()
```

The pdf file should contain the following plots.









In addition to selecting subsets of the same variables, it is possible to select different variables in each pass of a loop.

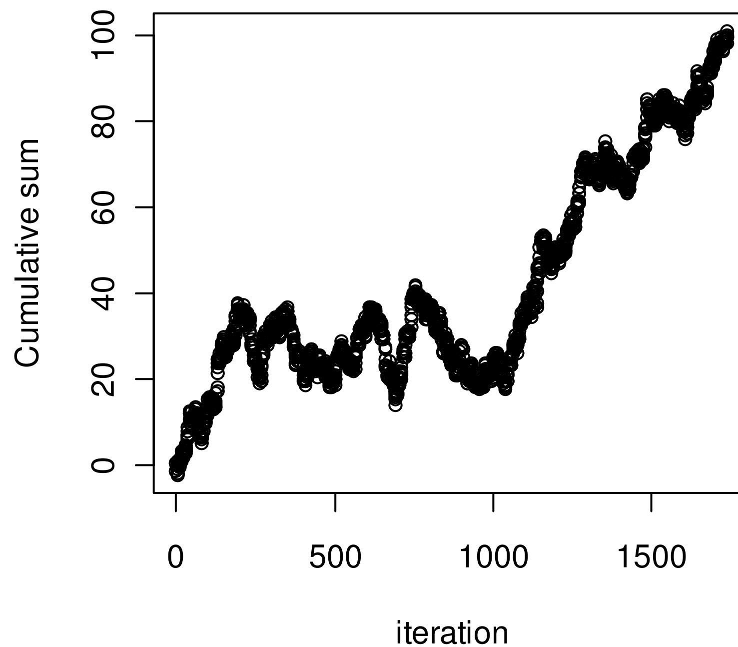
```
> mud.crk.dat<-read.table('Muddy_Crk.txt',header=TRUE)

> names(mud.crk.dat)
 [1] "site.no"          "date"             "discharge"        "discharge.code"
 [5] "max.temp"         "max.temp.code"    "min.temp"         "min.temp.code"
 [9] "DO.max"           "DO.max.code"      "DO.min"           "DO.min.code"
```

```
> names(mud.crk.dat)[c(3,5,7,9,11)]
[1] "discharge" "max.temp"  "min.temp"  "DO.max"    "DO.min"

> mud.crk.summ.dat<-
data.frame(row.names=names(mud.crk.dat)[c(3,5,7,9,11)])

> for (i in names(mud.crk.dat)[c(3,5,7,9,11)]) {
+ dat<-na.omit(mud.crk.dat)
+ mud.crk.summ.dat[i,'Smallest']<-min(dat[,i])
+ mud.crk.summ.dat[i,'Biggest']<-max(dat[,i])
+ mud.crk.summ.dat[i,'Mean']<-mean(dat[,i])
+ }

> mud.crk.summ.dat
          Smallest Biggest      Mean
discharge     13.0   915.0 85.735144
max.temp       2.4    22.3  8.655178
min.temp       0.4    20.0  6.950255
DO.max         3.5    12.6  9.108489
DO.min         2.1    10.6  7.836333
```

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.

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.

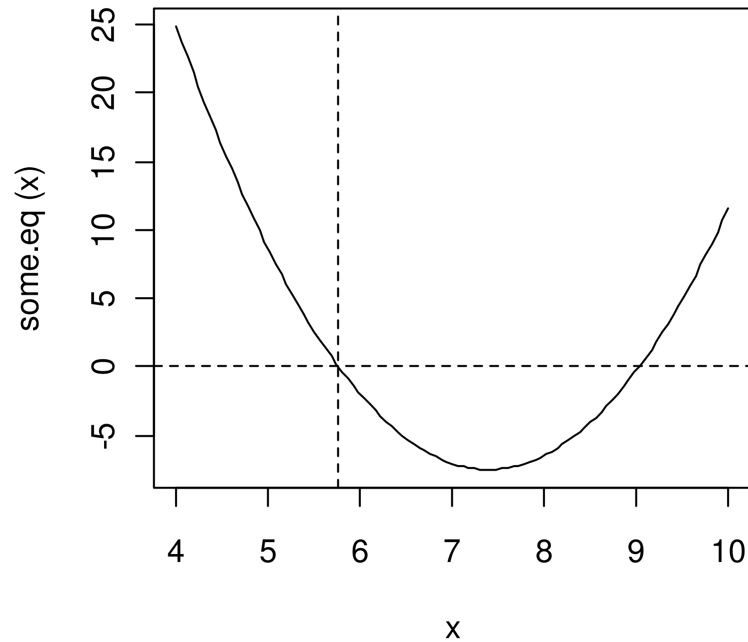
```
> par(mfrow=c(1,1))

> r.nums<-NULL
> i<-0

> while (sum(r.nums) < 100) {
+ i<-i + 1
+ r.nums[i]<-rnorm(n=1,mean=0.1,sd=1)
+ }

> plot(cumsum(r.nums),xlab="iteration",ylab="Cumulative sum")
```



Here is a more complicated (and useful) example that calculates a root of a specified equation. The object `some.eq`, created in the first line of code below, is a function that returns the value of a polynomial equation given a value for `x`—we will cover creating functions in more details in a later section.

```
> some.eq<-function(x) 1.72E-5*x^4 + 2.8*(x-7.4)^2 -7.527

> guess1<-1
> guess2<-1.001*guess1
> error<-100

> while (abs(error) > 1E-10) {
+ slope<-(some.eq(guess2) - some.eq(guess1))/(guess2 - guess1)
+ guess1<-guess2
+ guess2<-guess2 - some.eq(guess2)/slope
+ error<-some.eq(guess2) - 0
+ }

> guess2
[1] 5.76249

> some.eq(guess2)
[1] 9.884538e-12

> curve(some.eq,4,10)
> abline(h=0,lty=2)
```

```
> abline(v=guess2,lty=2)
```



Of course, this method is only going to give the root closest to your initial guess.

15.2. Conditional statements

Conditional statements are another ubiquitous feature of programming languages. In R, there are two conditional statements, `if...else` and `ifelse`. Although they seem similar, they are designed for different applications.

The `if` construct will execute a command if a specified condition is met.

