Statistical Computing

Master Data Science

Winter Semester 2017/18 **Prof. Tim Downie**

BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

Workshop 2 — Matrices and Graphics

Introduction

Last Week you used RStudio for the first time and learnt some basic principles of R. The most important aspects were:

`<-`	assigns the value (right hand side) to an object (left hand side)
`[...]`	is used to access an element of a range of elements in a vector object and

functions are called using round brackets with arguments specified inside the brackets e.g. `log(3, base=10)`

Today's main theme is an introduction to Graphics in R. We will be using graphics throughout this course and in most of the statistics courses, and there is a course on "Data Visualisation" in the second semester. First of all there is a section on Matrices and Arrays, extending the concept of vectors in R, which you met last week.

A break is scheduled between 2pm and quarter past.

Preliminaries

► Start RStudio using the Icon on the desktop

► Strg+Shift+N opens a new R script.

► Type the comment

```
#Statistical Computing: Workshop 2 - Matrices and Graphics
```

as a title.

► Save the file in your `H:\\StatComp` folder with the name `Workshop2.R` using *File > Save as*

Work through the worksheet entering the commands and examples as you did last week. In many of the code examples the output is not included, you should always see what the output is and make sure you understand it. Please add comments (in your own words) to your code so that it will make sense to you when you return several weeks (or years) later. Finally, **please do not copy and paste the commands from this PDF file**, type them in yourself. Copy/paste is the quickest way to forget what you have just done!

1 Matrices and Arrays

1.1 Matrices

A matrix is a two dimensional “table” of numbers. In mathematical notation for a matrix is

$$M = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

M is a matrix with 2 rows and 3 columns. $M_{1,3}$ is the element of M in the 1st row and 3rd column and equals 5.

The R code for this uses the `matrix()` function:

```
> M <- matrix(1:6,nrow=2)
> M
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Note that in the vector `1:6`=$(1, 2, 3, 4, 5, 6)$ is assigned to the matrix by column. `matrix()` takes an option called `byrow=TRUE` if you want to assign the values by row.

```
> M <- matrix(1:6,nrow=2,byrow=TRUE)
> M
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

You specify the number of rows and/or the number of columns using `nrow` or `ncol`, you probably want to supply at least one of them. `dim()` returns the number of rows and columns of a matrix:

```
> M <- matrix(1:6,nrow=2)
> M
> dim(M)
> M <- matrix(1:6,ncol=3)
> M
> dim(M)
> M <- matrix(1:6)
> M
> dim(M)
```

Subsetting a Matrix A subset of elements in a vector can be accessed using square brackets. An example from last week was:

```
> fibonacci<-c(1,1,2,3,5,8,13,21,34,55)
> fibonacci[7]
```

The elements of a matrix are identified by the row and the column, so the square bracket notation for a matrix requires two indices separated by a comma `[.,.]`. The row index always comes before the column index.

```
> M <- matrix(1:6,nrow=2)
> M[1,3]
[1] 5
```

You can specify multiple rows and/or columns:

```
> M[1:2 , 1]              : 1st and 2nd rows, 1st column
> M[2 , c(1,3)]           : 2nd row, 1st and 3rd columns
> M[1:2 , c(1,3)]         : 1st and 2nd rows, 1st and 3rd columns
```

The subsetting of vectors, matrices, arrays and data frames is a very useful tool and we will be using it a lot.

To get one complete row of a matrix omit the term after the comma:

```
> M[2, ]
```

To get one complete column of a matrix omit the term before the comma:

```
> M[,3]
```

Note that in the last example R returns the column as a vector. This is usually what one wants so is the default behaviour.

