

R Basics

R.D. Moore

2019-Jan-03

Contents

1	Introduction	2
2	What is R?	2
3	Running R	3
4	Writing R scripts	3
5	R Markdown	4
6	Finding help	5
7	Data types and structures within R	5
7.1	Variable types	5
7.2	Data objects	6
7.3	Digression on operators for assignment and basic arithmetic	6
7.4	Vectors and vectorized calculation	7
7.5	Matrices	9
7.6	Time series	10
7.7	Arrays	11
7.8	Data frames	11
7.9	Lists	12
7.10	User-defined objects	13
7.11	Finding out information about an object	13
8	Calculations and functions	14
8.1	Basic calculations and operators	14
8.2	Trigonometric functions	15
8.3	Mathematical functions	15
8.4	Statistical and related functions	15
8.5	Missing data	16
8.6	Relational operators and applications	17
8.7	Logical operators and applications	19
8.8	Functions for characters	20
8.9	Functions for factors	21
8.10	<code>ifelse()</code>	23
8.11	<code>approx()</code>	23
8.12	<code>expand.grid()</code>	24
8.13	<code>outer()</code>	25
8.14	User-defined functions	26
9	Overview of graphing packages/approaches	27
9.1	Base graphics	27
9.2	Lattice graphics	27
9.3	<code>ggplot2</code>	27
9.4	Comparing the three approaches	28

10 Case study - Data processing and graphing using base R	28
10.1 Data set	28
10.2 Application of base graphics to visualize the Alsea Watershed Study sediment yield data . . .	30

1 Introduction

Data analysis is an integral part of hydrology. Hydrologists frequently use techniques, such as regression analysis, which are incorporated into conventional statistical packages and spreadsheet software. However, many hydrological analyses are not, including intensity-duration-frequency analysis and flood frequency analysis. Hydrologists either need to seek stand-alone applications that perform these analyses, or gain the ability to use a programming language to implement these analyses. Moore and Hutchinson (2017) argued that, although programming languages such as Python and Julia may have advantages for some specific applications, the R language allows almost any conceivable hydrologic analysis to be performed with an integrated workflow.

The aim of the current document is to introduce R in more detail, and provide an overview of its features, functions and applications.

An effective way to work through the material below is to have R open and running, and to copy and paste the lines of code onto the command line to see the result.

2 What is R?

R is:

- a programming language that
 - is a dialect of the S programming language
 - was created by Ross Ihaka and Robert Gentleman
 - combines elements of procedural, functional and object-oriented programming
- an environment for data analysis and graphics
- part of the GNU (Gnu’s Not Unix) project

The R language is built around a set of packages, which are collections of functions and defined data types to perform specific tasks. Packages can also contain data sets that can be used to test or demonstrate R programs without reading in data from an external file.

The default installation, often called “base R,” includes more than a dozen packages, including:

- *base* – provides data types and functions for fundamental operations such as file input/output, calculations, looping, and conditional execution)
- *stats* – includes functions for descriptive statistics, linear and nonlinear statistical models, time series analysis
- *graphics* – provides a range of graphical tools for data visualization
- *datasets* – contains a variety of data sets, such as a digital elevation model of Auckland’s Maunga Whau volcano, passenger miles on commercial U.S. airlines, and speed and stopping distances of cars

A major strength of R is that a plethora of packages have been contributed to the project, many by leading statisticians and data scientists. Thus, R provides a rich set of tools for data analysis, simulation and visualization – all for free. To access functions in these packages, you need first to install the package, then load it into the workspace during a session or within a script.

The remainder of this document primarily covers functions available in base R. Later installments will cover alternative functions available in contributed packages such as *dplyr*.

3 Running R

R can be run in a number of ways. The two most popular are:

- entering code at the command prompt within R, either by
 - directly typing it or
 - by pasting in code that was typed into an editor (handy for entering multiple lines of code)
- using R Studio (or another graphical user interface) to edit and run scripts

4 Writing R scripts

When you write a script, not only should it generate correct results, but it should be readable and easy to interpret. Writing clear, understandable code is especially important if someone else will work with your script, or even if you will use the script at some later time. Readability is enhanced through a combination of (a) formatting and (b) commenting.

For detailed advice on formatting scripts, review the style guides developed by Google and Hadley Wickham. You should be able to find these by a simple query in a search engine. You should pay careful attention to the style guides in terms of naming conventions and use of spaces within lines of code.

To enter comments in a script, note that R does not execute anything on a line that follows a `#` symbol. Thus, if the first column of a line contains `#`, the entire line is interpreted as a comment. You can also add a comment following a short statement. See examples below.

```
# example of using an entire line for a comment
```

```
# the line below shows how to include a comment on the same line as code to be executed
z <- x - y      # this represents an informative comment on a line containing code
```

In the example above, `<-` is an assignment operator. That is, the numeric value of `x - y` is assigned to the variable named `z`.

It is good practice to include a header in your script, which should include an explanation of what the script is intended to do. I like to include the date and the name of the coder(s) who worked on the script. I also include dates and explanations of any revisions I have made to the script. It is common to generate multiple versions of a script (e.g., using variations on a method of calculation or analysis, or drawing upon different data sets), and inclusion of headers can assist greatly with “version control.” As you gain more experience coding, you can look into the use of [Github](#) and other approaches for tracking changes to code.

During an R session, you can access all data objects within your workspace. You can generate a list of the objects in your workspace by using the `ls()` command.

R can get confused if you create a variable with a given name, e.g. `x`, and then read in a data set that also contains a variable with the same name. It is good practice to begin any session or script by “removing” objects from your workspace using the `rm()` function. It is also good practice to remove objects from your workspace once they are no longer needed in a script. I always begin my scripts with `rm(list = ls())`, which removes all objects that are currently within the workspace.

Many of the R functions that you use will typically be in contributed packages rather than base R; you need to load contributed packages prior to using the functions within them. I like to load all libraries at the start of a script.

You will often keep all scripts and data files associated with a project within the same directory. It is usually convenient to define a “working directory” that R will look in by default when you wish to read in a data file or write output. This can be accomplished using the `setwd()` command. Note that R uses Unix-style forward slashes in path names (`/`), even on Windows-based systems.

The example below shows a typical layout for the top portion of a script. Note that you would not need to include comments for lines of code that are self-evident; the comments here are for the benefit of novice R users.

```
# example header for a script that does nothing
#
# 2017-Dec-29 RD Moore
#####

# clear workspace
rm(list = ls())

# load contributed packages to be used
library(reshape2)

# set the working directory
setwd("c:/Project name/Data files")

# read in the data as a comma-separated-value format file
dat <- read.csv("data_file_name.csv")

# then would follow lines of code to analyse the data
```

When writing code, it is imperative to recognize that R is case-sensitive. That is, C02 and co2 would be interpreted as different variables or objects.

5 R Markdown

R Markdown is a utility that is built into the R Studio application. It:

- is a simplified but very powerful mark-up language based on Markdown
- uses simple tags and formatting to control the rendering of a document
- allows R code to be embedded within the text, which is run when the R Markdown file is “knitted” (i.e., processed to create the formatted document).
- can be used to generate “dynamic documents,” in which figures, tables and specific values in the text will be automatically updated each time the file is knit.

R Markdown can be used to generate three types of output documents:

- Word documents
- Portable document format (pdf) files
- html files

To generate pdf files directly from R Markdown, you need to have LaTeX installed on your computer. A work-around is to generate a Word document, then convert that to a pdf document within the Word application.

R Markdown is a valuable research tool, especially for keeping a diary of your data analysis, because you can store both your code and comments on the analysis (e.g., how you processed your data) in one document.

Furthermore, extensions such as **bookdown** allow R Markdown to be used to generate complex reports, and even full-length books.

6 Finding help

Learning R can be a challenge. Fortunately, a range of resources are available, including a number of manuals that are available for free via links on the R web site (<http://cran.r-project.org/>).

Of particular interest to novice R users are *An Introduction to R* by Venables et al. (2013), *Using R for Data Analysis and Graphics - Introduction, Examples and Commentary* by Maindonald (2008), and the *R Reference Card* (Version 1 by Short, 2004; Version 2 by Baggett, 2012).

If you know the name of the command that you are seeking help about, you can view the built-in help page by using one of the following commands (used here for the function `lm()`):

```
?lm
help(lm)
```

If you do not know the name of the function, a broader search within the help pages can be conducted using key words. For example, use one of the two following commands to find functions with the term “skewness” in the help page:

```
??skewness
help.search("skewness")
```

Note, however, that the above searches will only find instances of “skewness”; they will not locate instances of “skew” in the help pages. In my experience, the built-in help pages are most useful to users after they have gained some experience with R.

The mailing lists (<http://www.r-project.org/mail.html>) provide another source of help. Users can send questions to the lists via email to elicit responses from the R user community. Be forewarned that many members of the R community can be blunt in their responses, especially when it is clear that the author of the question has not done his or her homework before posting. Be sure to read the posting guide (<http://www.r-project.org/posting-guide.html>) before sending any questions to the list.

An invaluable source of information is the Rseek search engine (<http://www.rseek.org/>), which will search a number of R-related sites, including the help pages and the mail lists.

There are many blogs that contain a wealth of information. One that I frequently end up at when I am searching for information is <https://www.r-bloggers.com/>. Click on the “LearnR” link near the top to access a number of well-designed and informative tutorials.

Another highly recommended reference is *R for Data Science* by Garrett Grolemund and Hadley Wickham, the latter of whom is the lead developer of a number of great packages, such as `dplyr` and `ggplot2`. You can access the e-book for free via <http://r4ds.had.co.nz/>. The book is particularly good for aspects of data wrangling.

Another book by Hadley Wickham, for those wanting to get deeper into advanced programming and applications in R, is *Advanced R*, accessible via <http://adv-r.had.co.nz/>.

I often use a search engine using a text string like “*r question to be asked.*” For example, if I were trying to figure out how to use Greek letters on graph labels I might search on the following string: “r greek letters in graphs”. This approach rarely fails to lead to useful information.

7 Data types and structures within R

7.1 Variable types

R supports a range of data types. The main types within Base R follow:

- numeric

- factor (for categorical variables)
 - unordered (e.g., tree species)
 - ordered (e.g, top 25%, middle 50%, bottom 25%)
- character (for character strings such as names)
- logical (true/false, 1/0)
- date-time

In addition, more complex data types are defined within packages. For example, the `sp` package defines objects for spatial data, such as `SpatialPoints` and `SpatialPolygons`.

7.2 Data objects

R is an object-oriented language, and supports a range of objects for storing data values and the results of analyses. The most common basic types are:

- vector
- matrix
- time series
- array
- data frame
- list

A feature of object-oriented languages is that each object type not only stores data, but is also associated with a set of “methods” for operating on the data. For example, most objects have a `plot` method, which behaves differently depending on the object type.

As you work with R, you will likely come across references to **S3** and **S4** objects. **S3** objects were defined within version 3 of the S language. The examples provided above are **S3** objects.

S4 objects were, not surprisingly, defined within version 4 of the S language. An example of **S4** objects that are commonly used by hydrologists are the objects defined for spatial data within the `sp` package, including `SpatialPoints` and `SpatialPolygonsDataFrame`.

7.3 Digression on operators for assignment and basic arithmetic

“Assignment” refers to the operation in which a variable is assigned a specific value. Different programming languages use different symbols for this operation. Within R, both equal sign (=) and a left-arrow (<-) can be used, although most R programmers favour <-.

I tend to use = out of habit, as I began programming in the 1970s with FORTRAN, which uses = for assignment. Also, = only requires one key stroke rather than the three required for <-. If you are just starting out, I suggest you use <- to be consistent with the current standard.

```
x = 80/3
x <- 80/3
```

Note that a rightward assignment operator is available, as in the following example, but is rarely used and is not recommended.

```
80/3 -> x
```

The basic arithmetic operators used in R are essentially the same as in Excel (see table below), with some important exceptions to be considered later.

Operator	Operation
+	addition

Operator	Operation
-	subtraction
*	multiplication
/	division

7.4 Vectors and vectorized calculation

Vectors

- are collections of values, all of which must be the same variable type (e.g., numeric or character)
- can be created using the `concatenate` function, `c()`. For example, the code below creates a vector named `x` with four elements (2, 4, 6 and 8).

```
x = c(2, 4, 6, 8)
x  # output the value to the console
```

7.4.1 Built-in constant vectors

The following vectors with constant values are defined in R.

```
letters
LETTERS
month.name
month.abb
pi
```

7.4.2 Extracting individual values from vectors

Individual values can be extracted from a vector by their index values, as in the examples below. Typing the name of a vector on the command line, followed by the return key, results in a listing of the values in a vector. Using a negative index results in the corresponding value being dropped.

```
x[1]
x[4]
x[-1]
x[-4]
month.name[6]
```

7.4.3 Generating sequences of values

In the examples below, two approaches are used to generate sequences of integers, the `seq()` function and `i:j`. The latter generates a sequence of integers beginning with `i` and ending with `j`.

```
seq(1, 13, 1)
6:8
-6:-8
-(6:8)
-6:8
```

One can create non-integer sequences using `seq()`, as in the following example, which generates a sequence ranging from 0 to 1, with an interval of 0.1.

```
seq(0, 1, 0.1)
```

More complex sequences with repeating values can be generated using the `rep()` function.

```
x = rep(1:3, each = 3)
x
y = rep(1:3, times = 3)
y
```

7.4.4 Extracting sets of values from a vector

Vectors containing multiple index values can be used to extract the corresponding values from another vector. Using a negative value for one or more index values causes the corresponding values in the vector to be left out.

```
i = seq(1, 13, 1)
a2m = letters[i]
a2m

i.even = seq(2, 26, 2)
even.letters = letters[i.even]
even.letters
odd.letters = letters[-i.even]
odd.letters

j = 6:8
summer.months = month.name[j]
summer.months

month.name[-(6:8)]
```

7.4.5 Vectorized calculations

A powerful feature of R (and many other programming languages) is the implementation of *vectorized arithmetic*, in which calculations are conducted elementwise. That is, for example, if `x` and `y` are vectors of length 3, then the following relations hold (note that the “=” here is mathematical equality, not assignment).

```
x + y = c(x[1] + y[1], x[2] + y[2], x[3] + y[3])
x - y = c(x[1] - y[1], x[2] - y[2], x[3] - y[3])
x*y = c(x[1]*y[1], x[2]*y[2], x[3]*y[3])
x/y = c(x[1]/y[1], x[2]/y[2], x[3]/y[3])
```

Before running the code below, try calculating the results by hand to ensure you understand the concept of vectorized calculations.

```
x = c(2, 4, 6, 8)
y = c(1, 3, 5, 7)
x + y
x - y
x*y
x/y
x[1:3]
y[1:3]
x[1:3]*y[1:3]
x[1]*y
x[1:2]*y
x[1:3]*y
```


The last three examples illustrate what happens if the vectors are different lengths. In these cases, the values of the shorter vector are “recycled” to match the length of the longer vector. If the length of the longer vector is not a multiple of the length of the shorter vector, a warning message is displayed.

7.5 Matrices

Matrices are two-dimensional structures in which all elements have the same type.

For example, gridded spatial data, such as raster digital elevation models, can be represented as a matrix `z[i, j]`, where `i` indexes the x direction and `j` the y direction. R has functions for analysing gridded data (e.g., filtering, smoothing), which we will learn about in a later document.

Many statistical analyses (e.g., linear regression and principle components analysis) involve matrix algebra (e.g., finding inverse of a matrix). R supports all major functions in matrix algebra.

Similarly to vectors, values in matrices can be extracted by index values.

```
y[i, j] # to access the ith row and jth column of a matrix
y[, j]  # to access the jth column of a matrix
y[i, ]  # to access the ith row of a matrix
```

For illustration, the following code creates a matrix of randomly generated values drawn from a uniform distribution (ranging from 0 to 1) with 3 rows and 5 columns. By default, the matrix is filled column by column.

```
x = runif(15, 0, 1)  # uniform random numbers between 0 and 1
x
y = matrix(x, nrow = 3)
y
y[2, ]
y[, 2]
y[2, 2]
y[-2, ]
y[, -2]
y[-2, -2]
```

The rows and columns in a matrix can be assigned names, which can be extracted or set using the `rownames()` and `colnames()` functions. The following code illustrates these functions applied to the matrix `y` that was generated in the previous code sample. Note the use of `nrow()` and `ncol()` to determine the numbers of rows and columns in the matrix.

```
# set row and column names -- note use of paste0() function, to be covered in more detail later
rownames(y) = paste0("Row_", seq(1, nrow(y)))
colnames(y) = paste0("Column_", seq(1, ncol(y)))
rownames(y)
colnames(y)
y
```

Rows or columns can be extracted by name, as well as by index. This can be handy if you manipulate a matrix, and the number of rows or columns, or their order, changes.

```
y[, 'Column_1']
y['Row_1', ]
str(y['Row_1', ])
```

7.6 Time series

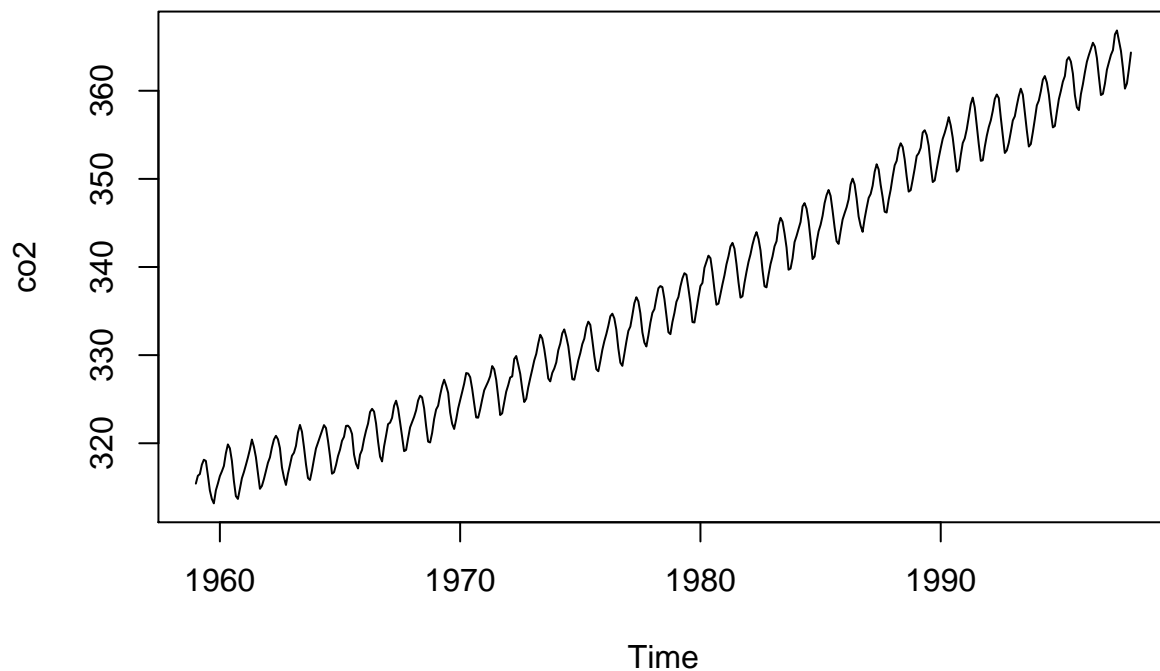
Time series objects contain a vector of values observed at a regular time interval (for univariate time series), as well as information about the start and end times and the frequency of observation. Time series objects for multiple time series contain a matrix in place of a vector, in which the columns represent the different variables and the rows are the times.