```
> if (10>3) x<-101.8
> x
[1] 101.8

> if (10<3) x<-22.2
> x
[1] 101.8
```

If you would like several commands to be dependent on a single condition, then simply group the commands together.

```
> a<-100
> if (a>3) {
```

```
+ print(a)
+ a<-0
+ }
[1] 100

> a
[1] 0
```

The `if` construct can include an `else`, which should precede a command that should be executed if the original condition is not met.

```
> your.grade<-82

> if (your.grade > 65) result<-"pass" else result<-"fail"

> result
[1] "pass"
```

Another way to do this is by grouping the entire `if…else` construct.

```
> {if (your.grade > 65) result<-"pass"
+ else result<-"fail"}

> 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")
> results
 [1] "pass" "fail" "pass" "fail" "pass" "pass" "pass" "pass" "pass" "pass"
```

This construct is very 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
```

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. 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.

16. Distributions and simulations

Crawley 2007: Chapter 7, Dalgaard 2008: Chapter 3, Kuhnert & Venables 2005: pp. 38-40

16.1. Available distributions

Many of the distributions associated with statistical modeling have been built into R. There are nearly 30 such distributions, including the typically used normal, *t*, *F*, Chi-squared, exponential, gamma, beta, binomial, exponential, uniform, and Weibull. These distributions can be used for simulating data, determining quantiles, probabilities, and density functions. There are 4 prefixes that can be used with each distribution name, indicating the type of output that is desired. The prefixes are `p, q, d, r`, and they represent probabilities, quantiles, density, and random, respectively. Since all distributions can be used in the same way, we will only describe these in detail for the normal distribution. In the code below we use `pnorm` to calculate the probability of obtaining a value of 0 or less. The value that is returned is a cumulative probability. The two lines of code below return the same value, even though a mean and standard deviation were not specified in the second example (R defaults to a standard normal distribution if `mean` and `sd` are not specified by the user). Also note that the first argument is a quantile or a vector of quantiles, which simply represent the number of standard deviations from the mean.

```
> pnorm(0,mean=0,sd=1)
[1] 0.5
> pnorm(0)
[1] 0.5
>
```

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):

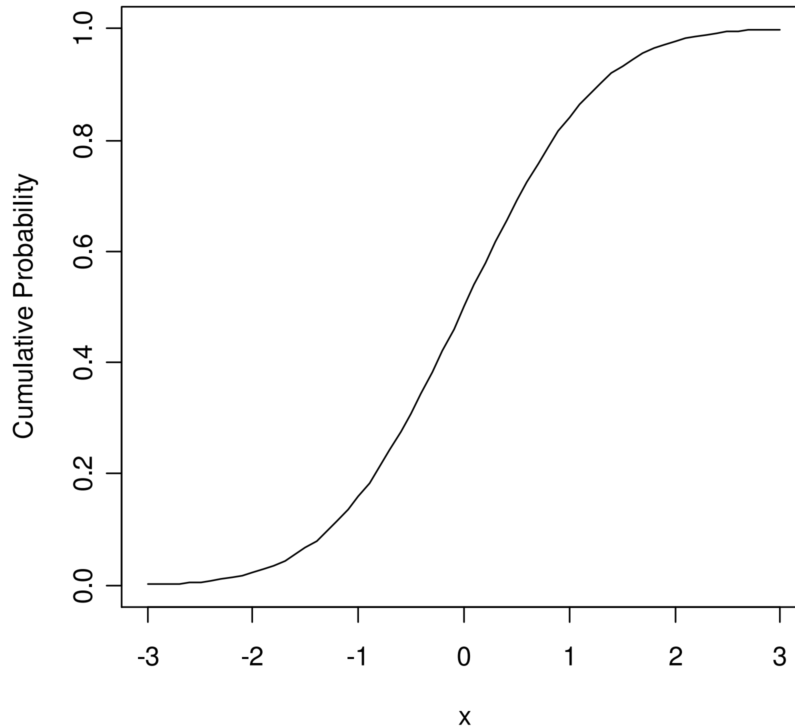
```
> pnorm(-3:3)
[1] 0.001349898 0.022750132 0.158655254 0.500000000 0.841344746
[6] 0.977249868 0.998650102
>
```

Suppose that we wanted to know the probability of obtaining a value that was within one standard deviation of the mean. This can be easily calculated by subtracting the cumulative probability of the higher quantile from the probability of the lower quantile:

```
> pnorm(1)-pnorm(-1)
[1] 0.6826895
```