The time series objects include not just the data objects, but also methods for plotting the time series, printing, and extracting the time of the first and last observations and the frequency of observation.

An example in the `datasets` package is the monthly mean concentrations of atmospheric CO₂ as measured at the Mauna Loa observatory in Hawaii. The data object is named `co2`.

The script below plots the time series, extracts the vector of CO₂ concentrations and generates vectors for year and month to accompany the vector of CO₂ concentrations.

```
plot(co2) # generate a time series plot
```



```
C02 = as.numeric(co2) # extract vector of numeric values  
head(C02) # display first few values
```

```
## [1] 315.42 316.31 316.50 317.56 318.13 318.00
```

```
start(co2) # starting year and month for time series, as a vector of length 2
```

```
## [1] 1959 1
```

```
end(co2) # end year and month for time series, as a vector of length 2
```

```
## [1] 1997 12
```

```

y1 = start(co2)[1]           # extract start year
y2 = end(co2)[1]            # extract end year
ny = y2 - y1 + 1            # number of years of data
year = rep(y1:y2, each = 12)
month = rep(1:12, times = ny)
year[1:24]

## [1] 1959 1959 1959 1959 1959 1959 1959 1959 1959 1959 1959 1959 1959 1960 1960
## [15] 1960 1960 1960 1960 1960 1960 1960 1960 1960 1960 1960 1960

month[1:24]

## [1]  1  2  3  4  5  6  7  8  9 10 11 12  1  2  3  4  5  6  7  8  9 10 11
## [24] 12

```

7.7 Arrays

Arrays are multi-dimensional structures, in which all entries have the same type (e.g., numeric). A matrix is a special case of an array with two dimensions.

Arrays can be useful for spatio-temporal data. For example, mean monthly air temperature measured at three sites, for 12 months, over 10 years could be represented as `T[site, month, year]`. `T[2, 6, 7]` would refer to the air temperature at site 2 for June of the seventh year in the data set.

Arrays can also be useful for remote-sensing imagery. Two of the dimensions would represent the spatial coordinates, and the third could represent spectral band (for multi-spectral imagery) or time (e.g, for series of images of a single spectral band or an index such as NDVI).

7.8 Data frames

Data frames are two-dimensional data structures (rows by columns) in which

- each column is a vector representing a variable
- rows represent cases

Columns in a data frame must have the same length but, unlike a matrix, can be different data types. For example, one column could be a date-time variable, another could be a character variable (e.g., site name), and one or more could contain numeric vectors representing different observed variables.

Matrices can be converted to data frames using `as.data.frame()`, and data frames can be converted to matrices using `as.matrix()`.

The conversion from matrix to data frame will preserve the data type. That is, all columns in the data frame will have the same type as the values in the matrix.

However, the conversion from data frame to matrix depends upon the uniqueness of data types stored within the data frame. If the data frame consists of more than one data type, the conversion to matrix will assume the default data type of character for all variables to preserve values.

The following code creates a data frame that contains monthly mean concentrations of atmospheric CO₂ as measured at the Mauna Loa observatory in Hawaii. The `head()` and `tail()` functions display the first and last few (typically 6) observations in an object (rows in data frame), and is a useful “sanity check” to ensure that the object appears correct.

```

CO2.df = data.frame(year, month, CO2)
head(CO2.df)

```

```
##   year month    C02
## 1 1959     1 315.42
## 2 1959     2 316.31
## 3 1959     3 316.50
## 4 1959     4 317.56
## 5 1959     5 318.13
## 6 1959     6 318.00
```

```
tail(C02.df)
```

```
##   year month    C02
## 463 1997     7 364.52
## 464 1997     8 362.57
## 465 1997     9 360.24
## 466 1997    10 360.83
## 467 1997    11 362.49
## 468 1997    12 364.34
```

Data can be extracted in a variety of ways. For example, individual columns or rows can be extracted using index values, or by using the `dataframe$variable` or `dataframe[['variable']]` constructions. The code below illustrates the three approaches to extracting the column containing `year` from the data frame.

```
C02.y = C02.df[, 1]
C02.y[1:24]
C02.y = C02.df$year
C02.y[1:24]
C02.y = C02.df[['year']]
C02.y[1:24]
```

Subsets can be extracted using the `subset()` function in base R. Note that the double equal sign (`==`) indicates a test for equality.

```
ss = subset(C02.df, year == 1980)
ss
ss = subset(C02.df, month == 8)
ss
```

7.9 Lists

Lists are collections of objects. A list can be thought of as a vector in which the individual elements do not have to be the same type. A list can include vectors, matrices, data frames or even other lists, in any combination.

Data frames are actually a special case of a list in which each object is a vector of the same length but possibly different types. Output from many statistical procedures is structured as a list.

The code below illustrates the creation of a list, specification of names for the elements, and three approaches to extracting specific elements.

```
# define a list with three elements: a character string, a numeric vector and a vector of factors
my_list = list("this is a character string", seq(0, 1, 0.1), as.factor(letters[1:13]))
my_list

## [[1]]
## [1] "this is a character string"
##
## [[2]]
## [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

```
##
## [[3]]
## [1] a b c d e f g h i j k l m
## Levels: a b c d e f g h i j k l m
names(my_list) = c("char_string", "num_vec", "factor_letters")
my_list

## $char_string
## [1] "this is a character string"
##
## $num_vec
## [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
##
## $factor_letters
## [1] a b c d e f g h i j k l m
## Levels: a b c d e f g h i j k l m
# extract an element (a) by position and (b) by name (two versions)
my_list[[3]]

## [1] a b c d e f g h i j k l m
## Levels: a b c d e f g h i j k l m
my_list$factor_letters

## [1] a b c d e f g h i j k l m
## Levels: a b c d e f g h i j k l m
my_list["factor_letters"]

## $factor_letters
## [1] a b c d e f g h i j k l m
## Levels: a b c d e f g h i j k l m
```

7.10 User-defined objects

It is possible to define other object types that are specific to an application. For example, the `sp` package for spatial data defines objects such as `SpatialPointsDataFrame`. However, this is an advanced topic, which we will not delve into further at this point.

If you are interested in learning more about object-oriented programming in R, the following link is a useful starting point: <https://stackoverflow.com/questions/4143611/sources-on-s4-objects-methods-and-programming-in-r>.

7.11 Finding out information about an object

In many cases, you will encounter data objects for which you need to figure out how to access information within them. Handy functions include `class()`, `str()` (“structure”), `summary()` and, for some objects, `names()`.

```
str(co2)

## Time-Series [1:468] from 1959 to 1998: 315 316 316 318 318 ...
str(CO2.df)

## 'data.frame': 468 obs. of 3 variables:
```

```
## $ year : int 1959 1959 1959 1959 1959 1959 1959 1959 1959 1959 ...
## $ month: int 1 2 3 4 5 6 7 8 9 10 ...
## $ CO2 : num 315 316 316 318 318 ...

class(co2)

## [1] "ts"

names(CO2.df)

## [1] "year" "month" "CO2"

summary(co2)

##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    313.2  323.5   335.2   337.1   350.3   366.8

summary(CO2.df)

##      year      month      CO2
##  Min.   :1959   Min.    : 1.00   Min.    :313.2
## 1st Qu.:1968   1st Qu.: 3.75   1st Qu.:323.5
##  Median :1978   Median : 6.50   Median :335.2
##   Mean  :1978   Mean   : 6.50   Mean   :337.1
## 3rd Qu.:1988   3rd Qu.: 9.25   3rd Qu.:350.3
##   Max.   :1997   Max.    :12.00   Max.    :366.8

str(my_list)

## List of 3
## $ char_string : chr "this is a character string"
## $ num_vec     : num [1:11] 0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 ...
## $ factor_letters: Factor w/ 13 levels "a","b","c","d",...: 1 2 3 4 5 6 7 8 9 10 ...
```

8 Calculations and functions

The lists of functions below are not exhaustive, but should cover most of the functions and operators you are likely to need. A more complete list of functions in base R can be found at <https://stat.ethz.ch/R-manual/R-devel/library/base/html/00Index.html>.

8.1 Basic calculations and operators

Many of the operators are similar to those used in Excel, so that users familiar with Excel can apply most of their experience while coding formulae in R.

An important point to remember when coding formulae is that the order of operations generally follows the PEMDAS rule: begin with operations within parentheses, then progress through exponentiation, multiplication and division, then addition and subtraction.

I have seen many cases of code that runs but yields incorrect results because the order of operations was not rendered correctly. When in doubt, add extra parentheses, and it is good practice always to perform check calculations by hand.

Note that Excel interprets the unary operator “-” differently than R and most programming languages. Try typing `-3^2` into an Excel cell and at the R command prompt. For consistent results in Excel, you need to use the binary minus operator, i.e., `0 - 3^2`. See https://en.wikipedia.org/wiki/Order_of_operations for more details.

```

x = 19
y = 3
x + y          # addition
x - y          # subtraction
x/y            # division
x*y            # multiplication
x^y            # exponent: x to the power y
x %% y         # modulus (remainder from division)
x %/% y        # integer division

```

8.2 Trigonometric functions

The argument `x` is an angle in radians and `y` is a real number, except in `atan2()`, for which `x` and `y` are coordinates.

```

x = 30*pi/180      # 30 degrees converted to radians
cos(x)
sin(x)
tan(x)

y = 0.5
acos(y)
asin(y)
atan(y)

y = 2
x = 5
atan2(y, x)

```

8.3 Mathematical functions

Some commonly used functions are provided below.

```

sqrt(x)          # square root
log(x)           # natural (base e) logarithm
log10(x)         # base 10 logarithm
exp(x)           # "e to the power x"
abs(x)           # absolute value

```

The following functions round numbers in various ways.

```

floor(x)         # returns the largest integer values that are not less than x
ceiling(x)       # returns the smallest integer values that not greater than x
round(x)         # rounds the values to the specified number of decimal places (default 0)
trunc(x)         # returns integer values formed by truncating x toward 0
signif(x)        # rounds the values to the specified number of significant digits

```

As an exercise, set `x = pi` and `x = -pi` and apply the rounding functions listed above. Try to anticipate the results before you run the code.

8.4 Statistical and related functions

Below are a list of common functions for summarizing and manipulating vectors of data. Try setting `x = co2` and then running the functions.

```

sum(x)           # sum of values in x
length(x)        # number of elements (including missing values)
mean(x)          # sample mean
var(x)           # sample variance
sd(x)            # sample standard deviation
min(x)           # minimum value of x
max(x)           # maximum value of x
range(x)         # minimum and maximum values of x
median(x)        # median of elements in x
quantile(x)       # quantiles of x
fivenum(x)       # minimum, lower-hinge, median, upper-hinge, maximum for the input data
boxplot.stats(x) # statistics used to generate box plots
unique(x)        # extracts only the unique values of x
sort(x)          # sort, smallest to largest
rank(x)          # return ranks, with r = 1 for the smallest and r = n for the largest
rev(x)           # reverse the order of elements
weighted.mean(x, w) # weighted mean, w is a vector of weights

```

Note that `quantile()` can take additional arguments to specify which probabilities to use in generating quantile values. The following code extracts the minimum, lower quartile, median, upper quartile and maximum of a set of values.

```
quantile(co2, probs = c(0, 0.25, 0.5, 0.75, 1))
```

```
##      0%      25%      50%      75%     100%
## 313.180 323.530 335.170 350.255 366.840
```

8.5 Missing data

Missing data are a fact of life in most environmental data sets. In R, such values are coded as `NA`. Many functions, when applied to a set of values that include any `NA` values, will return the value `NA` unless there is an option within the function to specify alternative handling of `NA` values.

In the example below, the `na.rm = TRUE` argument causes `NA` values to be removed prior to applying the function.

```

a = c(1, 3, 8, 15, 24, NA)
a.mean = mean(a)
a.mean

```

```
## [1] NA
```

```

a.mean = mean(a, na.rm = TRUE)
a.mean

```

```
## [1] 10.2
```

To find `NA` values in a data set, you can use the `is.na()` function, which returns a vector of `TRUE/FALSE` values (`TRUE` if the value is `NA`). The `!is.na()` function returns `TRUE/FALSE` values, but `TRUE` if a value is not `NA`.

```
is.na(a)
```

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

```
!is.na(a)
```

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


8.6 Relational operators and applications

The operators below operate element-wise on a pair of vectors, and return a vector of logical (TRUE/FALSE) values.

```
x < y      # x less than y
x > y      # x greater than y
x <= y     # x less than or equal to y
x >= y     # x greater than or equal to y
x == y     # x equal to y
x != y     # x not equal to y
x %in% y   # is a specific value of x contained within the vector y
```

An important caution is that testing for equality of non-integer numeric variables using `==` may lead to incorrect results due to round-off error associated with the internal precision of the computer.

The following functions test for equality of objects, returning a single TRUE/FALSE value.

```
identical() # tests whether two objects (e.g, vectors) are identical
all.equal()  # tests whether two objects (e.g, vectors) are "nearly" equal (i.e., within a tolerance)
```

The examples below illustrate some example applications of relational operators.

```
x = c(1, 2, 3, 4, NA)
y = rev(x)
x[x != 3]                # extracts values of x for which x does not equal 3

## [1] 1 2 4 NA
which(x >= 4)            # returns index values for cases in which the argument is TRUE

## [1] 4
is.na(x)                 # returns vector of TRUE/FALSE according to the whether x is NA or not

## [1] FALSE FALSE FALSE FALSE TRUE
any(x > 0)                # TRUE if any value of x is less than 0

## [1] TRUE
all(x > 0)                # TRUE if all values of x are greater than 0

## [1] NA
all(x > 0, na.rm = TRUE) # TRUE if all non-NA values of x are greater than 0

## [1] TRUE
3 %in% x                  # TRUE if any values in x equal 3

## [1] TRUE
8 %in% x                  # TRUE if any values in x equal 8

## [1] FALSE
```

An interesting and occasionally useful point is that logical values are actually coded as numerical 0 and 1 values, and can be used in calculations.

```
x = c(1, 2, 3, 4, NA)
y = rev(x)
x < y
```

```
## [1] NA TRUE FALSE FALSE NA
```

```
as.numeric(x < y)
```

```
## [1] NA 1 0 0 NA
```

```
sum(x < y)
```

```
## [1] NA
```

```
x > y
```

```
## [1] NA FALSE FALSE TRUE NA
```

```
sum(x > y, na.rm = TRUE)
```

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

```
x > 2
```

```
## [1] FALSE FALSE TRUE TRUE NA
```

```
x*(x > 2)
```

```
## [1] 0 0 3 4 NA
```

One can extract elements of a vector based on values of another vector. Below are some examples that draw upon the CO₂ data. When using this approach, one must be careful to ensure that the vectors are “lined up” properly. Otherwise, results are likely to be spurious.

```
C02_jan = C02[month == 1]
```

```
C02_jan
```

```
## [1] 315.42 316.27 316.73 317.78 318.58 319.41 319.27 320.46 322.17 322.40
```

```
## [11] 323.83 324.89 326.01 326.60 328.37 329.18 330.23 331.58 332.75 334.80
```

```
## [21] 336.05 337.84 339.06 340.57 341.20 343.52 344.79 346.11 347.84 350.25
```

```
## [31] 352.60 353.50 354.59 355.88 356.63 358.34 359.98 362.09 363.23
```

```
C02_1990 = C02[year == 1990]
```

```
C02_1990
```

```
## [1] 353.50 354.55 355.23 356.04 357.00 356.07 354.67 352.76 350.82 351.04
```

```
## [11] 352.69 354.07
```

A safer approach is to keep the variables in a data frame and use the `subset()` function from base R. We will look at working with data frames in more detail in a later installment.

```
C02_jan.df = subset(C02.df, month == 1)
```

```
head(C02_jan.df)
```

```
##   year month    C02
```

```
## 1  1959     1 315.42
```

```
## 13 1960     1 316.27
```

```
## 25 1961     1 316.73
```

```
## 37 1962     1 317.78
```

```
## 49 1963     1 318.58
```

```
## 61 1964     1 319.41
```

```
C02_1990.df = subset(C02.df, year == 1990)
```

```
head(C02_1990.df)
```

```
##   year month    C02
```

```
## 373 1990     1 353.50
```

```
## 374 1990      2 354.55
## 375 1990      3 355.23
## 376 1990      4 356.04
## 377 1990      5 357.00
## 378 1990      6 356.07
```

8.7 Logical operators and applications

The most common logical operators are summarized below.

```
A && B # logical AND - returns TRUE if both A and B are TRUE; otherwise returns FALSE
A || B # logical OR - returns TRUE if at least one of A and B is TRUE; otherwise returns FALSE
A & B  # vectorized AND - operates element-wise
A | B  # vectorized OR - operates element-wise
!A     # logical negation: returns TRUE if A is FALSE and vice versa
```

Below are examples of the vectorized versions of the operators that follow on from earlier examples using the CO₂ data set.

```
CO2_jan_pre1990 = CO2[year < 1990 & month == 1]
CO2_jan_pre1990
```

```
## [1] 315.42 316.27 316.73 317.78 318.58 319.41 319.27 320.46 322.17 322.40
## [11] 323.83 324.89 326.01 326.60 328.37 329.18 330.23 331.58 332.75 334.80
## [21] 336.05 337.84 339.06 340.57 341.20 343.52 344.79 346.11 347.84 350.25
## [31] 352.60
```

```
CO2_jan_pre1990.df = subset(CO2.df, year < 1990 & month == 1)
head(CO2_jan_pre1990.df)
```

```
##   year month    CO2
## 1  1959     1 315.42
## 13 1960     1 316.27
## 25 1961     1 316.73
## 37 1962     1 317.78
## 49 1963     1 318.58
## 61 1964     1 319.41
```

```
tail(CO2_jan_pre1990.df)
```

```
##   year month    CO2
## 301 1984     1 343.52
## 313 1985     1 344.79
## 325 1986     1 346.11
## 337 1987     1 347.84
## 349 1988     1 350.25
## 361 1989     1 352.60
```

```
CO2_JJA.df = subset(CO2.df, month == 6 | month == 7 | month == 8)
head(CO2_JJA.df)
```

```
##   year month    CO2
## 6  1959     6 318.00
## 7  1959     7 316.39
## 8  1959     8 314.65
## 18 1960     6 319.43
## 19 1960     7 318.01
```

```
## 20 1960      8 315.74
```

```
tail(CO2_JJA.df)
```

```
##      year month    CO2
## 450 1996      6 365.01
## 451 1996      7 363.70
## 452 1996      8 361.54
## 462 1997      6 365.68
## 463 1997      7 364.52
## 464 1997      8 362.57
```

We will see applications of the non-vectorized operators (&& and ||) in a later installment on programming.

8.8 Functions for characters

Character variables are often used to indicate sampling locations or measurement sites. In addition, a facility to work with character strings is useful for plotting legends and other text on graphs and for “scraping” data from web sites. Below are a few commonly used functions from the rich set of functions for string processing that are available within base R. Note, however, that there are functions in contributed packages such as `stringi` and `stringr` that are generally preferred to those in base R.

<code>paste(x, y)</code>	# combine two strings "x" and "y" with, by default, a space between them
<code>paste0(x, y)</code>	# combine two strings "x" and "y" with, by default, no space between them
<code>substr(x, i, j)</code>	# extract i-th to j-th characters of each element of x (i and j are integers)
<code>nchar(x)</code>	# number of characters in each element of x
<code>grep("a", x)</code>	# which elements of x contain letter "a"?
<code>grep("a b", x)</code>	# which elements of x contain letter "a" or letter "b"?
<code>strsplit(x, "a")</code>	# split x into pieces wherever the letter "a" occurs
<code>as.character(y)</code>	# converts a numeric or factor variable "y" to a character string
<code>as.numeric(y)</code>	# converts a character or factor variable to numeric, if possible

Here are some examples.

```
paste(letters, LETTERS)
```

```
## [1] "a A" "b B" "c C" "d D" "e E" "f F" "g G" "h H" "i I" "j J" "k K"
## [12] "l L" "m M" "n N" "o O" "p P" "q Q" "r R" "s S" "t T" "u U" "v V"
## [23] "w W" "x X" "y Y" "z Z"
```

```
paste0(letters, LETTERS)
```

```
## [1] "aA" "bB" "cC" "dD" "eE" "fF" "gG" "hH" "iI" "jJ" "kK" "lL" "mM" "nN"
## [15] "oO" "pP" "qQ" "rR" "sS" "tT" "uU" "vV" "wW" "xX" "yY" "zZ"
```

```
nwm1 = "He's a real nowhere man,"
nwm2 = "sitting in his nowhere land"
nwm = paste(nwm1, nwm2)
nwm
```

```
## [1] "He's a real nowhere man, sitting in his nowhere land"
```

```
substr(nwm, 1, 6)
```

```
## [1] "He's a"
```

```
substr(c(nwm1, nwm2), 1, 6)
```

```
## [1] "He's a" "sittin"
```

```

nchar(nwm)

## [1] 52
grep("a", c(nwm1, nwm2))

## [1] 1 2
grep("d", c(nwm1, nwm2))

## [1] 2
grep("a|d", c(nwm1, nwm2))

## [1] 1 2
nwm_words_list = strsplit(c(nwm1, nwm2), " ") # split string at spaces and return a list
str(nwm_words_list)

## List of 2
## $ : chr [1:5] "He's" "a" "real" "nowhere" ...
## $ : chr [1:5] "sitting" "in" "his" "nowhere" ...
nwm_words_list[[1]] # extract character vector from list

## [1] "He's"      "a"          "real"       "nowhere" "man,"
nwm_words_list[[2]]

## [1] "sitting" "in"         "his"        "nowhere" "land"
y = 1:4
x = as.character(y)
x

## [1] "1" "2" "3" "4"
as.numeric(x)

## [1] 1 2 3 4

```

8.9 Functions for factors

Factors are categorical variables that are often coded as character strings (e.g., site names), but can also be coded as numbers (e.g., site number). In some cases, classification of a variable can be ambiguous. For example, one might identify sampling locations along a river by the distance upstream of some specified point (e.g., the mouth of the river). In this case, one might treat the site identifier as a numeric variable, especially in a spatial analysis, or as a categorical variable. Factors are commonly used in fitting linear models, such as analysis of variance and analysis of covariance.

Understanding factors, and especially ordered factors, is important when generating multiple graphs of subsets of data, separated on the basis of a factor variable, as we will see later.

A common source of grief occurs when R interprets a character variable as a factor or vice versa, especially when reading data in from a file. We will address this in more detail in a later installment.

Some common functions related to factors are listed below.

```

factor(x)           # codes the vector "x" as a factor
is.factor(x)        # returns TRUE if "x" is a factor variable
levels(x)           # returns or sets "levels" (unique values) of a factor "x"

```

```
as.factor(x)      # returns "x" as converted into a factor
ordered(x)       # creates an ordered factor
```

A few examples follow. Note the ordering of factor levels in the various examples.

```
y = 1:4
class(y)
```

```
## [1] "integer"
```

```
x = as.factor(y)
class(x)
```

```
## [1] "factor"
```

```
x = ordered(y)
x
```

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

```
class(x)
```

```
## [1] "ordered" "factor"
```

```
y = rep(1:3, times = 3)
y
```

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

```
class(y)
```

```
## [1] "integer"
```

```
x = factor(y)
x
```

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

```
class(x)
```

```
## [1] "factor"
```

```
levels(x)
```

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

```
x = factor(y, levels = 1:3)
class(x)
```

```
## [1] "factor"
```

```
ordered(x)
```

```
## [1] 1 2 3 1 2 3 1 2 3
## Levels: 1 < 2 < 3
```

```
x = factor(y, levels = 3:1)
class(x)
```

```
## [1] "factor"
```

```
ordered(x, levels = 3:1)
```

```
## [1] 1 2 3 1 2 3 1 2 3
## Levels: 3 < 2 < 1
```

8.10 ifelse()

The `ifelse()` function allows for alternative values to be returned depending on whether or not a condition is true. The syntax is as follows:

```
ifelse(test, yes, no)
```

- `test` = expression evaluating to TRUE or FALSE
- `yes` = value returned if test returns TRUE
- `no` = value returned if test returns FALSE

The following example computes the “water year” based on the calendar year and month, using the vectors generated for the Mauna Loa CO₂ data set. A water year begins Oct. 1 and extends to Sept. 30. For the months Oct. through Dec. the water year is one greater than the calendar year. For example, the 2015 water year ran from Oct. 1, 2014, to Sept. 30, 2015.

```
wy = ifelse(month < 10, year, year + 1)
```

`ifelse()` statements can be nested to accommodate multiple alternatives, as in the following example, which breaks a year into four three-month seasons (DJF, MAM, JJA, SON).

```
season = ifelse(month < 3 | month == 12, "winter",
               ifelse(month < 6, "spring",
                     ifelse(month < 9, "summer", "autumn")))
```

8.11 approx()

The `approx()` function can be useful for three purposes:

- infilling missing observations in a regularly sampled data series,
- estimating values at locations or times between observations, and
- estimating regularly spaced values from irregularly spaced observations

The arguments to the function are vectors representing the locations of the observations and the observed values (`x` and `y`), a vector representing the locations at which observations are desired (`xout`). You can also specify whether the values should be interpolated linearly between observations (`method = "linear"`), which is the default, or assumed to be constant between observations (`method = "constant"`). For the latter, an additional argument, `f`, controls the weights used to average between the two bounding observations. Type `?approx` at the command prompt for further details.

The result is a list with two elements: `x`, which corresponds to the values in the argument `xout`, and `y`, which is the estimated values.

For example, suppose that river temperature has been recorded at several irregularly spaced locations along a reach, and you want to estimate regularly spaced temperatures. The following code would work.

```
# x = distance along the river relative to some specified location
x = c(0, 125, 650, 980, 1200, 1450, 1810)
T = c(15.2, 14.8, 14.0, 13.8, 13.6, 13.2, 12.9)

# xout = locations at which estimates are desired, use default method = "linear"
x_int = seq(0, 1800, 100)
T_approx = approx(x = x, y = T, xout = x_int)
str(T_approx)
```

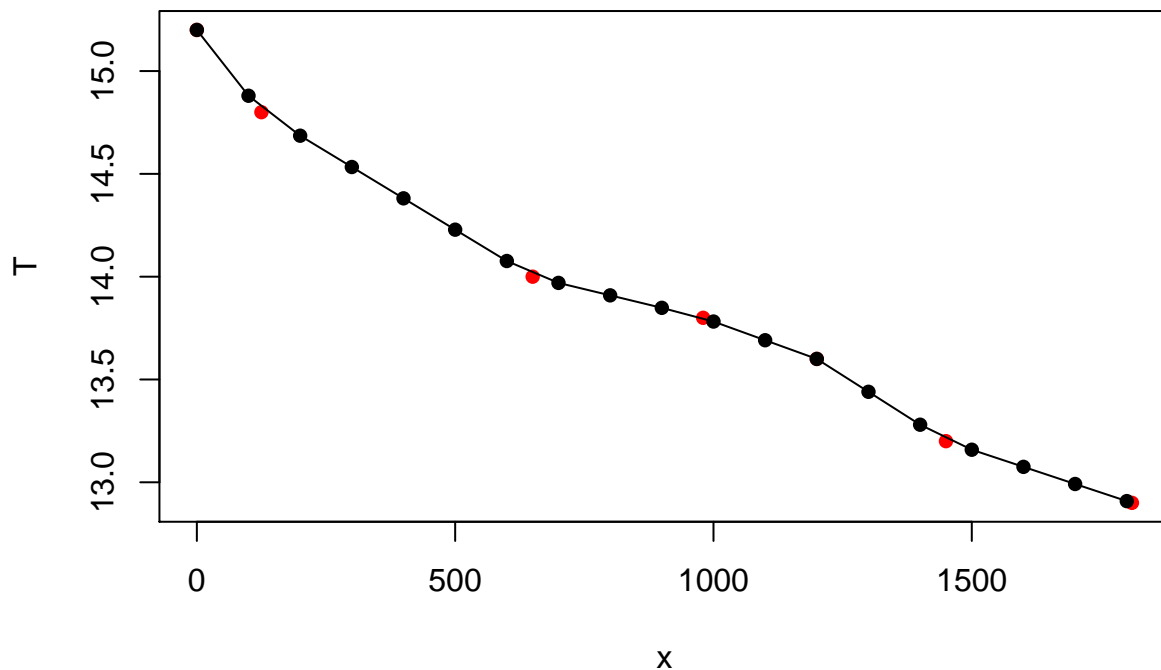
```
## List of 2
## $ x: num [1:19] 0 100 200 300 400 500 600 700 800 900 ...
## $ y: num [1:19] 15.2 14.9 14.7 14.5 14.4 ...

T_int = T_approx$y    # extract estimated temperatures

# compare estimated values to observed values at x = 0 and x = 1200
T_int[x_int == 0 | x_int == 1200]

## [1] 15.2 13.6
T[x == 0 | x == 1200]

## [1] 15.2 13.6
# plot observed and interpolated series; observed are red symbols,
# interpolated a black line with black filled circles
plot(x, T, pch = 16, col = "red")
lines(x_int, T_int, type = "o", pch = 16)
```



For a more complex approach, one could use a cubic spline using the `splinefun()` function in base R.

8.12 `expand.grid()`

A number of situations arise in which you want to generate all possible combinations of two or more vectors. For example, suppose you have some gridded data in the form of a matrix, in which the rows represent values of latitude and the columns represent values of longitude, and you want to create a matrix in which each observation is in its own row, along with the location coordinates.


```
# create a 3 x 3 matrix; by default, values are placed in matrix by column
z = matrix(runif(9, 0, 1), nrow = 3)
z
```

```
##           [,1]      [,2]      [,3]
## [1,] 0.42418865 0.1991107 0.06562967
## [2,] 0.22232591 0.4398213 0.90446441
## [3,] 0.03173034 0.9329622 0.26704958
```

```
lat = 54:52
long = -120:-118
```

```
latlong = expand.grid(lat, long)
latlong
```

```
##   Var1 Var2
## 1    54 -120
## 2    53 -120
## 3    52 -120
## 4    54 -119
## 5    53 -119
## 6    52 -119
## 7    54 -118
## 8    53 -118
## 9    52 -118
```

```
# `as.numeric()` converts `z` to a vector
zlatlong = cbind(latlong, as.numeric(z))
zlatlong.df = as.data.frame(zlatlong)
names(zlatlong.df) = c("lat", "long", "z")
zlatlong.df
```

```
##   lat long      z
## 1  54 -120 0.42418865
## 2  53 -120 0.22232591
## 3  52 -120 0.03173034
## 4  54 -119 0.19911072
## 5  53 -119 0.43982128
## 6  52 -119 0.93296218
## 7  54 -118 0.06562967
## 8  53 -118 0.90446441
## 9  52 -118 0.26704958
```

8.13 outer()

The `outer()` function applies a function using all combinations of two variables and stores the results in a matrix. The variables can be vectors or arrays. Two examples using pairs of vectors are provided below.

```
# create a 3 x 3 matrix of lat/long coordinates
lat = 54:52
long = -120:-118

latlong = outer(lat, long, FUN = paste)
latlong
```

```
##      [,1]      [,2]      [,3]
## [1,] "54 -120" "54 -119" "54 -118"
## [2,] "53 -120" "53 -119" "53 -118"
## [3,] "52 -120" "52 -119" "52 -118"

# generate a table of vapour pressure as a function of air temperature (T_a) and relative humidity (RH)
T_a = seq(0, 20, 2)
RH = seq(10, 100, 10)

# saturation vapour pressure (kPa) as a function of temperature
e = function(T, RH) {
  # T in deg C, RH in %
  ifelse(T >= 0, (RH/100)*0.6108*exp(17.27*T/(T + 237.3)),
         (RH/100)*0.6108*exp(21.87*T/(T + 265.5)))
}

e_table = outer(T_a, RH, FUN = e)
colnames(e_table) = as.character(RH)
rownames(e_table) = as.character(T_a)
e_table
```

	10	20	30	40	50	60	70
## 0	0.06108000	0.1221600	0.1832400	0.2443200	0.3054000	0.3664800	0.4275600
## 2	0.07056414	0.1411283	0.2116924	0.2822566	0.3528207	0.4233849	0.4939490
## 4	0.08132611	0.1626522	0.2439783	0.3253044	0.4066305	0.4879567	0.5692828
## 6	0.09351094	0.1870219	0.2805328	0.3740438	0.4675547	0.5610656	0.6545766
## 8	0.10727688	0.2145538	0.3218306	0.4291075	0.5363844	0.6436613	0.7509382
## 10	0.12279626	0.2455925	0.3683888	0.4911850	0.6139813	0.7367776	0.8595738
## 12	0.14025639	0.2805128	0.4207692	0.5610255	0.7012819	0.8415383	0.9817947
## 14	0.15986049	0.3197210	0.4795815	0.6394419	0.7993024	0.9591629	1.1190234
## 16	0.18182867	0.3636573	0.5454860	0.7273147	0.9091433	1.0909720	1.2728007
## 18	0.20639892	0.4127978	0.6191968	0.8255957	1.0319946	1.2383935	1.4447924
## 20	0.23382813	0.4676563	0.7014844	0.9353125	1.1691406	1.4029688	1.6367969
##	80	90	100				
## 0	0.4886400	0.5497200	0.6108000				
## 2	0.5645131	0.6350773	0.7056414				
## 4	0.6506089	0.7319350	0.8132611				
## 6	0.7480875	0.8415985	0.9351094				
## 8	0.8582151	0.9654919	1.0727688				
## 10	0.9823701	1.1051664	1.2279626				
## 12	1.1220511	1.2623075	1.4025639				
## 14	1.2788839	1.4387444	1.5986049				
## 16	1.4546293	1.6364580	1.8182867				
## 18	1.6511914	1.8575903	2.0639892				
## 20	1.8706250	2.1044531	2.3382813				

8.14 User-defined functions

Defining your own functions is a major advantage of using a coding language for data analysis, especially for developing code that you can re-use in other scripts. We will cover the writing of functions in more detail in a later installment, but introduce them here to whet your appetite.

It is often desired to quantify the variability of values within a data set. In conventional statistics, the standard deviation is commonly used, especially for data drawn from a normally distributed population.

In R, the built-in function `sd()` will compute the sample standard deviation. A drawback to the standard deviation is that, because it is based on squared deviations from the mean, it is sensitive to outliers.

As an alternative, a robust measure of spread in a data set can be based on absolute deviations around a measure of central tendency. In base R, the `mad()` function computes the median absolute deviation around the median. Suppose, however, that you wanted to compute the mean absolute deviation around the mean, defined as

$$MAD_{mean} = \frac{1}{n} \sum_{i=1}^n |x_i - \bar{x}|$$

This function definition could be coded as follows:

```
mad_mean = function(x) mean(abs(x - mean(x, na.rm = TRUE)), na.rm = TRUE)
```

In the function definition, `x` is called an argument to the function.

To apply the function, one would first execute the line of code defining the function, then invoke the function name using the name of the vector containing the data of interest as an argument. For example, the following code applies the function to the Mauna Loa CO₂ data.