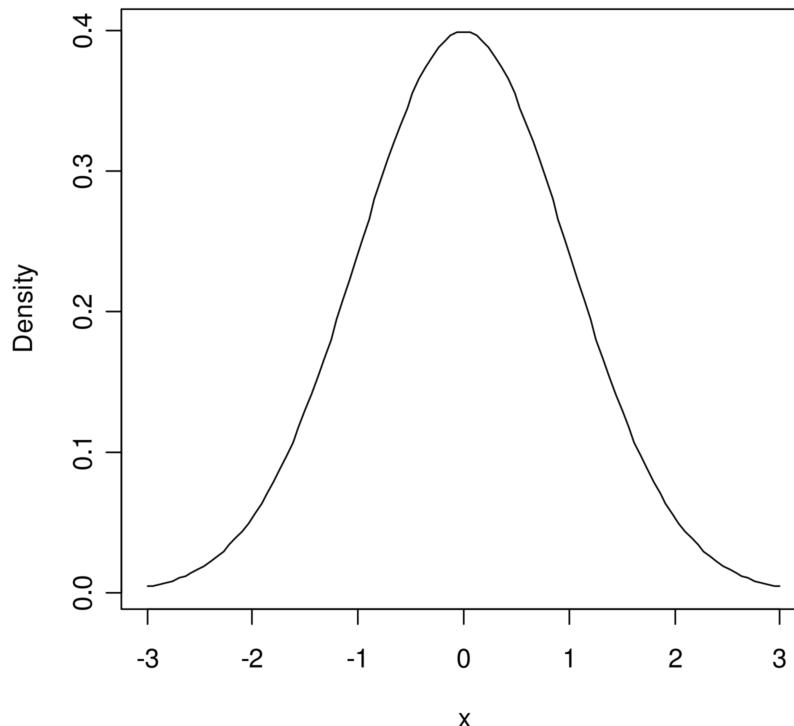
This shows that approximately 68.3% of the observations should be within 1 standard deviation of the mean. This is well known for the normal distribution. A cumulative probability plot for the standard normal distribution can be obtained by either of the following lines of code:

```
> plot(seq(-3,3,0.1),pnorm(seq(-3,3,0.1)),type="l",
     ylab="Cumulative Probability",xlab="x")
> curve(pnorm(x),-3,3,ylab="Cumulative Probability")
```



A probability density curve can also easily be constructed for the normal distribution using `dnorm`. 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) was 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
```

This provides the mean (or median from a normal distribution). To demonstrate that `pnorm` and `qnorm` are inverses of one another:

```
> pnorm(1)
[1] 0.8413447
> qnorm(0.8413447)
[1] 0.9999998
> qnorm(pnorm(1))
[1] 1
```

Also note that the last line of code demonstrates that the functions can be nested within one another.

Let's create a data set by randomly sampling from a normal distribution. To do this, we will use `rnorm`, which has 3 arguments (`n`, `mean`, and `sd`). Of the three arguments for `rnorm`, only the

first is required. In the following lines of code, we show how the resulting distribution of (pseudo-)random samples changes with the number of observations.

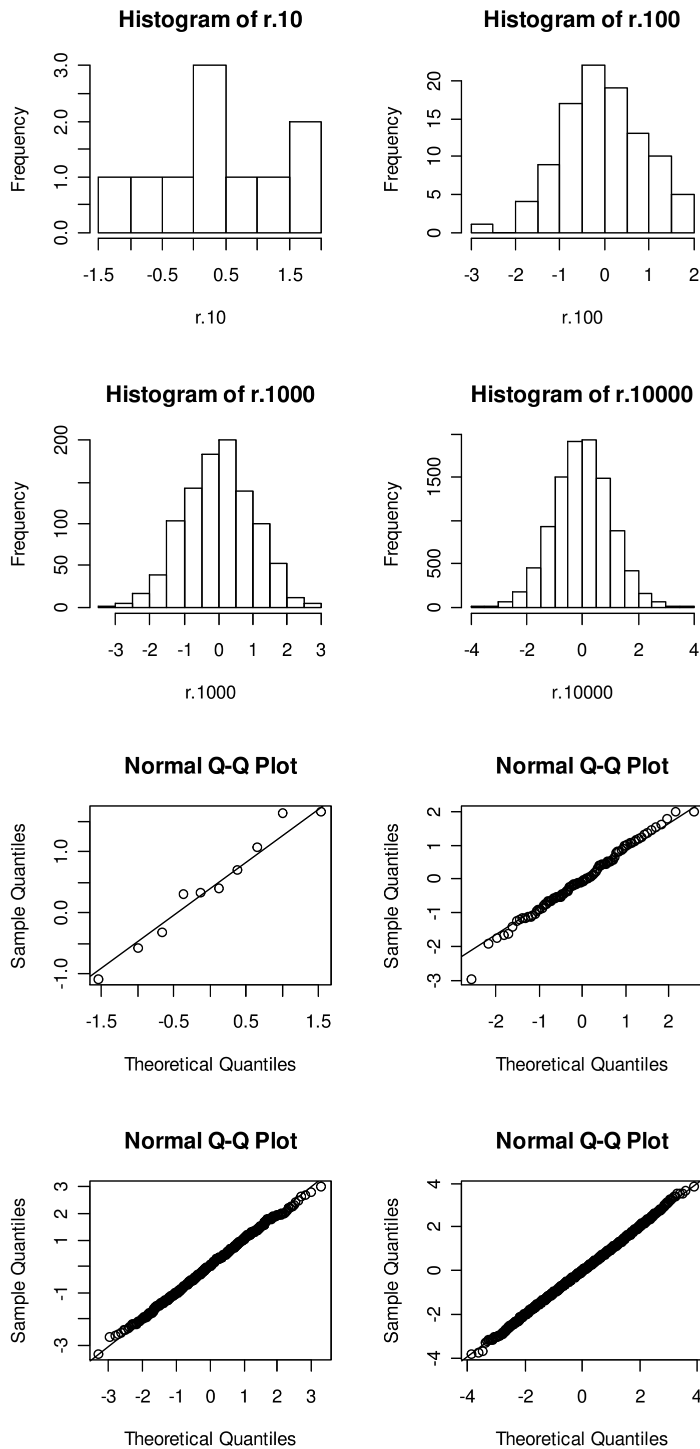
```
> r.10<-rnorm(10)
> r.100<-rnorm(100)
> r.1000<-rnorm(1000)
> r.10000<-rnorm(10000)
```

Type `r.10` to see what the 10 samples look like. They should vary around a mean of zero. Note that your output will not look exactly like the output below. Feel free to look at the output for the other vectors as well.