```
mad_co2 = mad_mean(co2)
mad_co2
```

```
## [1] 13.06726
```

Note that the argument name in the function call (i.e., `co2` in `mad_co2 = mad_mean(co2)`) does not have to be the same as the argument name in the function definition.

We will look at creating functions in more detail in a later installment.

9 Overview of graphing packages/approaches

9.1 Base graphics

- built into core R
- based on pen-on-paper paradigm
- each element on a graph (generally) involves a separate function call
 - individual elements can be tweaked relatively easily
 - can require many lines of code and manual transformations for complex graphs

9.2 Lattice graphics

- based on the low-level graphical routines in the *grid* package
- each graph generated by a single function call
- customization handled by parameters within function call
- facilitates “conditioning” and “grouped” plots

9.3 ggplot2

- developed by Hadley Wickham
- based on Leland Wilkinson’s “Grammar of Graphics” concept
- can generate complex graphs using a compact set of function calls

- can often be difficult to customize details
- renders more slowly than Lattice (sometimes much more slowly)

9.4 Comparing the three approaches

An interesting set of discussions can be found via <http://stackoverflow.com/questions/2759556/r-what-are-the-pros-and-cons-of-using-lattice-versus-ggplot2> and links contained therein, particularly this one: <https://learnr.wordpress.com/2009/08/26/ggplot2-version-of-figures-in-lattice-multivariate-data-visualization-with-r-final-part>

10 Case study - Data processing and graphing using base R

10.1 Data set

10.1.1 The Alsea Watershed Study

A paired-catchment experiment was conducted in the Oregon Coast Range to examine the effects of forest harvesting on hydrology, water quality and fish populations. Three catchments were monitored both before and after harvest, including a control, which remained unlogged through the study, and two treatment catchments. The map below shows the study catchments.

Further information can be found via the following two references:

- Harris, D.D. 1977. Hydrologic changes after logging in two small Oregon coastal watersheds. US Geological Survey Water-Supply Paper 2037, Washington, DC.
- Stednick, J.D. (ed.) 2008. *Hydrological and Biological Responses to Forest Practices*. Springer, New York. <http://www.springerlink.com/content/u24j5036x81385l2/fulltext.pdf>

As an example data set, we will use the time series of annual sediment yield for the three catchments. “Sediment yield” refers to the mass of sediment exported from the catchment by the stream per unit catchment area. SI-related units are $\text{kg} \cdot \text{km}^{-2}$ or $\text{tonnes} \cdot \text{km}^{-2}$, where a “tonne” is 1000 kg.

10.1.2 Experimental setup

The assignment of experimental status and the timing of treatments is summarized in the two tables below.

Catchment	Experimental treatment
Needle Branch	Clearcut logging, hot slash burn, no buffer
Deer Creek	30% patch cut, slash burn, riparian buffer
Flynn Creek	Control

Period	Experimental phase
1959-1965	Pre-treatment
1966	Logging treatments applied
1967-1973	Post-treatment

By 1975, roughly a decade after logging, vegetation recovery was well underway at Needle Branch.

At Deer Creek, despite the gentler harvest treatments, a number of landslides originated from the road network and transported sediment to the stream channels.

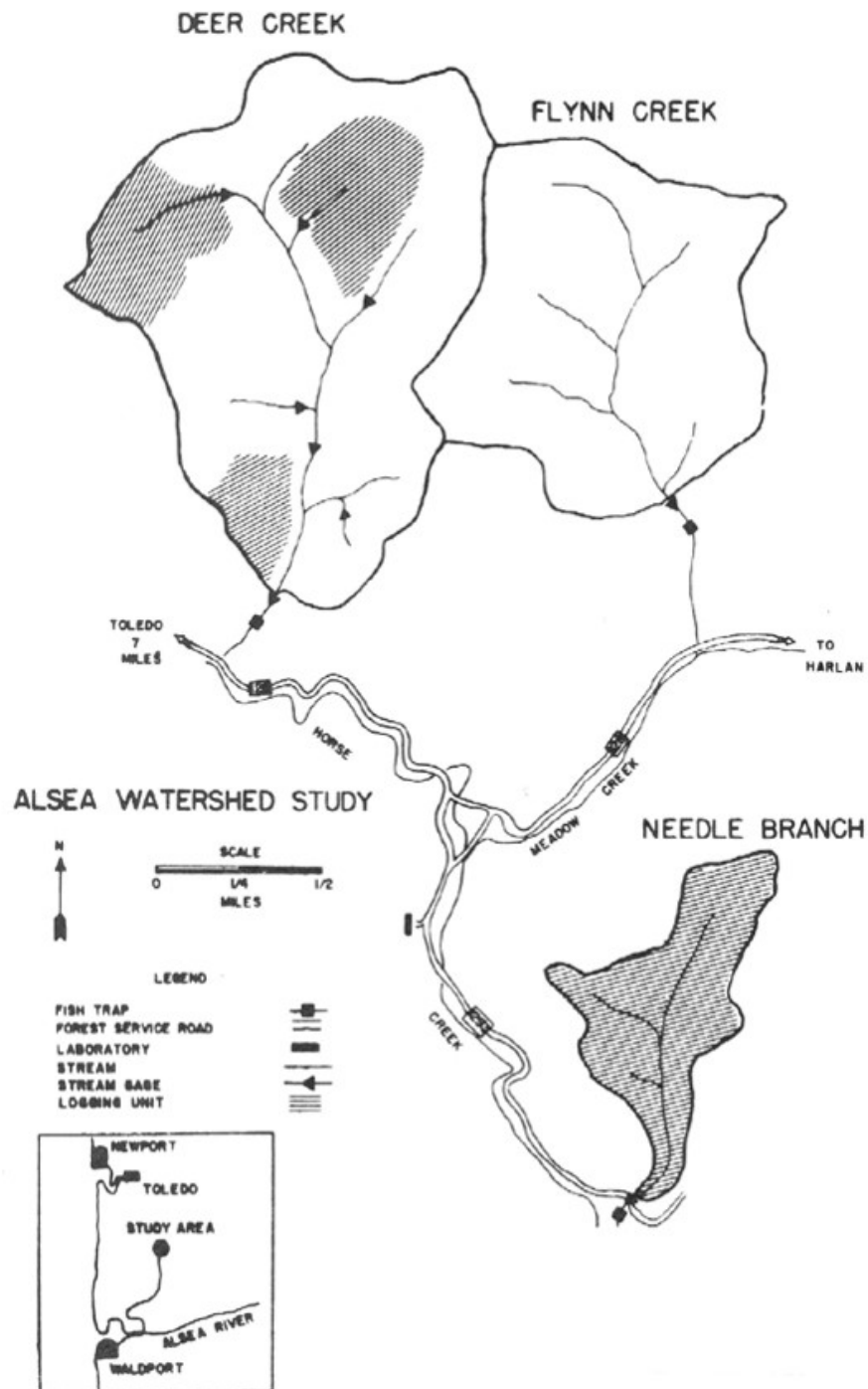


Figure 1: Map of Alsea Watershed Study catchments. Shaded areas indicate logging. Figure taken from Harris (1977).

10.1.3 Processing the data for analysis

The following code generates vectors for year and the sediment yield time series. The data were taken from tables in Harris (1977).

```
year = seq(1959, 1973, 1)
Flynn = c(88, 65, 338, 138, 114, 226, 1270, 291, 131, 67, 142, 121, 189, 1103, 88)
Needle = c(59, 41, 186, 141, 117, 184, 430, 368, 905, 490, 515, 232, 415, 519, 132)
Deer = c(91, 91, 340, 118, 162, 213, 1070, 746, 218, 87, 161, 147, 211, 1411, 131)
```

The values taken from the original study publications are in tons per square mile. We convert them to tonnes per square kilometer:

Next, we create a vector to hold a variable indicating the period within the study (i.e., pre-logging, logging, post-logging).

```
# note nested ifelse() function to create three options
period = ifelse(year < 1966, "pre-logging",
  ifelse(year == 1966, "logging", "post-logging")
)
```

Now we combine these vectors into a data frame:

```
##   year    Flynn   Needle    Deer    period
## 1 1959  15.43483  10.34836  15.96102 pre-logging
## 2 1960  11.40073   7.19123  15.96102 pre-logging
## 3 1961  59.28380  32.62363  59.63459 pre-logging
## 4 1962  24.20463  24.73082  20.69671 pre-logging
## 5 1963  19.99513  20.52131  28.41413 pre-logging
## 6 1964  39.63946  32.27284  37.35932 pre-logging
## 7 1965 222.75273  75.42022 187.67356 pre-logging
## 8 1966  51.04019  64.54567 130.84531    logging
## 9 1967  22.97686 158.73325  38.23630 post-logging
##10 1968  11.75152  85.94397  15.25944 post-logging
##11 1969  24.90621  90.32886  28.23873 post-logging
##12 1970  21.22290  40.69184  25.78319 post-logging
##13 1971  33.14982  72.78928  37.00852 post-logging
##14 1972 193.46162  91.03045 247.48355 post-logging
##15 1973  15.43483  23.15225  22.97686 post-logging

## 'data.frame':   15 obs. of  5 variables:
##  $ year  : num  1959 1960 1961 1962 1963 ...
##  $ Flynn : num  15.4 11.4 59.3 24.2 20 ...
##  $ Needle: num  10.35 7.19 32.62 24.73 20.52 ...
##  $ Deer  : num  16 16 59.6 20.7 28.4 ...
##  $ period: Factor w/ 3 levels "logging","post-logging",...: 3 3 3 3 3 3 1 2 2 ...

## [1] "year" "Flynn" "Needle" "Deer" "period"
```

10.2 Application of base graphics to visualize the Alsea Watershed Study sediment yield data

Base graphics use different functions to generate different types of plots. The main functions we will use in this course are:

- `plot()` for scatterplots and time series
- `hist()` for histograms

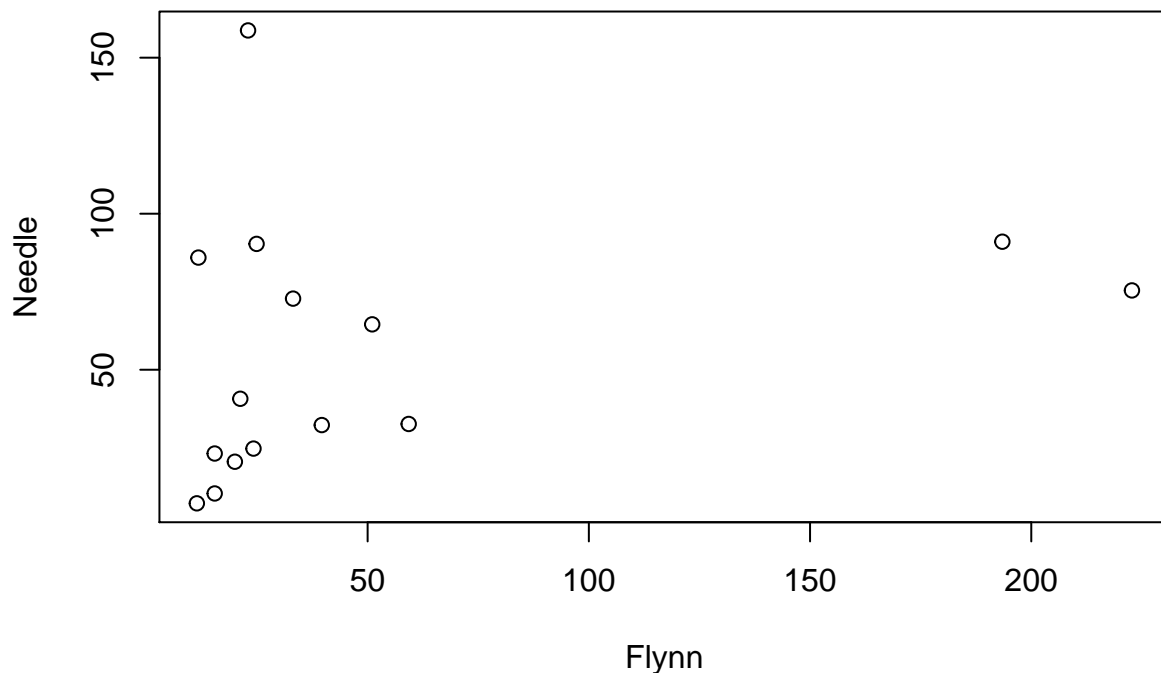
- `boxplot()` for box plots
- `qqnorm()` for normal probability plots

The following sections illustrate the use of these functions as applied to the Alsea Watershed Study sediment yields.

10.2.1 Scatterplots

Scatterplots are commonly used to explore the extent to which two variables are related. For example, one might assume that a year with high sediment yield in one stream might also be associated with high sediment yield in a nearby stream. For example, the following graph shows the relation between Sediment yield at Needle Branch (clearcut and burned) and at Flynn Creek (the unlogged control).

```
plot(Flynn, Needle)
```



It would be useful to use different colours and symbols to identify which period each point is associated with. In the following script, the symbol type is specified by the `pch` argument and the fill colour by the `bg` argument. We have also added a grid using the `grid()` function, and changed the x and y axis limits using `xlim` and `ylim`.

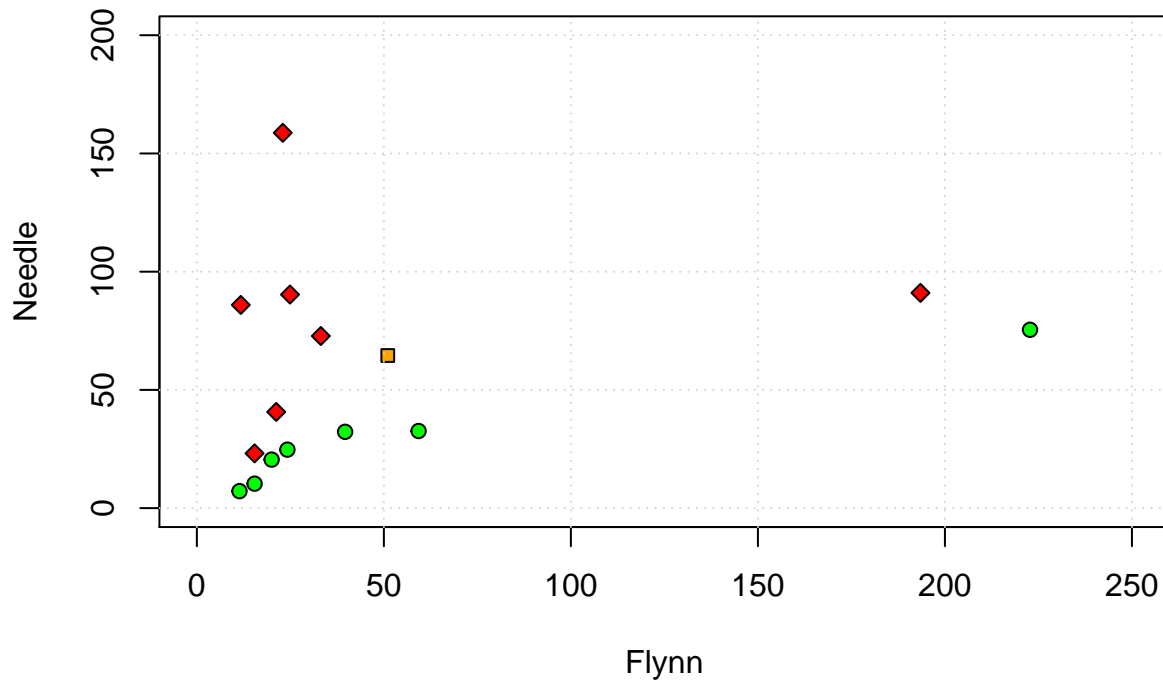
As can be seen, the data points for the logging and post-logging periods do appear to fall above the general trend of the pre-logging data.

```
periodnum = ifelse(year < 1966, 1, ifelse(year == 1966, 2, 3))
symcol = c("green", "orange", "red")
symtype = periodnum + 20
plot(Flynn, Needle,
     pch = symtype,
```

```

xlim = c(0, 250),
ylim = c(0, 200),
bg = symcol[periodnum]
)
grid()

```



To view the different symbol types available, type `?pch` at the command prompt.

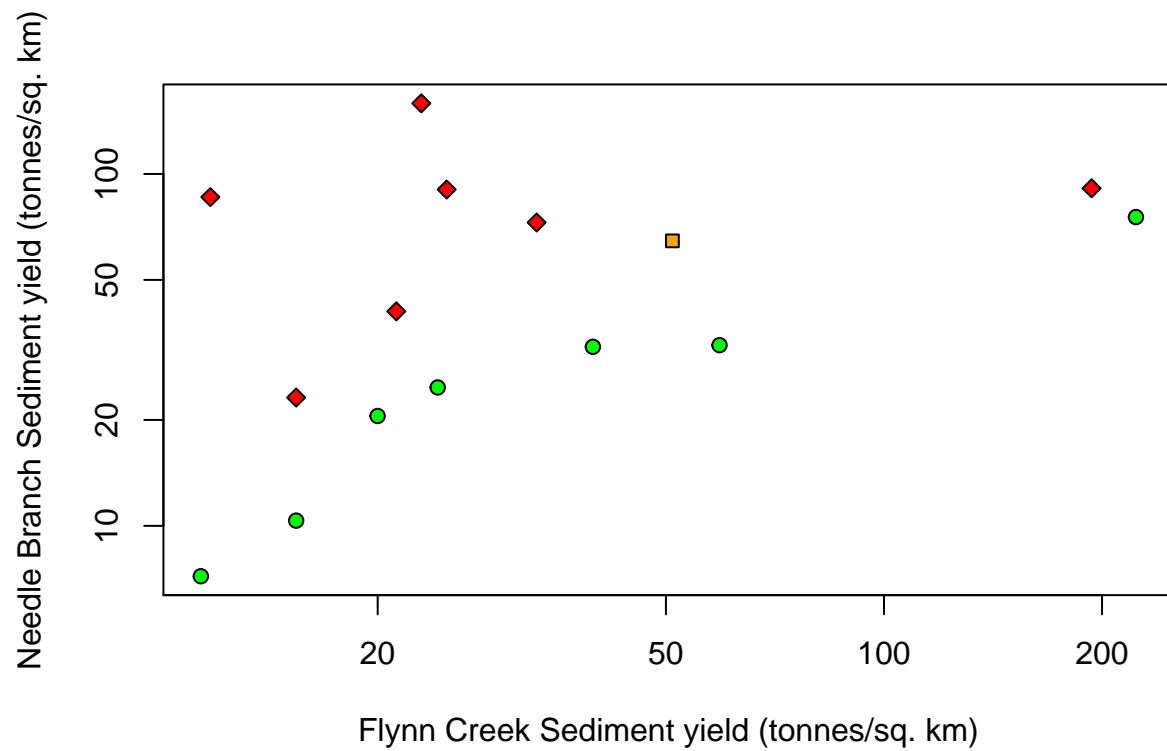
The use of different plotting symbols, colours and/or sizes, as in the above example, is a useful approach for representing multivariate patterns within a two-dimensional graph. In `ggplot2` terms, this is called “mapping variables to aesthetics.”

The following variation uses logarithmic axes for both x and y axes and also meaningful axis labels.

```

plot(Flynn, Needle,
     pch = symtype,
     bg = symcol[periodnum],
     log = "xy",
     ylab = "Needle Branch Sediment yield (tonnes/sq. km)",
     xlab = "Flynn Creek Sediment yield (tonnes/sq. km)"
)

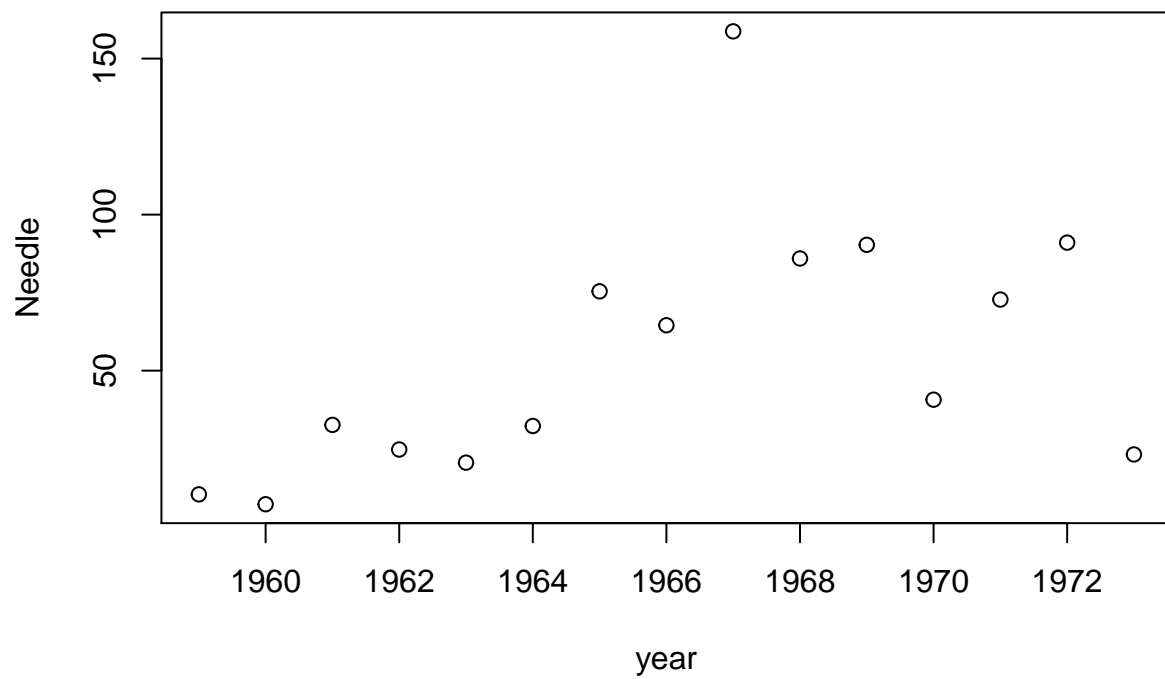
```

10.2.2 Time series graphs

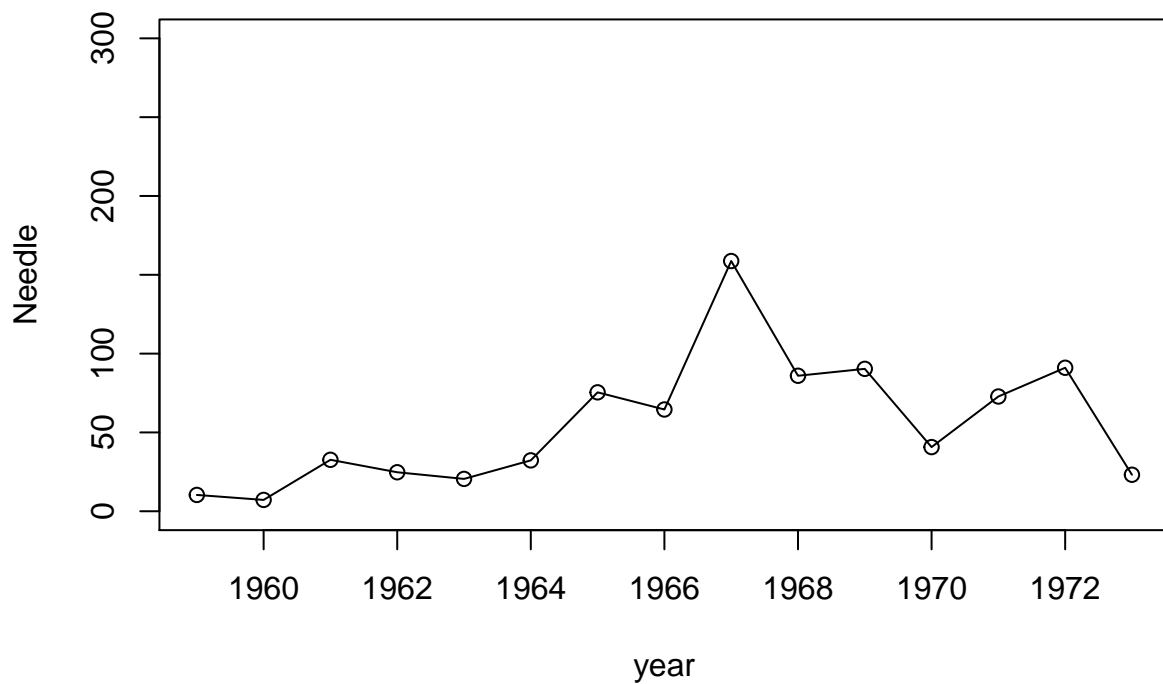
The `plot()` function can also generate time series plots by using a time variable on the x axis. The following command generates a plot of Needle Branch Sediment yield on the y axis against year on the x axis, using all defaults.

```
plot(year, Needle)
```



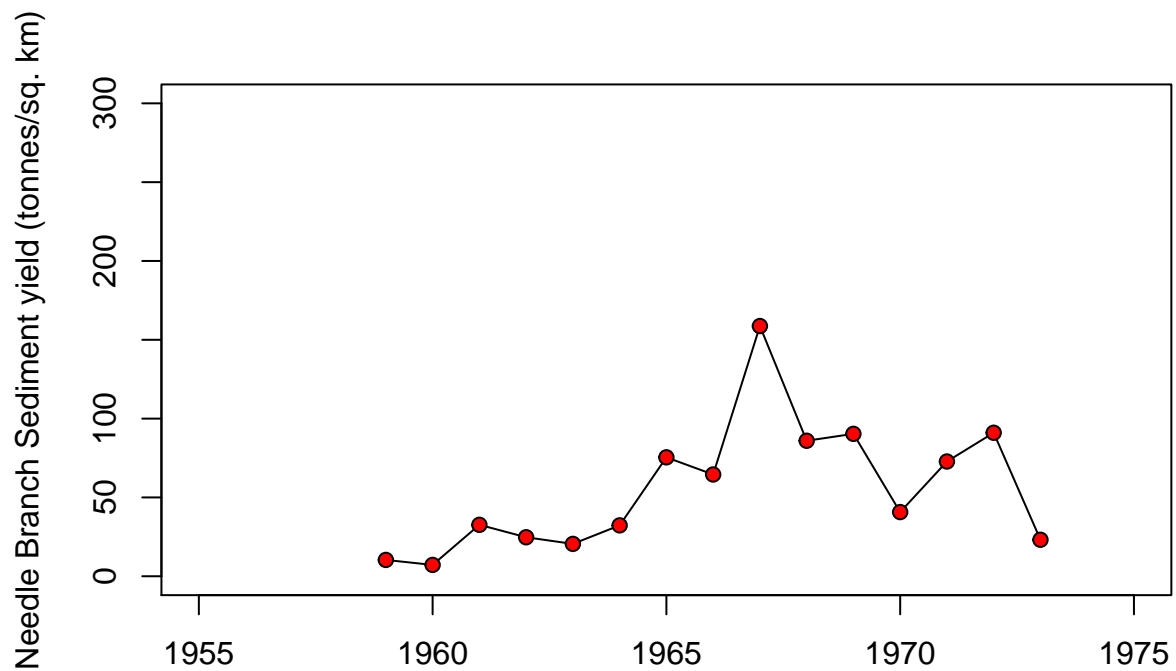
The following function call specifies both points and lines, overplotted (`type = "o"`), and sets the y axis limits from 0 to 300.

```
plot(year, Needle, type = "o", ylim = c(0, 300))
```



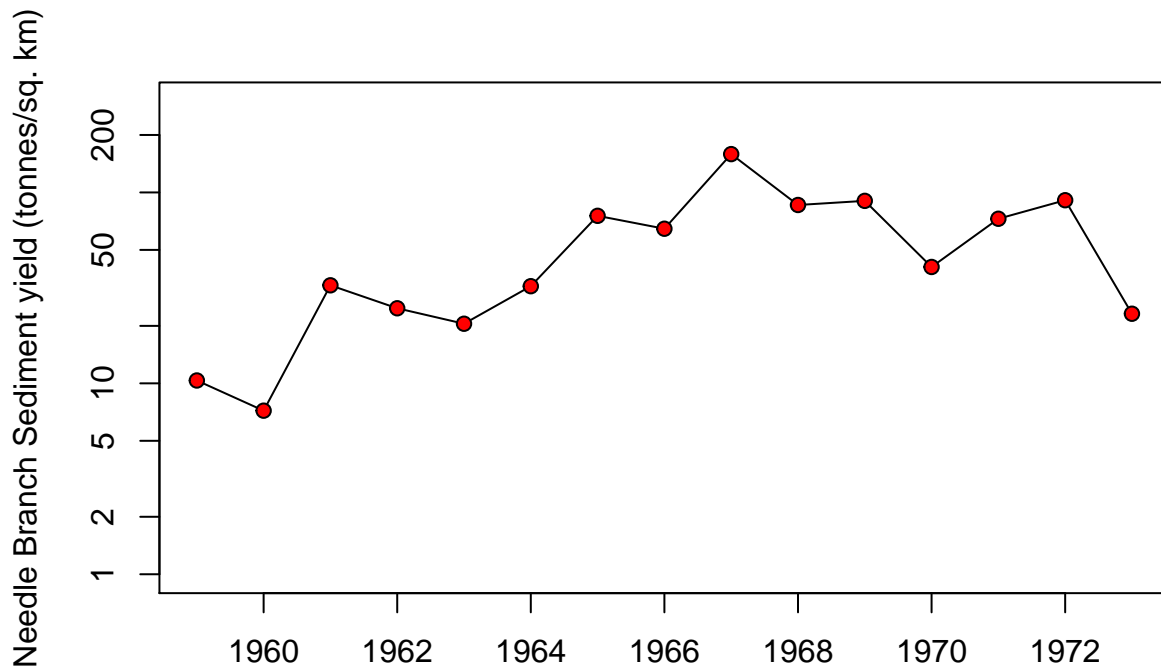
The next plot command uses a filled circle (`pch = 21`) with black outline (default) and red fill (`bg = "red"`). It specifies a meaningful y axis label, suppresses the x axis label, and sets the x axis limits to 1955 and 1975.

```
plot(year, Needle, type = "o",  
     pch = 21, bg = "red",  
     ylim = c(0, 300),  
     xlim = c(1955, 1975),  
     ylab = "Needle Branch Sediment yield (tonnes/sq. km)",  
     xlab = ""  
)
```



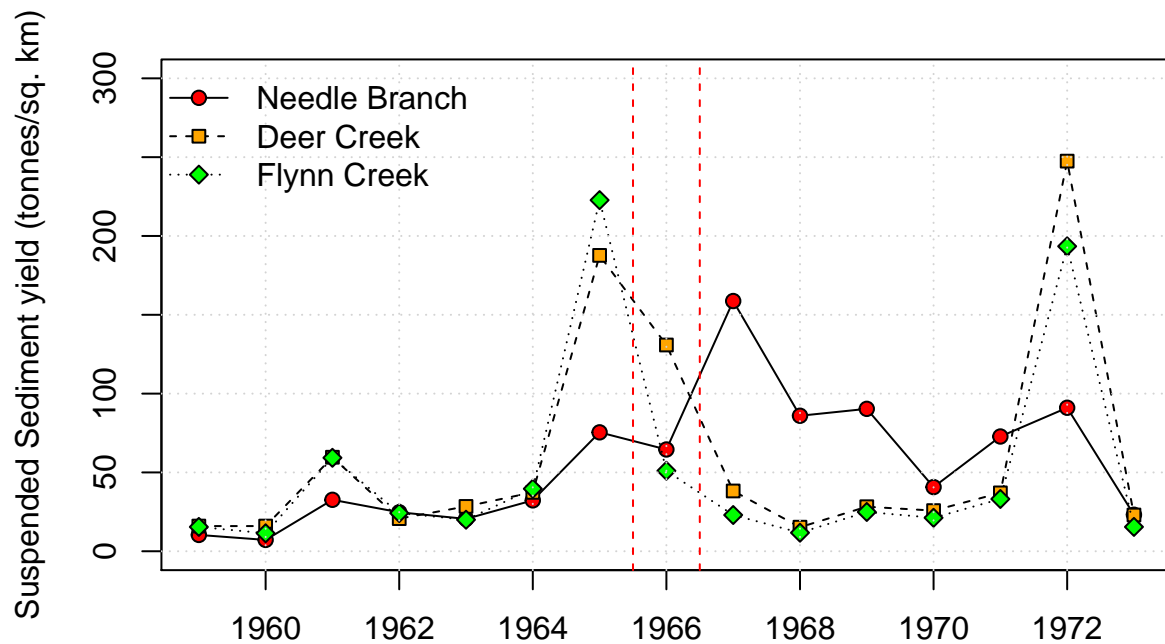
The following command specifies a logarithmic y axis. Note that the lower y axis limit cannot be 0 for a logarithmic scale.

```
plot(year, Needle, type = "o",  
      pch = 21, bg = "red",  
      log = "y",  
      ylim = c(1, 300),  
      ylab = "Needle Branch Sediment yield (tonnes/sq. km)",  
      xlab = ""  
)
```



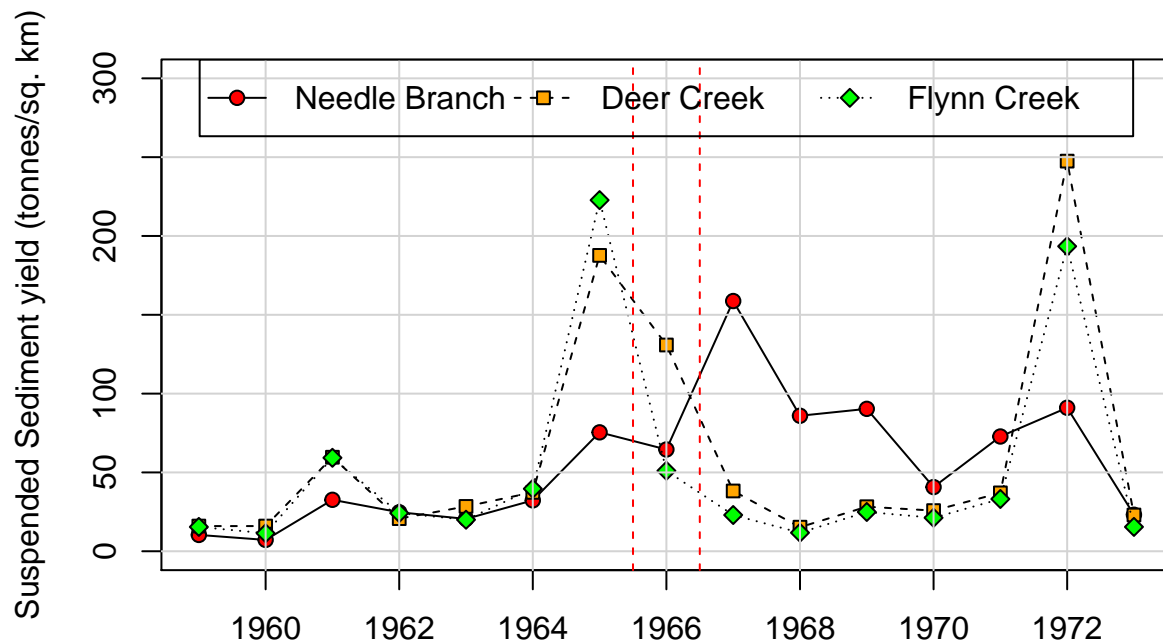
The following script plots all three series in one panel, using different symbols, colours and line types. Note the use of the `points()` function to add the time series for Deer and Flynn creeks. The script also adds a legend to the top left corner with no box around it. Vertical red dashed lines are added using the `abline()` argument to indicate boundaries between the pre-logging, logging and post-logging period. Finally, a grid has been added with the default line type of “dotted”.

```
plot(year, Needle, type = "o",
     pch = 21, bg = "red",
     ylim = c(0, 300),
     ylab = "Suspended Sediment yield (tonnes/sq. km)",
     xlab = ""
)
points( year, Deer, type = "o", pch = 22, lty = 2, bg = "orange")
points( year, Flynn, type = "o", pch = 23, lty = 3, bg = "green" )
grid()
legend( "topleft", bty = "n",
       lty = 1:3, pch = 21:23, pt.bg = c("red", "orange", "green"),
       legend = c("Needle Branch", "Deer Creek", "Flynn Creek")
)
abline( v = c(1965.5, 1966.5), lty = 2, col = "red" )
```



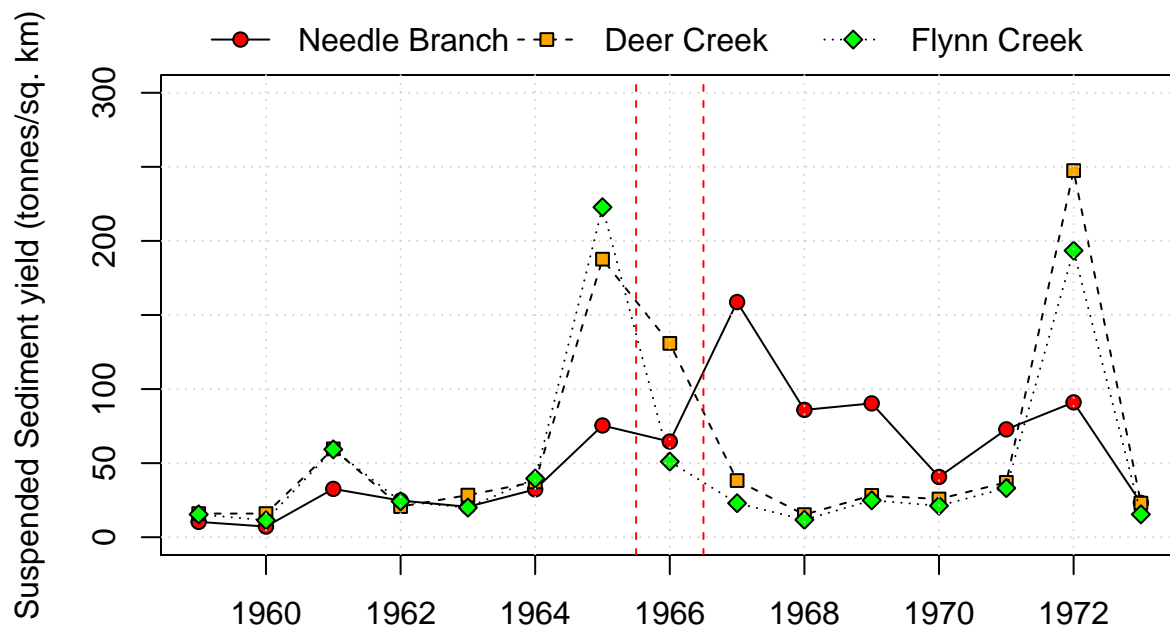
The following script generates the same graph, but with a different layout for the legend. Note the re-ordering of the `abline()` and `legend()` calls.