```
> r.10
 [1]  0.7010760  1.0855650  1.6327943  0.3270790 -1.0735387  0.3004477
 [7]  1.6461605  0.3915243 -0.3259995 -0.5777088
```

It is much easier to look at these results with `hist`, or in a `qqnorm`.

```
> par(mfrow=c(2,2))
> hist(r.10)
> hist(r.100)
> hist(r.1000)
> hist(r.10000)
>
> qqnorm(r.10)
> qqline(r.10)
> qqnorm(r.100)
> qqline(r.100)
> qqnorm(r.1000)
> qqline(r.1000)
> qqnorm(r.10000)
> qqline(r.10000)
```



Now let's consider a different distribution: the uniform distribution. First, we will see what the uniform distribution looks like. Here we were interested in displaying the quantiles associated with a uniform distribution that has a minimum of 10 and a maximum of 50.

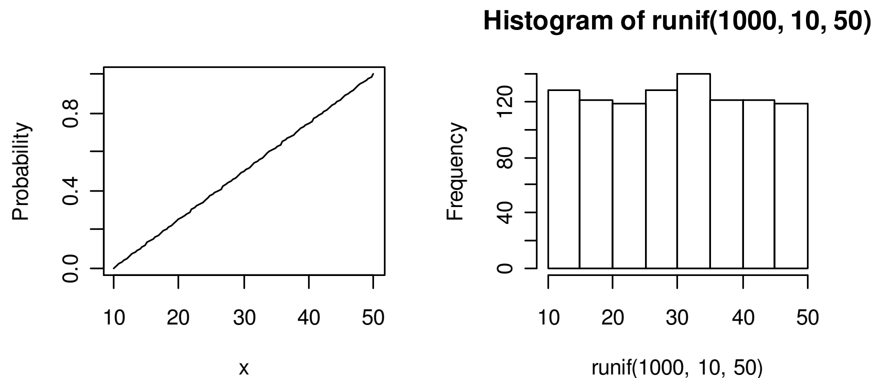
```
> qunif(seq(0.1,0.99,0.01),10,50)
 [1] 14.0 14.4 14.8 15.2 15.6 16.0 16.4 16.8 17.2 17.6 18.0 18.4 18.8 19.2
[15] 19.6 20.0 20.4 20.8 21.2 21.6 22.0 22.4 22.8 23.2 23.6 24.0 24.4 24.8
[29] 25.2 25.6 26.0 26.4 26.8 27.2 27.6 28.0 28.4 28.8 29.2 29.6 30.0 30.4
[43] 30.8 31.2 31.6 32.0 32.4 32.8 33.2 33.6 34.0 34.4 34.8 35.2 35.6 36.0
[57] 36.4 36.8 37.2 37.6 38.0 38.4 38.8 39.2 39.6 40.0 40.4 40.8 41.2 41.6
[71] 42.0 42.4 42.8 43.2 43.6 44.0 44.4 44.8 45.2 45.6 46.0 46.4 46.8 47.2
[85] 47.6 48.0 48.4 48.8 49.2 49.6
```

Now we are interested in seeing what the cumulative probabilities are for a sequence of quantiles, starting with a value of 10 and ending with a value of 50.

```
> punif(seq(10,50,0.5),10,50)
 [1] 0.0000 0.0125 0.0250 0.0375 0.0500 0.0625 0.0750 0.0875 0.1000 0.1125
[11] 0.1250 0.1375 0.1500 0.1625 0.1750 0.1875 0.2000 0.2125 0.2250 0.2375
[21] 0.2500 0.2625 0.2750 0.2875 0.3000 0.3125 0.3250 0.3375 0.3500 0.3625
[31] 0.3750 0.3875 0.4000 0.4125 0.4250 0.4375 0.4500 0.4625 0.4750 0.4875
[41] 0.5000 0.5125 0.5250 0.5375 0.5500 0.5625 0.5750 0.5875 0.6000 0.6125
[51] 0.6250 0.6375 0.6500 0.6625 0.6750 0.6875 0.7000 0.7125 0.7250 0.7375
[61] 0.7500 0.7625 0.7750 0.7875 0.8000 0.8125 0.8250 0.8375 0.8500 0.8625
[71] 0.8750 0.8875 0.9000 0.9125 0.9250 0.9375 0.9500 0.9625 0.9750 0.9875
[81] 1.0000
```

From this simple output, it appears that the cumulative probability is linearly related to the quantiles. This is exactly the case with this distribution. All observations are equally likely.

```
> curve(punif(x,10,50),10,50,ylab="Probability")
> hist(runif(1000,10,50))
```



Now that we know that the uniform distribution looks quite different than the normal distribution, we can use it to demonstrate the central limit theorem. The following example was modified from Crawley (2008).

First create a numeric vector of length 1000, using `numeric`, which creates a numeric object of the specified length. All elements of the created object have a default value of zero.

```
> m.10<-numeric(1000)
```

```
> m.10
   [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  [35] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  [69] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 [103] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
...
 [987] 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

We will populate this numeric vector with means calculated from samples drawn randomly from a uniform distribution. Each element in this vector will represent a mean of 10 randomly selected values. We will use the same uniform distribution as was used above: minimum value = 10 and maximum value = 50. For this example we will create a simple loop that calculates the ith mean and places it in the ith position in the vector of length 1000. So `i` will increase from 1 to 1000 in increments of 1. The loop structure to do this looks like:

```
> for (i in 1:1000) {
+       m.10[i]<-mean(runif(10,10,50))
+ }
```

Type `m.10` to see how the numeric object has changed. Each element of the vector is a mean calculated from 10 randomly selected samples from the uniform distribution.

```
> par(mfrow=c(1,1))
> hist(m.10)
```

The resulting distribution of means looks approximately normal, as expected. This would happen if means were calculated from random samples from any population with finite variance.

What would happen if we calculated more means? What would happen if we calculated fewer means?

16.2. Monte Carlo simulations

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.

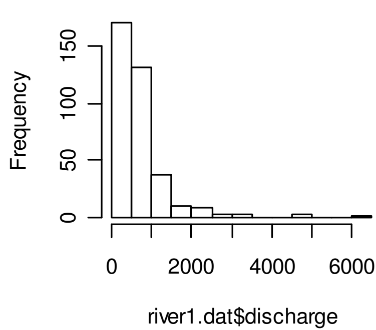
```
rivers.dat<-read.table("River_flow.txt",header=T)

> names(rivers.dat)
[1] "agency"         "site"           "date"           "discharge"
[5] "flag.discharge"
```

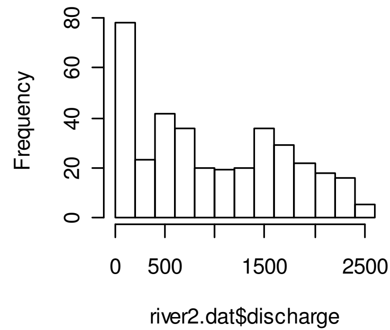
There are two different rivers in this dataset, so subset them so that we have 2 different dataframes. It has also been noticed that some of the observations are "NA", meaning that values were not reported.

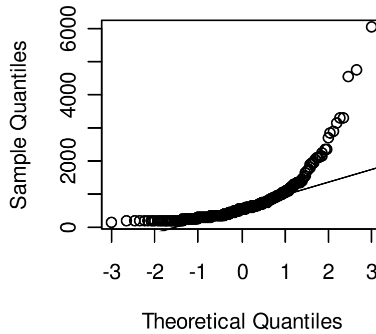
```
> river1.dat<-na.omit(subset(rivers.dat,site==1509000))
> river2.dat<-na.omit(subset(rivers.dat,site==4232730))

> 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
```

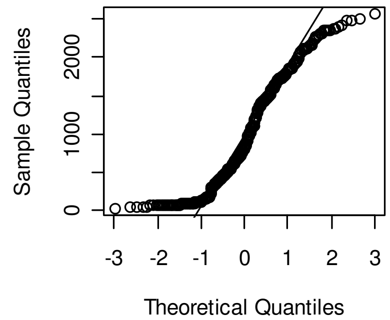
This result suggests that the mean flow from each river is in fact different. After conducting this t-test, you might wonder how appropriate this method is for this test, since you have assumed that the assumptions were not violated. In this case, the assumption that the data are normally distributed. 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)
```



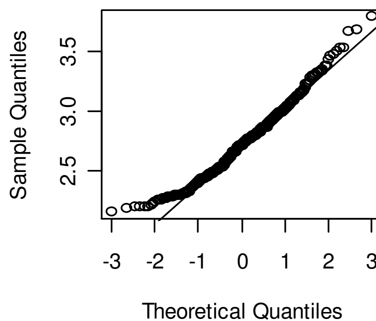


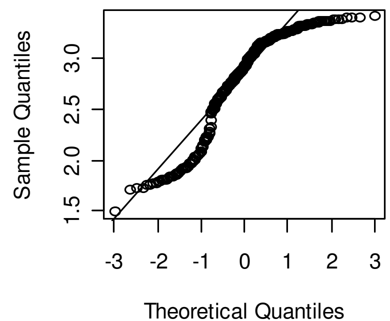




The data do not appear to be normally distributed. Perhaps the non-parametric alternative, the Wilcoxon signed rank test, would have been more appropriate, or maybe a log transformation might have helped. It looks like it would help with `river1.dat`, but maybe not with `river2.dat`. Log transform the 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.

16.3 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.

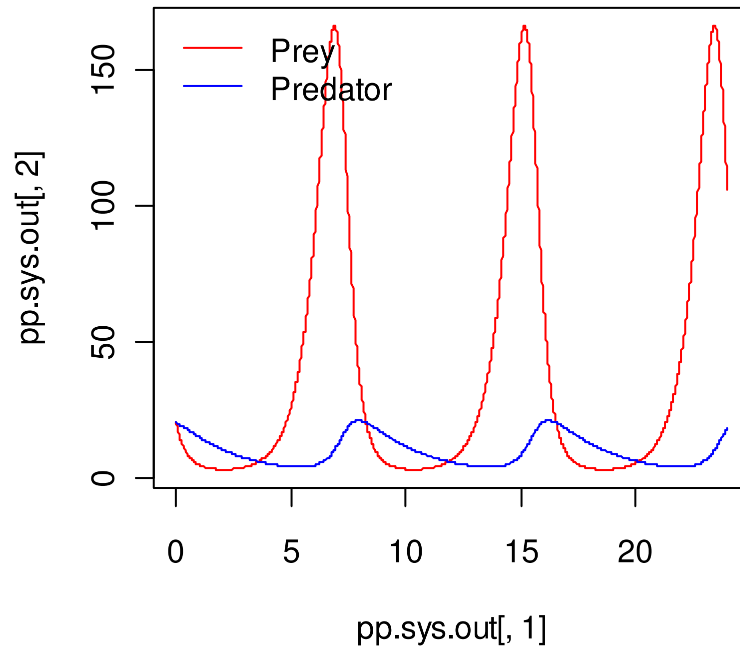
```
> install.packages("deSolve")
> library(deSolve)
```

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.

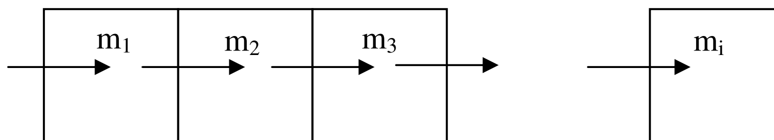
Let's model population growth of a predator-prey system using the Lotka-Volterra equations.