```
plot(year, Needle, type = "o",
     pch = 21, bg = "red",
     ylim = c(0, 300),
     ylab = "Suspended Sediment yield (tonnes/sq. km)",
     xlab = "")
)
points( year, Deer, type = "o", pch = 22, lty = 2, bg = "orange")
points( year, Flynn, type = "o", pch = 23, lty = 3, bg = "green" )
grid(lty = 1)
abline( v = c(1965.5, 1966.5), lty = 2, col = "red" )
legend( "top",
       lty = 1:3, pch = 21:23, pt.bg = c("red", "orange", "green"),
       legend = c("Needle Branch", "Deer Creek", "Flynn Creek"),
       ncol = 3
)
```



A final example shows how to plot the legend outside the graph frame. Note the inclusion of the `xpd` argument in the `legend()` function call.

```
par(mar = c(5, 5, 5, 1))
plot(year, Needle, type = "o",
     pch = 21, bg = "red",
     ylim = c(0, 300),
     ylab = "Suspended Sediment yield (tonnes/sq. km)",
     xlab = "")
)
points(year, Deer, type = "o", pch = 22, lty = 2, bg = "orange")
points(year, Flynn, type = "o", pch = 23, lty = 3, bg = "green")
grid()
abline(v = c(1965.5, 1966.5), lty = 2, col = "red")
legend(x = mean(year), xjust = 0.5, y = 310, yjust = 0,
     bty = "n", xpd = TRUE,
     lty = 1:3, pch = 21:23, pt.bg = c("red", "orange", "green"),
     legend = c("Needle Branch", "Deer Creek", "Flynn Creek"),
     ncol = 3
)
```

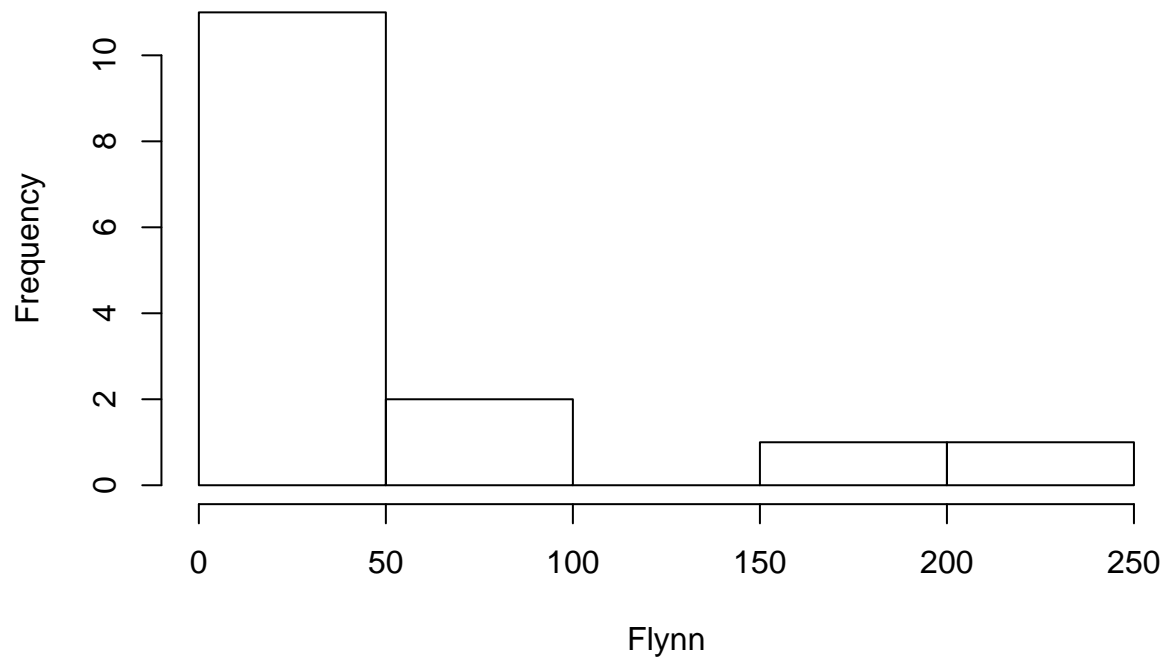


10.2.3 Histograms

Histograms display the frequency distribution of a variable, and can be generated using the `hist()` function.

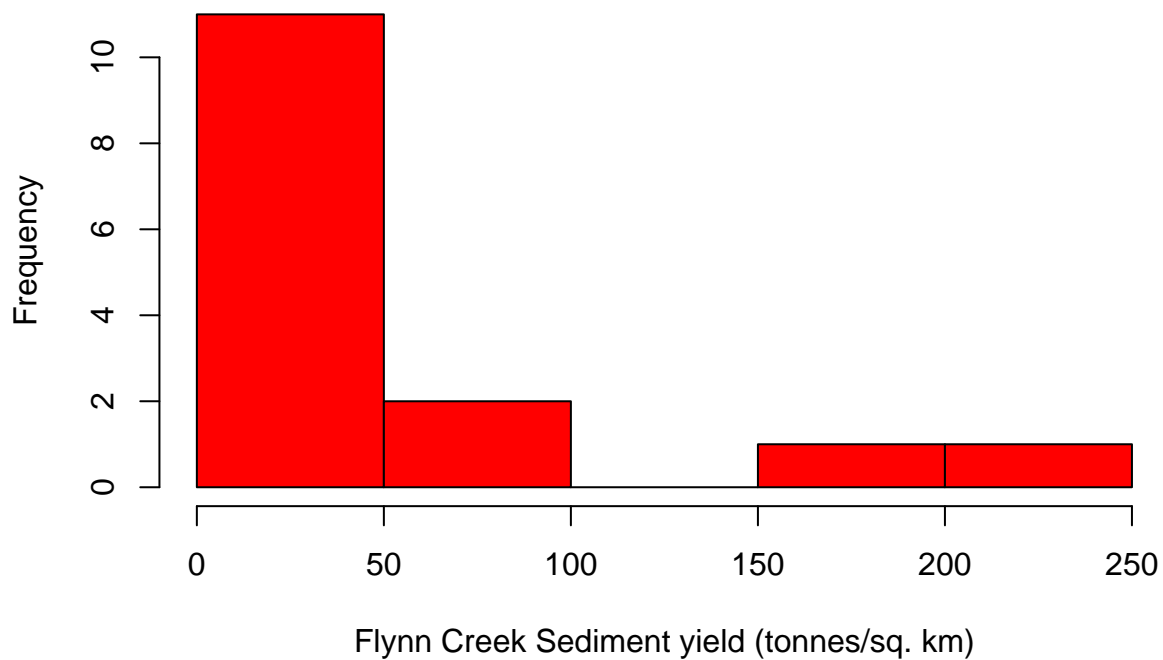
```
hist(Flynn)
```


Histogram of Flynn



Like `plot()`, `hist()` allows a number of arguments to be specified to modify the resulting graph. The following example colours the histogram bars red and adds a meaningful x axis label.

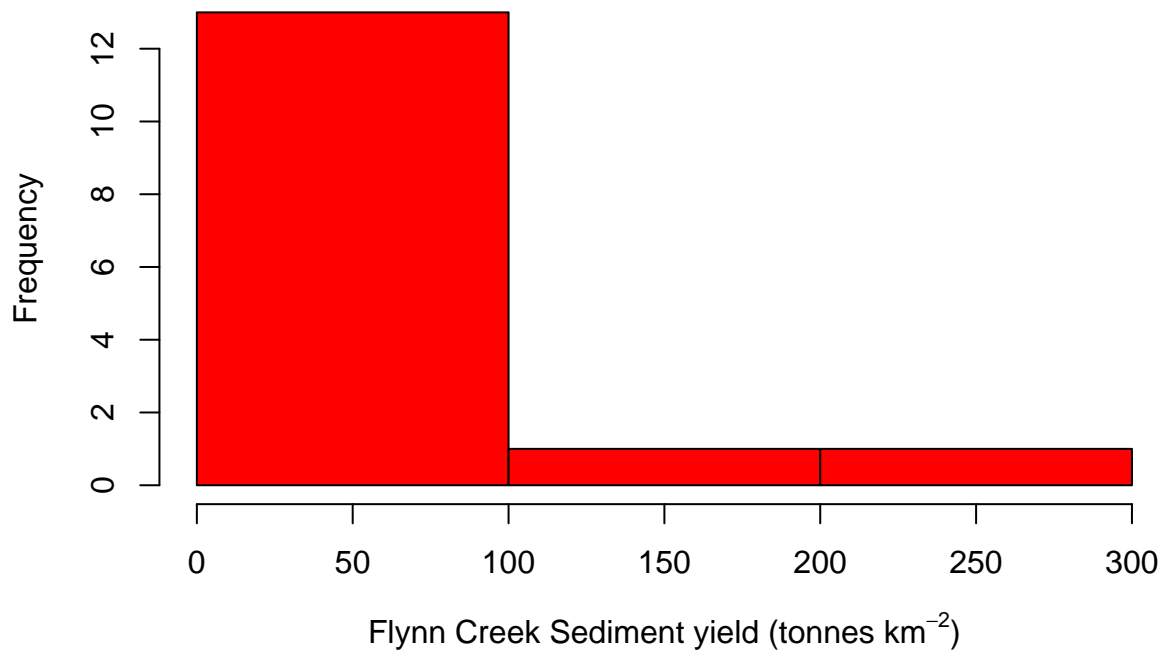
```
hist( Flynn, col = "red",  
      xlab = "Flynn Creek Sediment yield (tonnes/sq. km)",  
      main = ""  
    )
```



The following example illustrates the use of the `expression()` function as an argument to incorporate a superscript. As we will see in later examples, it can also be used to include subscripts, italic text, Greek letters and mathematical symbols. The script also illustrates how to modify the cutpoint boundaries for the histogram bars.

```
cp = c(0, 100, 200, 300)

hist( Flynn, col = "red",
      xlab = expression("Flynn Creek Sediment yield (*tonnes* " "km-2*)"),
      main = "",
      breaks = cp
    )
```



A handy tip

There are many situations in which you want to “bin” a set of data into distinct categories and count the number in each bin. One can accomplish this with the `hist()` function.

```
cp = seq(0, 300, 50)

nc = hist(Flynn, breaks = cp, plot = FALSE)
str(nc)

## List of 6
## $ breaks : num [1:7] 0 50 100 150 200 250 300
## $ counts : int [1:6] 11 2 0 1 1 0
## $ density : num [1:6] 0.01467 0.00267 0 0.00133 0.00133 ...
## $ mids : num [1:6] 25 75 125 175 225 275
## $ xname : chr "Flynn"
## $ equidist: logi TRUE
## - attr(*, "class")= chr "histogram"

nc$counts

## [1] 11 2 0 1 1 0
```

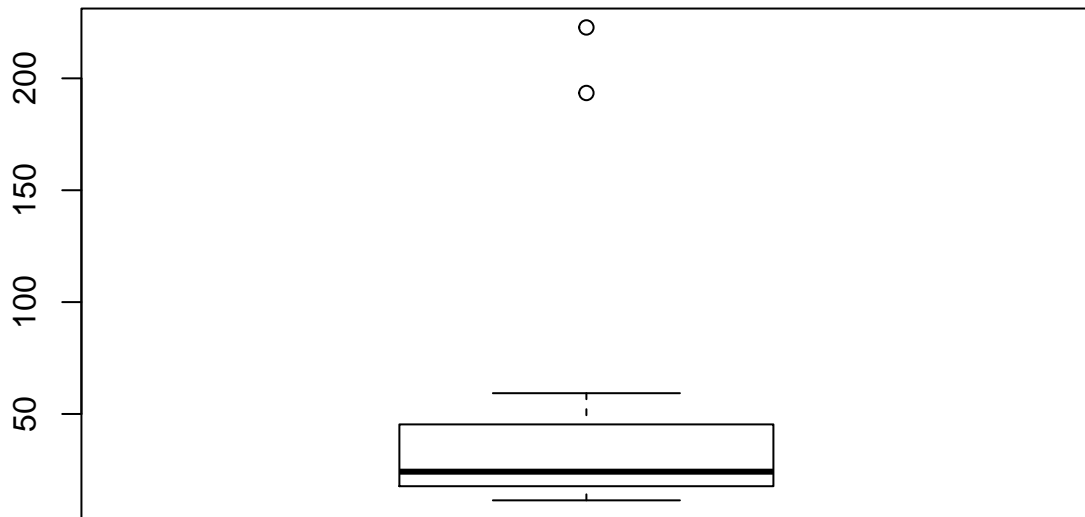
10.2.4 Box plots

Box plots are an alternative approach to displaying the distribution of a variable. The “box” portion indicated the lower quartile, median and upper quartile of the distribution. The whiskers extend from the lower/upper

quartiles to the lowest/highest data point that is not identified as an outlier. Outliers, if they are present, are plotted as symbols.

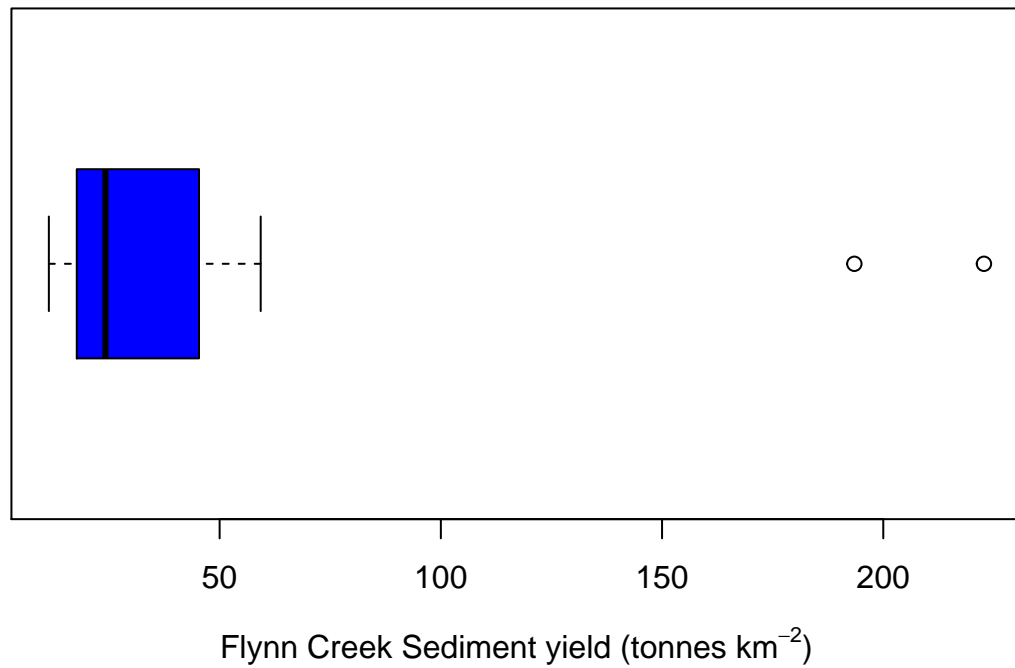
First we generate the most basic version for Flynn Creek Sediment yield.

```
boxplot(Flynn)
```



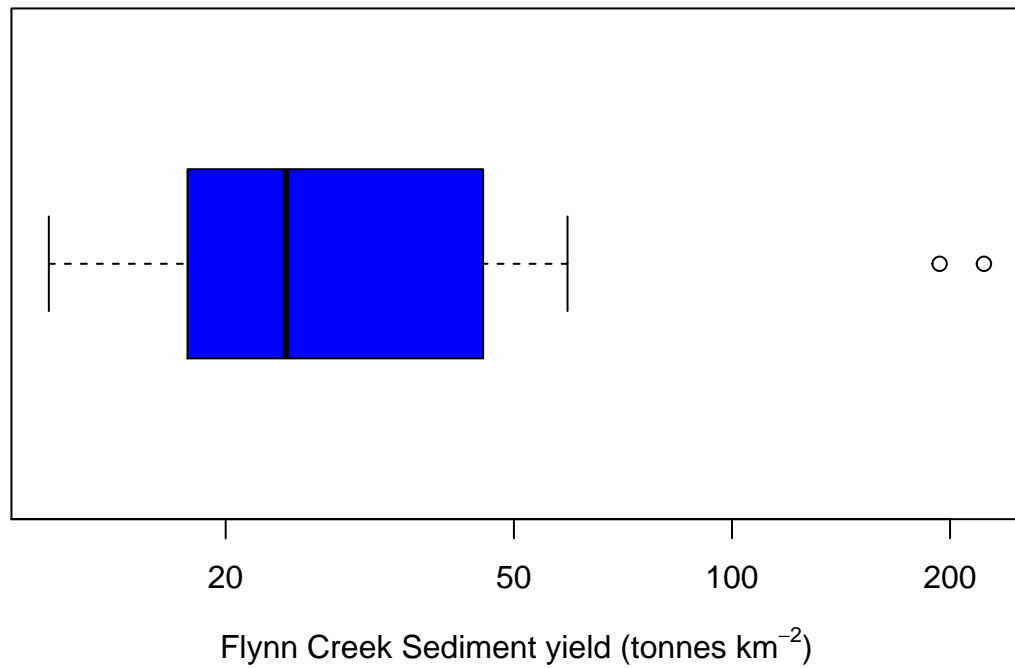
Now we colour the box, change the orientation, and add a meaningful x axis label.