```
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:

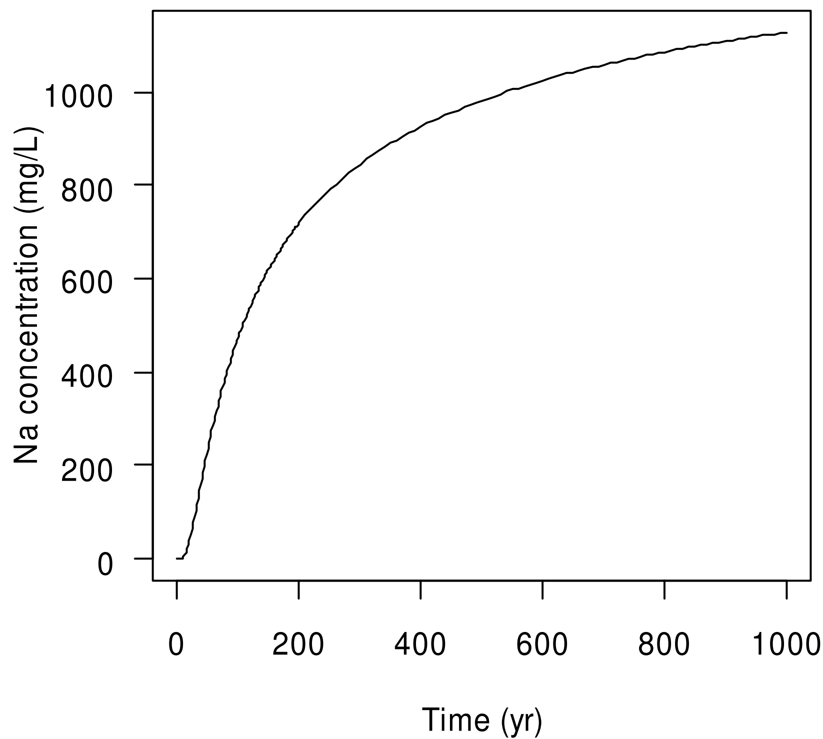
$$j = -D\frac{\Delta c}{\Delta x}$$

The concentration c is given by:

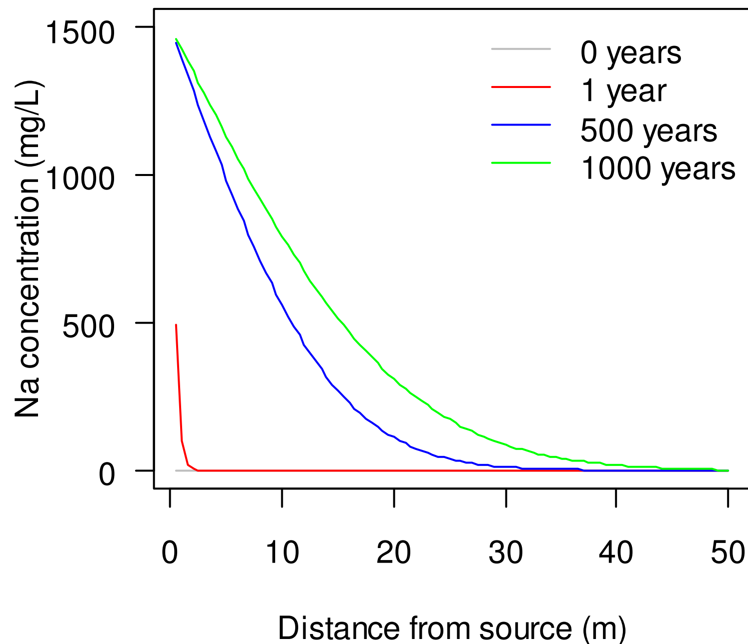
$$c = \frac{m}{width^3}$$

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 of no Na^+ at the far right.

```
# 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))

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")
points((w-0.5)*(1:dimens),na.diff.out[2,-1],type='l',col="red")
points((w-0.5)*(1:dimens),na.diff.out[57,-1],type='l',col="blue")
points((w-0.5)*(1:dimens),na.diff.out[62,-1],type='l',col="green")
legend("topright",c("0 years","1 year","500 years","1000
       years"),lty=1,col=c("gray","red","blue","green"),bty="n")
```

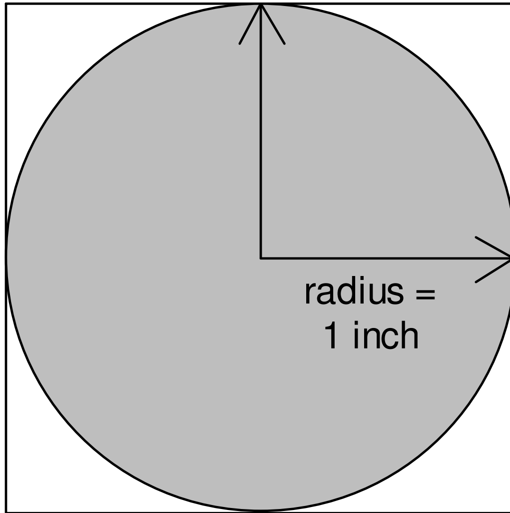


Exercises

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 2. The `quantile` function will be useful for summarizing the results of the simulation. For this example, assume exponential growth (i.e. $N=N_o e^{rt}$, 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 `lsodes`.

4. Create a two-dimensional version of the diffusion problem shown above.

17. Functions

Crawley 2007: 47, R-Intro: Chapter 10, R-Lang: Chapter 4)

17.1 Writing functions

One very useful feature of R is an ability to write your own functions. Once defined, these functions are stored internally, and 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 (this is very likely), you can use braces to group them.

```
> function_name<-function(arg1,arg2,arg3) {
+ expression1
+ expression2
+ expression3
+ }
```

Here is a simple example 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`. Note that 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
```

Compare this to `mean`.

```
> mean(x)
[1] 3700
```

Let's look at a more complicated example. Let's say we want a function that will calculate the % 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).