```
boxplot( Flynn, col = "blue",  
  horizontal = TRUE,  
  xlab = expression("Flynn Creek Sediment yield ("*tonnes*" "km^{-2}*")"),  
  main = ""  
)
```



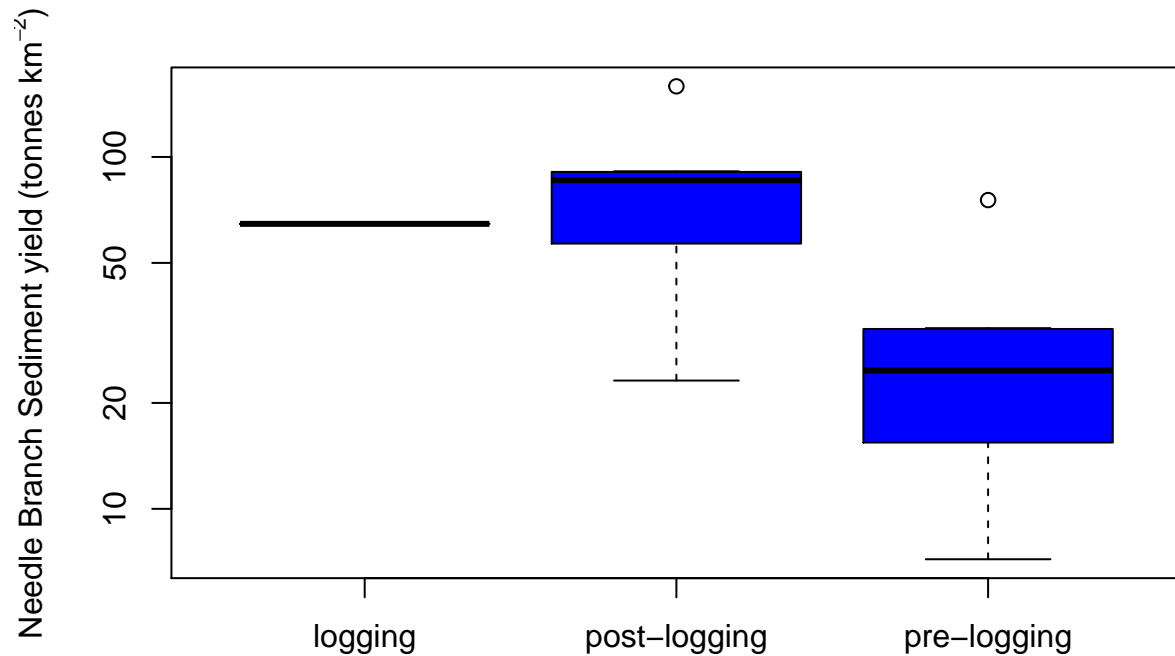
We can also use a logarithmic axis, as below.

```
boxplot( Flynn, col = "blue",
  horizontal = TRUE,
  xlab = expression("Flynn Creek Sediment yield (*tonnes* " "km^{-2}*")"),
  main = "",
  log = "x"
)
```



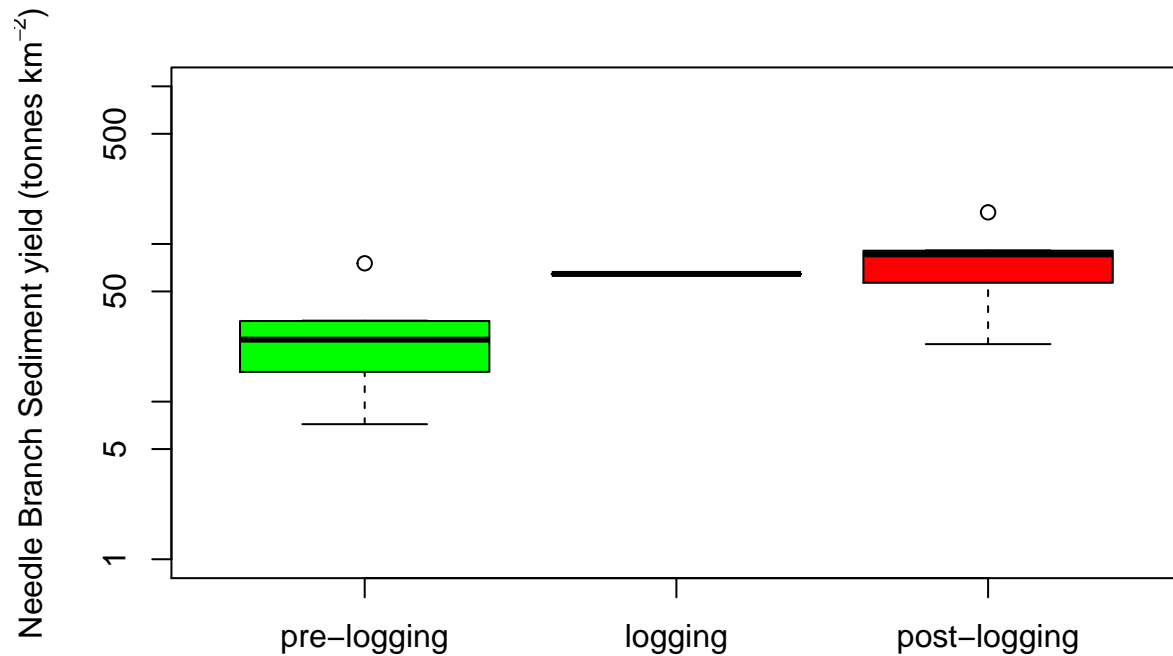
Finally, we can generate box plots stratified by a factor variable within the data set.

```
boxplot( Needle ~ period, col = "blue",
  ylab = expression("Needle Branch Sediment yield (*tonnes* " "km^{-2}*")"),
  main = "",
  log = "y"
)
```



Note that the period values are plotted in alphabetical order. We can change the order of the categories to chronological order by using a numerical value for period and specifying the period labels using the `names` argument. Note that `periodnum` was defined earlier; that line of code is repeated to remind you of how that variable was constructed.

```
periodnum = ifelse(year < 1966, 1, ifelse(year == 1966, 2, 3))
boxcol = c("green", "orange", "red")
boxplot( Needle ~ periodnum, col = boxcol,
  names = c("pre-logging", "logging", "post-logging"),
  ylab = expression("Needle Branch Sediment yield (*tonnes* " *km^{-2}*")"),
  main = "",
  log = "y",
  ylim = c(1, 1000)
)
```



Another approach is to make period a factor with a specified order to the levels. The second argument in the `factor()` function call below specifies the orders of the levels.

```
class(period)
```

```
## [1] "character"
```

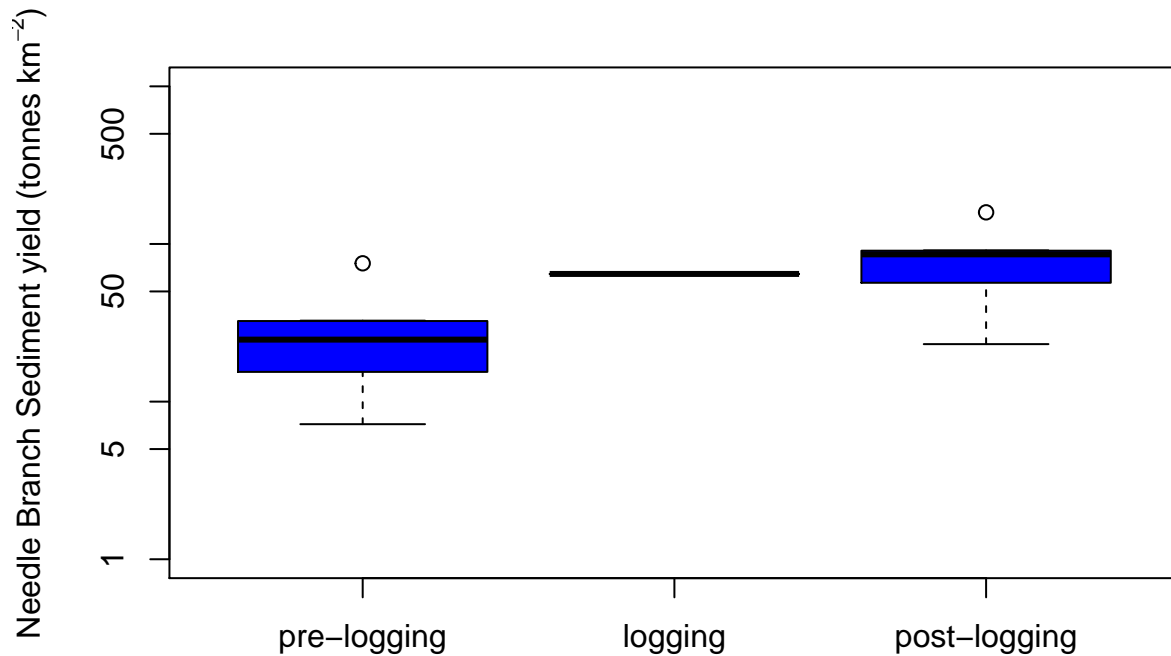
```
period = ordered(period, levels = c("pre-logging", "logging", "post-logging"))
class(period)
```

```
## [1] "ordered" "factor"
```

```
str(period)
```

```
## Ord.factor w/ 3 levels "pre-logging"<...: 1 1 1 1 1 1 1 2 3 3 ...
```

```
boxplot( Needle ~ period, col = "blue",
  ylab = expression("Needle Branch Sediment yield ("*tonnes*" "*km^{-2}*")"),
  main = "",
  log = "y",
  ylim = c(1, 1000)
)
```

As with the `hist()` function, it is possible to extract the summary values used to construct the plots, as in the following examples.

```
box_stat = boxplot(Flynn, plot = FALSE)
str(box_stat)
```

```
## List of 6
## $ stats: num [1:5, 1] 11.4 17.7 24.2 45.3 59.3
## $ n : num 15
## $ conf : num [1:2, 1] 12.9 35.5
## $ out : num [1:2] 223 193
## $ group: num [1:2] 1 1
## $ names: chr "1"
```

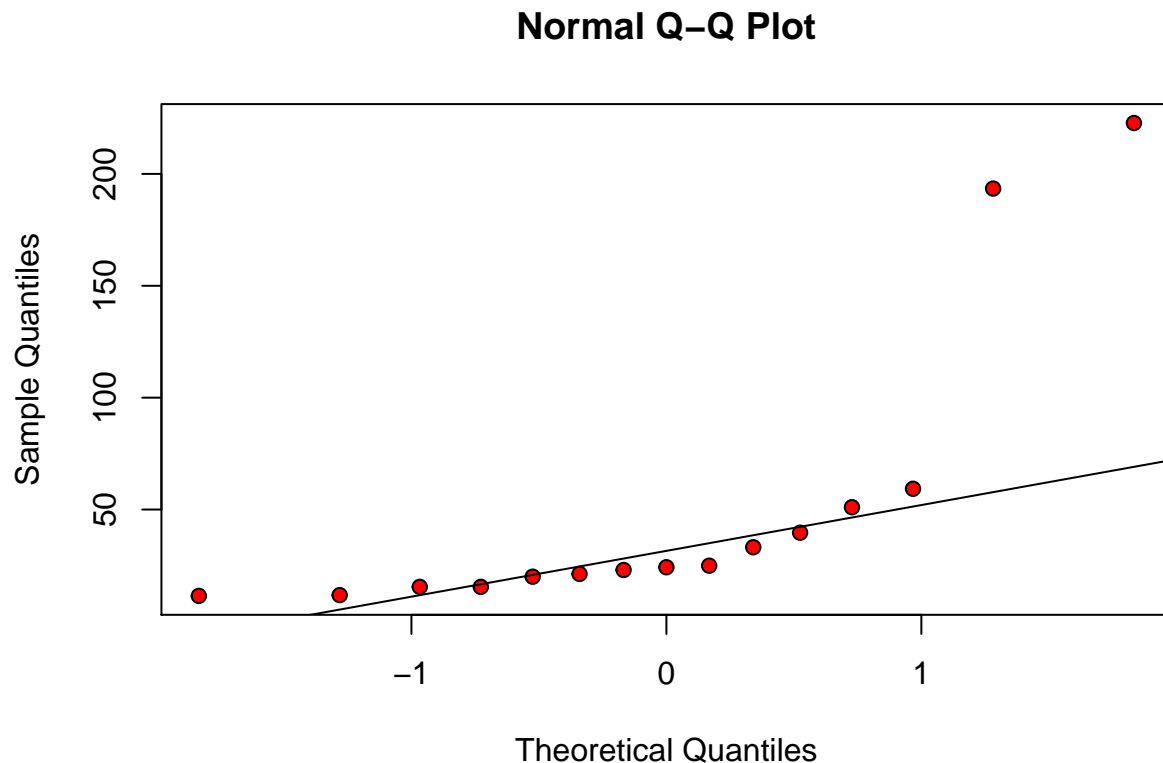
```
box_stat_group = boxplot(Flynn ~ period, plot = FALSE)
str(box_stat_group)
```

```
## List of 6
## $ stats: num [1:5, 1:3] 11.4 17.7 24.2 49.5 59.3 ...
## $ n : num [1:3] 7 1 7
## $ conf : num [1:2, 1:3] 5.25 43.16 51.04 51.04 16.59 ...
## $ out : num [1:2] 223 193
## $ group: num [1:2] 1 3
## $ names: chr [1:3] "pre-logging" "logging" "post-logging"
```

10.2.5 Normal probability plots

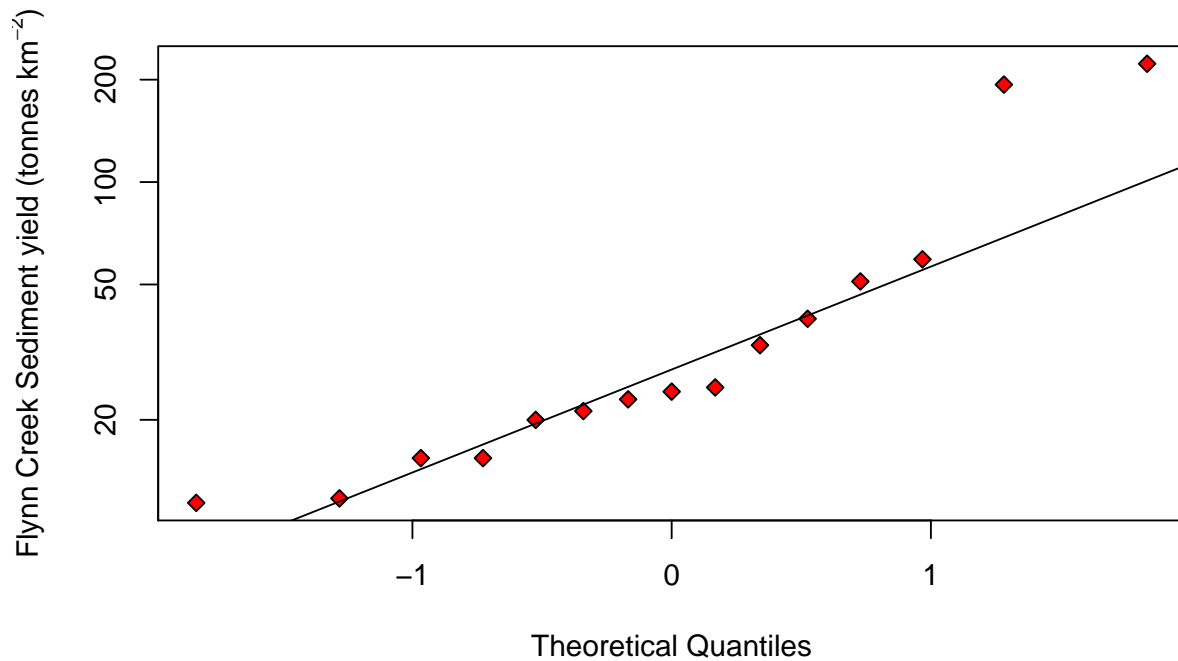
Normal probability plots are useful for examining whether data appear to have been drawn from a normally distributed population. The x axis is scaled so that data drawn from a normal distribution should plot roughly along a straight line. A concave-up pattern indicates a right-skewed distribution, and concave-down a left-skewed distribution. The first example is a basic normal probability plot for Flynn Creek Sediment yield using a linear y axis. The first function call `qqnorm` plots the points and the second, `qqline()`, fits a straight line to assist visual assessment.

```
qqnorm(Flynn, pch = 21, bg = "red")
qqline(Flynn)
```



The values are definitely right-skewed (consistent with the histogram). Let's try using a logarithmic axis to reduce the skew. As seen below, the log-transformation helps, but does not entirely remove the skew. Note that we have to transform `Flynn` in the `qqline()` function call.