```
> exd.var<-function(meas,mod) {
+ tss<-sum((meas - mean(meas))^2)
+ e<-(mod - meas)^2
+ 1.0 - sum(e)/tss
+ }
```

Here our function name is `exd.var`, and it expects two arguments, `meas` and `mod`. In this function, there are expressions assigned to symbolic variables `tss` and `e`, which are them used in the final calculation. 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

> exd.var(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.

```
> e
Error: object "e" not found
> tss
Error: object "tss" not found
```

As with all the functions we covered, you can use both named and positional arguments.

```
> exd.var(hard.dat$hardness,hard.dat$hard.pred)
[1] 0.9493278
```

If you ever forget the arguments or their order, jus type the function name to see all its code.

```
> exd.var
function(meas,mod) {
tss<-sum((meas - mean(meas))^2)
e<-(mod - meas)^2
1.0 - sum(e)/tss
}
```

Or, you can use the `args` function.

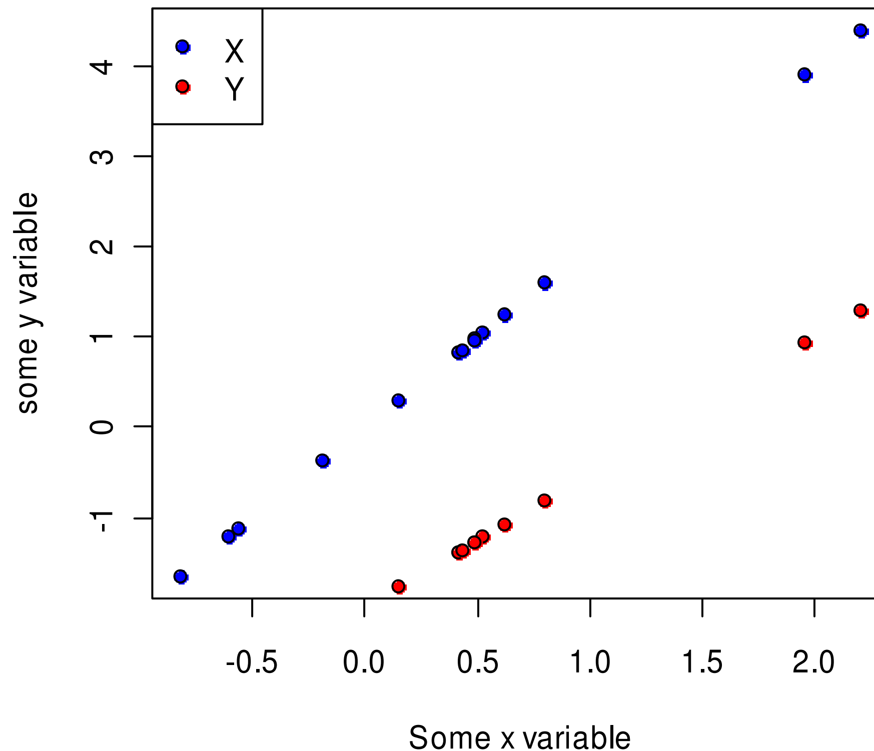
```
> args(exd.var)
function (meas, mod)
```

```
NULL
```

It is possible (and very common) to call up functions from within other functions. For example, let's say you want a modification of the `plot` function so that it 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.

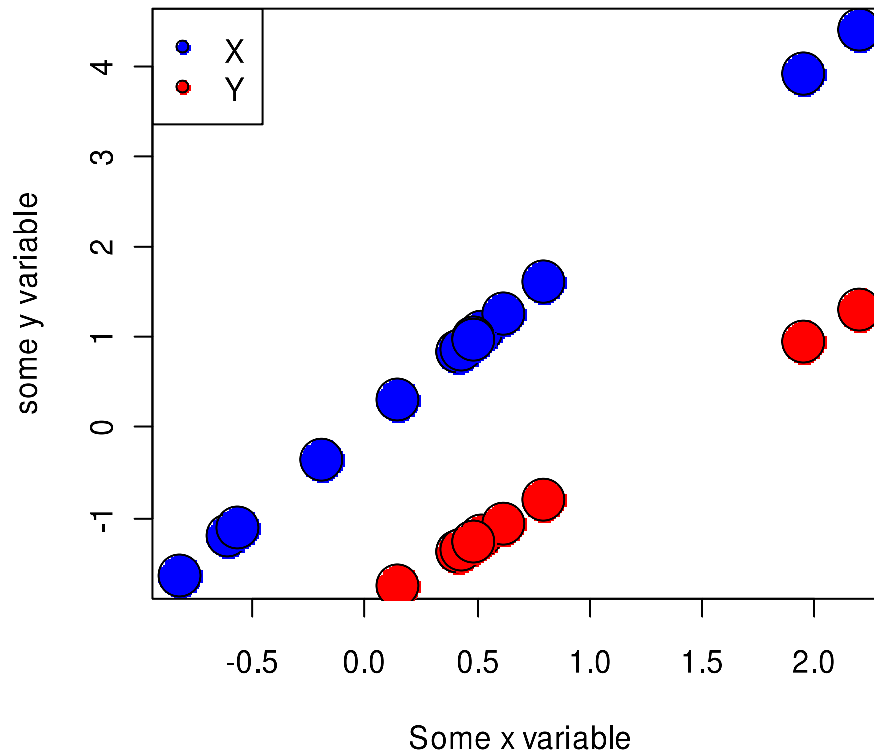
```
plot4me<-function(x,y,z,...) {
	plot(x,y,xlab="Some x variable",ylab="some y
	variable",pch=21,bg="blue",...)
	points(x,z,pch=21,bg="red",...)
	legend("topleft",c("X","Y"),pch=21,pt.bg=c("blue","red"))
}
```

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)
```



As mentioned above, R will attempt to put the additional arguments in any nested functions that have `...`, which may generate errors if `...` is present in more than one nested function call.

```
> 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

> mass<-rnorm(10,mean=175,sd=9)

> mass
 [1] 180.2066 175.0223 167.4249 167.5359 168.7322 165.2813 174.4403
175.7840 171.4901
[10] 167.9036
```

```
> lb2kg(mass)
[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 `alk2dic` function converts an alkalinity to a dissolved inorganic carbon concentration.