```
qqnorm( Flynn, pch = 23, bg = "red",
  log = "y",
  ylab = expression("Flynn Creek Sediment yield ("*tonnes*" "*km^{-2}*")"),
  main = ""
)
qqline(log10(Flynn))
```



10.2.6 Density plots

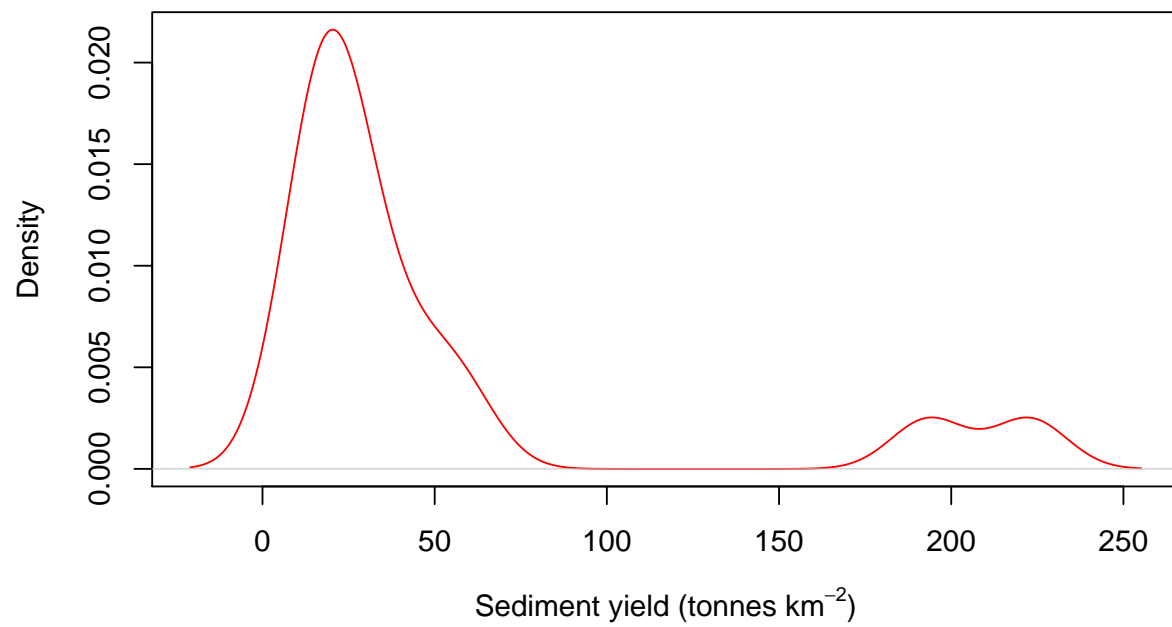
One problem with histograms is that they can be sensitive to the choice of bin interval. While box plots do not require a choice of bin interval, they cannot display multiple modes in a distribution. An approach that addresses both of these challenges is the use of a “kernel density estimator.”

The underlying theory is beyond the scope of this lecture, but you can find more information via https://en.wikipedia.org/wiki/Kernel_density_estimation. The main drawback is the need to specify the bandwidth. It may be useful to try several bandwidths to explore the sensitivity of the density curve. In base R, the `density()` function computes kernel density estimates. The output can be plotted using the `plot()` function.

```
dp = density(Flynn)
str(dp)

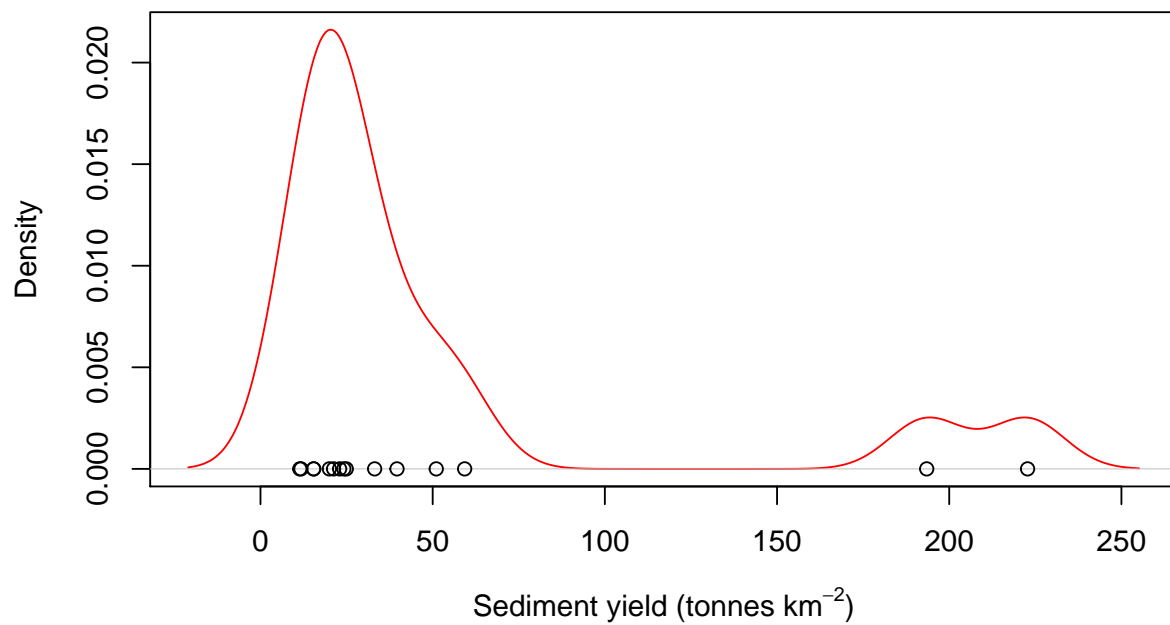
## List of 7
## $ x      : num [1:512] -21 -20.4 -19.9 -19.4 -18.8 ...
## $ y      : num [1:512] 7.42e-05 8.70e-05 1.01e-04 1.18e-04 1.37e-04 ...
## $ bw     : num 10.8
## $ n      : int 15
## $ call   : language density.default(x = Flynn)
## $ data.name: chr "Flynn"
## $ has.na  : logi FALSE
## - attr(*, "class")= chr "density"

plot(dp, col = "red",
      main = "",
      xlab = expression("Sediment yield ("*tonnes*" "*km^{-2}*")"))
)
```



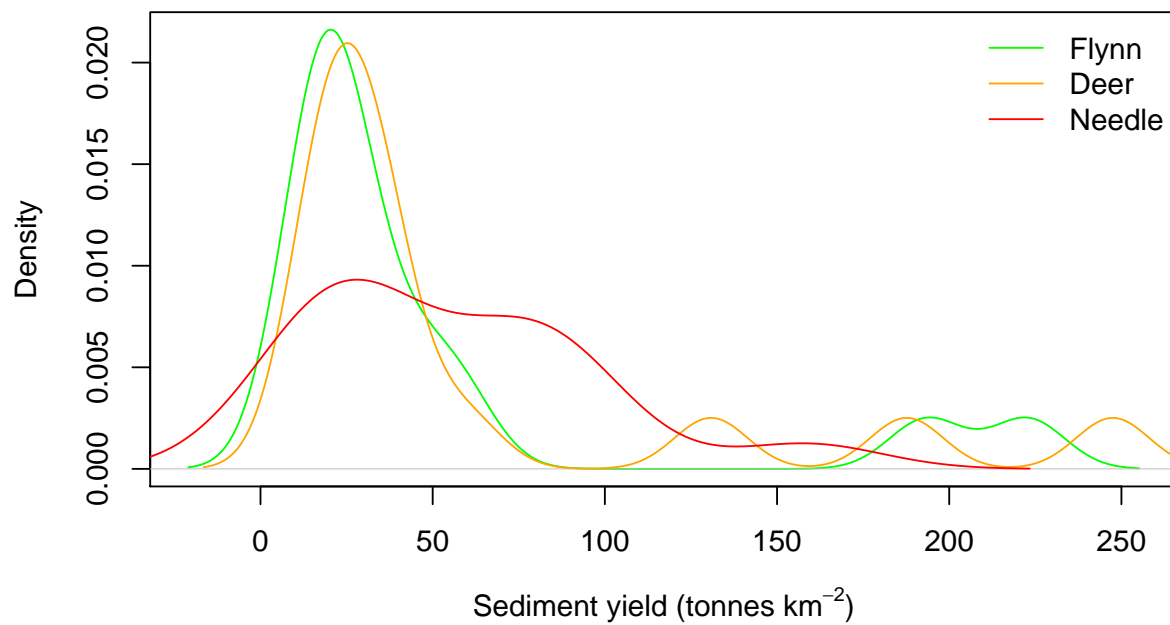
The graph below includes the data points.

```
plot(dp, col = "red",
     main = "",
     xlab = expression("Sediment yield ("*tonnes*" "*km^{-2}*"")")
)
points(Flynn, rep(0, length(Flynn)), pch = 4))
```



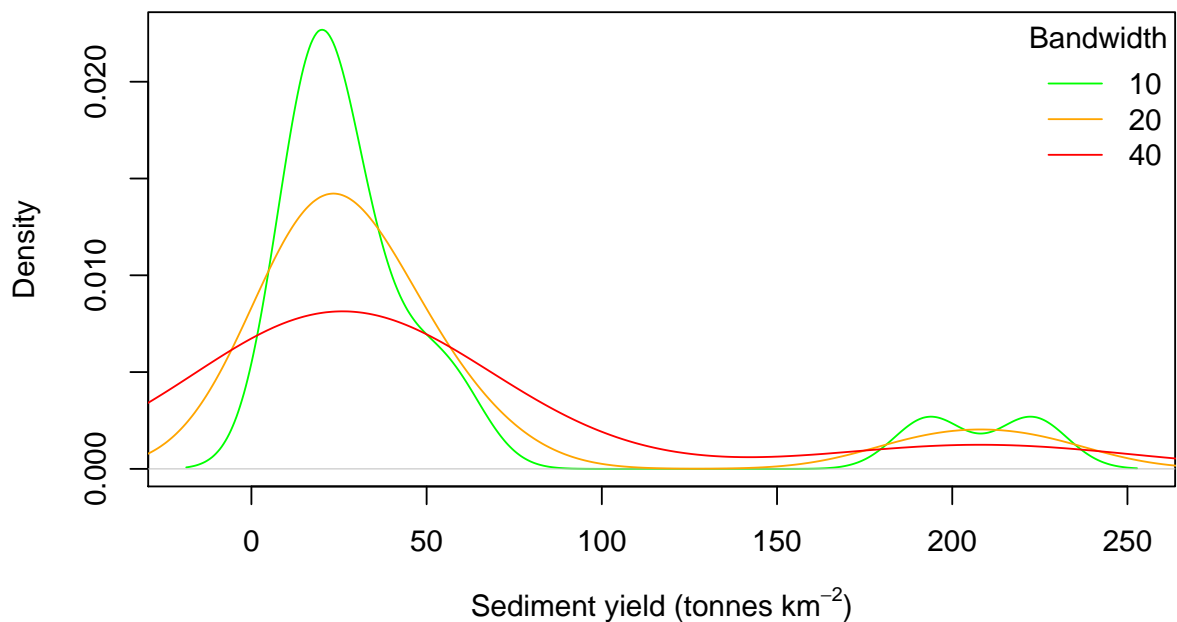
We can plot all three density curves on one graph for comparison.

```
dF = density(Flynn)
plot(dF, col = "green",
     main = "",
     xlab = expression("Sediment yield ("tonnes" "km^{-2}"))
)
dD = density(Deer)
lines(dD, col = "orange", main = "")
dN = density(Needle)
lines(dN, col = "red", main = "")
legend("topright", bty = "n", lty = 1,
     col = c("green", "orange", "red"),
     legend = c("Flynn", "Deer", "Needle")
)
```



We can also look at the effect of different bandwidths. The script below compares density curves for bandwidths of 10, 20 and 40 tonnes/km².

```
dF10 = density(Flynn, bw = 10)
dF20 = density(Flynn, bw = 20)
dF40 = density(Flynn, bw = 40)
plot(dF10, col = "green",
     main = "",
     xlab = expression("Sediment yield ("*tonnes*" "km^{-2}*")"))
)
lines(dF20, col = "orange")
lines(dF40, col = "red")
legend("topright", bty = "n", lty = 1,
     col = c("green", "orange", "red"),
     legend = c("10", "20", "40"),
     title = "Bandwidth"
)
```



10.2.7 Multi-panel plots

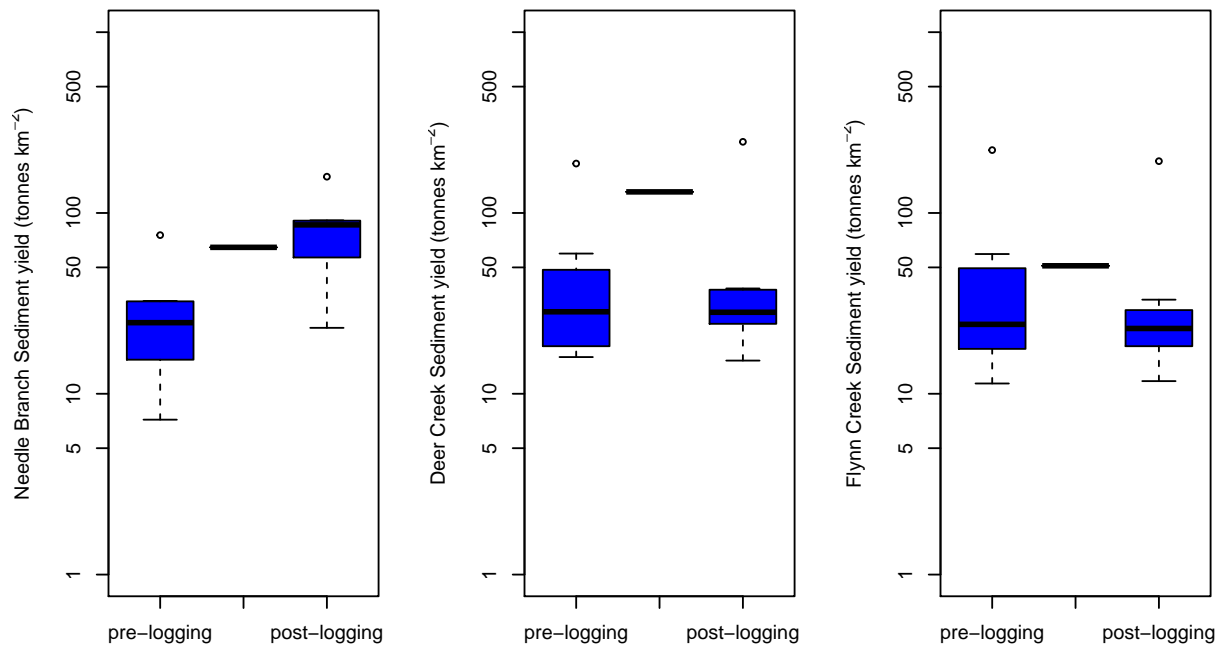
In many cases, an analyst wishes to include multiple graph panels in a plot. This can be accomplished by using the `mfrow` or `mfcol` arguments in a `par()` statement preceding the graphic function calls. For example, `mfrow = c(2, 2)` would create a 2 by 2 grid of graph panels inserted by row (i.e., the first and second graphs would be in the first row, and the third and fourth in the second row). Similarly `mfcol = c(2,2)` would generate a 2 by 2 grid of graph panels, but with the first and second in the first column and the third and fourth in the second column.

The following example illustrates the plotting of box plots by period for all three streams.

```
par(mfrow = c(1, 3))
boxplot(Needle ~ period,
  ylab = expression("Needle Branch Sediment yield ("*tonnes*" "*km^{-2}*")"),
  col = "blue", ylim = c(1, 1000), log = "y"
)

boxplot(Deer ~ period,
  ylab = expression("Deer Creek Sediment yield ("*tonnes*" "*km^{-2}*")"),
  col = "blue", ylim = c(1, 1000), log = "y"
)

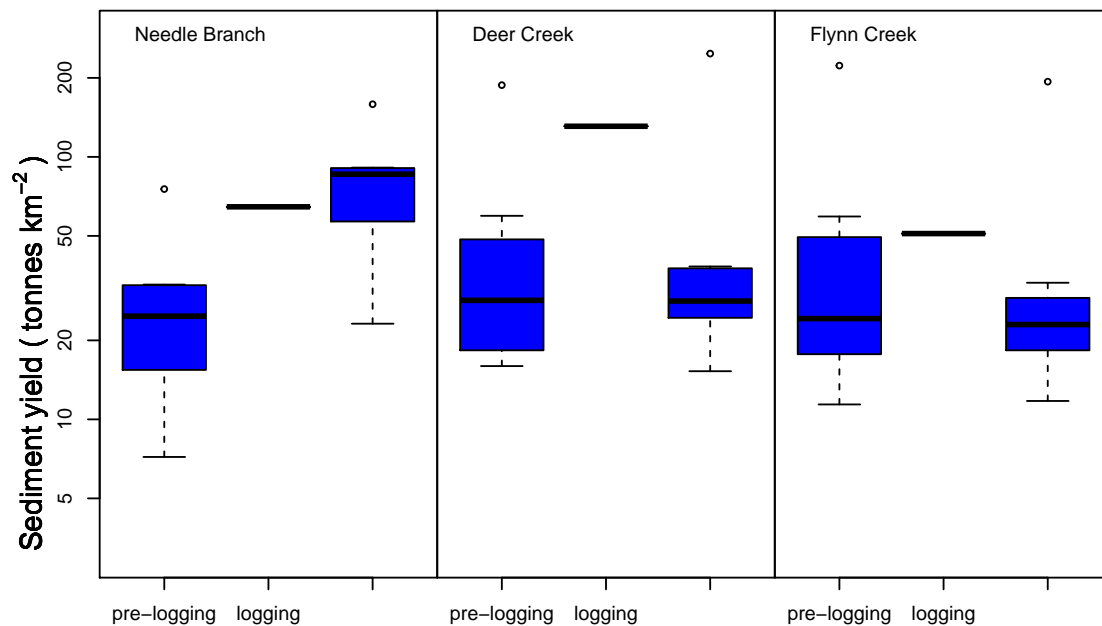
boxplot(Flynn ~ period,
  ylab = expression("Flynn Creek Sediment yield ("*tonnes*" "*km^{-2}*")"),
  col = "blue", ylim = c(1, 1000), log = "y"
)
```



In the above plot, the y axis labels take up a lot of space unnecessarily. We can reduce that wastage by suppressing the y axis labels and reducing the margins around each individual panel, and then adding a single y axis in an outer margin. Note how the y axis limits are stored in a variable preceding a function call. This is useful because the y axis limits of all three graphs can be changed by editing one line of code rather than changing the argument within each function call.

```
ylimits = c(3, 300)

par(mfrow = c(1, 3), mar = c(5, 0, 0, 0), oma = c(0, 5, 5, 5))
boxplot(Needle ~ period,
  ylab = "", yaxt = "n",
  col = "blue", ylim = ylimits, log = "y"
)
legend("topleft", bty = "n", legend = "Needle Branch")
boxplot(Deer ~ period,
  ylab = "", yaxt = "n",
  col = "blue", ylim = ylimits, log = "y"
)
legend("topleft", bty = "n", legend = "Deer Creek")
boxplot(Flynn ~ period,
  ylab = "", yaxt = "n",
  col = "blue", ylim = ylimits, log = "y"
)
legend("topleft", bty = "n", legend = "Flynn Creek")
axis(2, outer = T)
mtext(side = 2, line = 2, outer = T,
  text = expression("Sediment yield (" ~ tonnes ~ " * km^{-2} ~ ")"))
)
```

As a final example, we will generate a “stack” of the three Sediment yield time series.

```
axis(side = 1)
mtext(side = 2, outer = T, line = 0, text = expression("Sediment yield (" ~ tonnes ~ " " ~ km ~ {"-2"} ~ " ~ ")))
legend("topleft", bty = "n", legend = "Flynn Creek")
abline( v = c(1965.5, 1966.5), lty = 1, col = "grey" )
```