```
alk2dic<-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)

> alk2dic(alkalinity,ph)
[1] 0.0012445716 0.0012566845 0.0009355203
```

The flow control structions that were discussed earlier can, of course, be used in functions.

```
alk2dic<-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)

> alk2dic(alkalinity,pH)
[1] 0.0012445716 0.0012566845 0.0009355203
> alk2dic(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:

```
alk2dic<-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)
}

> alk2dic(c(100,700),c(7.5,9.3))
[1] 0.002138866          NA
```

Here is an example of a function that uses several of the features we discussed: `probplot`, which will make a cumulative probability plot.

```
probplot<-function(x,log=T,ylab='Value',ylim=NA,...) {
	x<-sort(na.omit(x))
	n<-NROW(x)
	if(log==T) {
		ll<-0.5*x[1]
		ul<-2*x[n]
		if(!is.na(ylim[1])) {
			ll<-ylim[1]
			ul<-ylim[2]
		}

	plot(qnorm(ppoints(x)),x,log='y',axes=F,xlab='',ylab=ylab,ylim=c(ll,ul),...)
		logaxis(2,ll,ul)
	}
	else {
		ll<-0.9*x[1]
		ul<-1.1*x[n]
		plot(qnorm(ppoints(x)),x,xaxt='n',xlab='',ylab=ylab,ylim=c(ll,ul),...)
	}
	axis(1,qnorm(c(0.0001,0.001,0.01,0.1,0.25,0.5,0.75,0.9,0.99,0.999,0.9999)),

	labels=c(0.01,0.1,1,10,25,50,75,90,99,99.9,99.99),las=1,tcl=0.3,mgp=c(0,0.2,0))
	mtext('Cumulative probability (%)',1,line=1.6)
}
```

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 (e.g. hundreds of lines of code), it can be very difficult to isolate a problem in your code, since the function acts like black box: you put in the arguments and get out the results. Typically, when you call up a function in R, you don't see any of the internal calculations. 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. This is demonstrated with the latest `alk2dic` function.

```
> debug(alk2dic)
```

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. This is probably easier done than seen—the example pasted below may be a bit difficult to follow.

```
> alkalinity<-c(55,62.1,45.1)
> ph<-c(7.23,8.10,7.75)

> alk2dic(alkalinity,ph)
debugging in: alk2dic(alkalinity, ph)
debug: {
    ifelse(alk < 500, {
        H <- 10^-pH
        K1 <- 10^-6.352
        K2 <- 10^-10.329
```

```
        2e-05 * alk * (((H/K1) + 1 + (K2/H))/(1 + 2 * (K2/H)))
    }, NA)
}
Browse[1]>
debug: ifelse(alk < 500, {
    H <- 10^-pH
    K1 <- 10^-6.352
    K2 <- 10^-10.329
    2e-05 * alk * (((H/K1) + 1 + (K2/H))/(1 + 2 * (K2/H)))
}, NA)
Browse[1]>
debug: H <- 10^-pH
Browse[1]>
debug: K1 <- 10^-6.352
Browse[1]> H
[1] 5.888437e-08 7.943282e-09 1.778279e-08
Browse[1]>
debug: K2 <- 10^-10.329
Browse[1]>
debug: 2e-05 * alk * (((H/K1) + 1 + (K2/H))/(1 + 2 * (K2/H)))
Browse[1]> K2
[1] 4.688134e-11
Browse[1]> H/K1
[1] 0.13243415 0.01786488 0.03999447
Browse[1]>
exiting from: alk2dic(alkalinity, ph)
[1] 0.0012445716 0.0012566845 0.0009355203
```

Exercises

1. Write a function for calculting the root mean square error (RMSE) of a set of model predictions. This is given by the following equation.

$$RMSE = \sqrt{\sum_{i=1}^{n}\left(x_{p,i} - x_{o,i}\right)^2}$$

where $x_{p,i}$ is ith predicted value, and $x_{o,i}$ is the ith observed value.

2. The density of puer water (in kg/L or g/mL etc.) 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.

3. Say you have a vector of names as single character strings, e.g. "Joahn Adams". Write a function that separates such a vector into two vectors, one with last names and one with first names. The functions `strsplit` might be handy here.

4. 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 scatterplot or a barplot.

18. Batch processing

R-Intro: Appendix B

18.1. 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")
	points((w-0.5)*(1:dimens),na.diff.out[2,-1],type='l',col="red")
	points((w-0.5)*(1:dimens),na.diff.out[57,-1],type='l',col="blue")
	points((w-0.5)*(1:dimens),na.diff.out[62,-1],type='l',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 a complex 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.

19. 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, 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 (if not all) 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. These include documents in several languages. After checking out Venables et al. (2008), you might want to check out these documents.

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*. Use 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.

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.

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.

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.

W. N. Venables, 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, Simon N. 2006. *Generalized Additive Models: An Introduction with R*. Texts in statistical science. Boca Raton, FL: Chapman & Hall/CRC.

Zar, Jerrold H. 1999. *Biostatistical Analysis*. 4th ed. Upper Saddle River, N.J: Prentice Hall.

Appendix: list of data files and their sources

We thank James Gibbs and Amy Roe 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 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 books were either entered manually or downloaded from associated websites.

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
Tortoise_length.txt	James Gibbs
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

Disclaimer:

The views expressed in this workbook do not necessarily represent the views of USDA or the United States.