To drop one row of a matrix use a minus sign in the row field:

```
> M[-2, ]                 : all rows except the 2nd, all columns
```

To drop one column of a matrix use a minus sign in the column field:

```
> M[, -1]                 : all rows, all columns except the 1st
```

You can even specify the rows or columns using an R-object `> x<-c(1,3)`

```
> M[,x]                   : all rows, 1st and 3rd columns.
```

Matrix arithmetic Matrices have their special arithmetic rules. Addition of two matrices is as one would expect:

$$\text{If } A = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix} \text{ and } B = \begin{bmatrix} 5 & 7 & 9 \\ 6 & 8 & 10 \end{bmatrix}, \text{ then } A + B = \begin{bmatrix} 6 & 10 & 14 \\ 8 & 12 & 16 \end{bmatrix}.$$

Which is what + in R does:

```
> A <- matrix(1:6,nrow=2)
> dim(A)
> B <- matrix(5:10,nrow=2)
> dim(B)
> A+B
     [,1] [,2] [,3]
[1,]    6   10   14
[2,]    8   12   16
```

Multiplication of a scalar and a matrix is also straightforward:

$$2A = \begin{bmatrix} 2 & 6 & 10 \\ 4 & 8 & 12 \end{bmatrix}.$$

```
> 2*A
     [,1] [,2] [,3]
[1,]    2    6   10
[2,]    4    8   12
```

However, **the multiplication of two matrices follows special rules**:

$$\text{If } A = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix} \text{ and } C = \begin{bmatrix} 5 & 8 \\ 6 & 9 \\ 7 & 10 \end{bmatrix}, \text{ then } A \times C = \begin{bmatrix} 58 & 85 \\ 76 & 112 \end{bmatrix}.$$

In R `A*B` performs elementwise multiplication and `A%*%B` performs the matrix multiplication used in mathematics.

```
> A <- matrix(1:6,nrow=2)
> B <- matrix(5:10,nrow=2)
> C <- matrix(5:10,nrow=3)
> A*B
     [,1] [,2] [,3]
[1,]    5   21   45
[2,]   12   32   60
> dim(A)
[1] 2 3
> dim(C)
[1] 3 2
> A%*%C
     [,1] [,2]
[1,]   58   85
[2,]   76  112
>
```

`t` transposes a matrix and `solve` computes the inverse.

```
> t(A)
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
> D <- matrix(1:4,nrow=2)
> solve(D)
     [,1] [,2]
[1,]   -2  1.5
[2,]    1 -0.5
> D%*%solve(D)
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

There are many more functions which apply to matrices, you will learn them when they are needed.

1.2 Data Frames (revisited)

In the Further Reading document, the concept of a data frame was introduced. It is easy to think that a data frame is a type of matrix because it stores data in a rectangular format. There are similarities, but they are not the same:

- All entries in a *matrix* must be numeric, otherwise matrix addition and multiplication won't work.
- A *data frame* consists of a number of variables and each variable is represented as a column in the data frame. The length of each variable is equal.
- A *data frame* can have a mix of "data types": numeric, factor, logical or character. All the elements of one variable must have the same "data type".

The main similarity between a matrix and a data frame is that they are both represented as rows and columns and the `[.,.]` notation can be used for both.
Example

```
> data(iris)
> iris[10,]  #returns the 10th row of the iris data frame
> iris[1:10,4] #first 10 rows of the 4th variable in iris
> iris[1:10,"Species"] #first 10 rows of the Species variable
```

The following three commands all give the same output

```
> iris[1:10,4]
> iris[1:10,"Petal.Width"]
> iris$Petal.Width[1:10]
```

1.3 Arrays

An array is a generalisation of a matrix. A matrix is 2-dimensional. An Array is n-dimensional, where n is a positive integer (usually $n \geqslant 3$). A 3-dimensional array can be visualised as a cube (or cuboid) consisting of numbers.

Formally, an array is a collection of data entries, indexed via one or more subscripts. The arguments for defining an array are: (1) a vector of data, and (2) a dimension vector giving the number of elements in each dimension.

Example

```
> x <- array(1:4,dim=c(4)) #1-dim array (like a vector)
> x
> is.array(x)
> is.vector(x)
> x <- array(1:4,dim=c(2,2)) #2-dim array (like a matrix)
> x
> x <- array(1:27,dim=c(3,3,3))
> dim(x)
> x[1,1,1]
> x[2,,]
```

Notice that you might expect that `dim(x)` return how many dimensions are in the array `x`, but it returns the size of each dimension. To find how many dimensions it has use:

```
> length(dim(x))
```

2 Graphics in R Introduction and Preliminaries

The ability to represent data graphically is an important part of data analysis for the following reasons:

- at the start of an analysis, investigating the data visually helps you to get to know the data that you will be working with. It can often be a good way of identifying data errors or outliers.
- As part of the analysis, graphs can be used to check the validity of any assumptions made (diagnostic plots) and to interpret the results of the analysis.
- To communicate your findings to other people in informal discussions, presentations or written reports (such as your Master Project).

R has a wide range of plotting capabilities and the graphics can be customised extensively. In this workshop we will cover many of the important basics, including enhancing plots by adding additional graphics and text, and using parameters to customise your graphics.

This section covers the “traditional” graphics functions in R. There are many packages in R which allow you to extend the capabilities of the software. Some enhance the visual investigation of data to observe characteristics of the data that might not be obvious from a non-visual investigation. An example of this is the Trellis Graphics package.

Other extensions enable more complex and sophisticated graphics such as textured surfaces or 3-D Graphics. In your courses *Data Visualisation* and *Machine Learning* you will be using the package `ggplot2`. If you’re curious to the graphical capabilities in R have a look at The R Graph Gallery (after the Workshop!)

In addition to the given examples and exercises in these notes, **you are encouraged to try out variations to the given commands**, observing how changes to the arguments alter the plots. This is especially relevant in Section 4 “Enhancing and Adapting Graphics”. You can also read more about functions in the help pages `?name.of.function`.

3 High-Level Plots

3.1 The `plot()` function

As you saw last week, there is a function in R called `plot()`. In its simple form the x- and y-axis variables are specified as arguments, and a diagram is shown in the *Plots* tab bottom left.

E.g. Define two vectors called `xvalues` and `quadx` and then plot a scatter plot.

```
> xvalues<- -5:5
> quadx<-xvalues^2
> plot(xvalues,quadx,type="p")
```

You can click on *Zoom* in this window and RStudio opens up a separate window with higher quality graphics. Note that sometimes axis labels etc. are hidden in the smaller window inside the RStudio window, so if you are getting unexpected results click on zoom to see if the problem only appears in the small version.

The argument `type="p"` specifies that “points” should be plotted (the default). The other values that the `type` argument can take are `"l"` lines, `"b"` both (lines and points), `"s"` stepped, `"h"` “hi-density” (vertical lines from the x-axis to the points), and `"n"` none – only plot the axes (you will see why this is useful later).

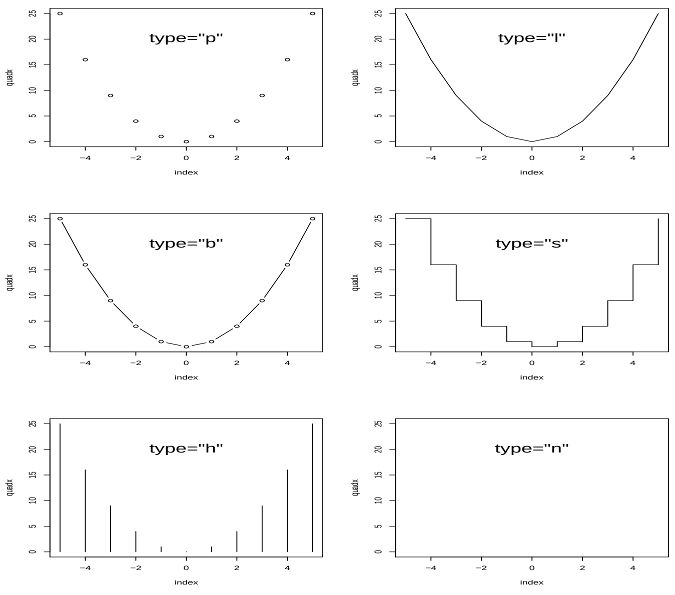

Figure 1: Some different “types” of plot

Exercises:

1. Repeat this plot command with different settings for `type`. Figure 1 shows some of the different “types” of plot.

2. The object `quadx` is used as an example. You should try plotting different functions such as `exp(xvalues)`, `abs(xvalues)` and `sin(xvalues)`.

3. Now try to plot `log(xvalues)` against `xvalues`, what happens and why?

4. The function `rnorm` generates a sample of random numbers that come from a standard normal distribution. Plot 10 random standard normal numbers. Repeat the command several times.

   ```
   > plot(rnorm(10))
   ```

`rnorm(10)` supplies the y-axis values. The x-axis values are just the index numbers 1 to 10. This is the default behaviour when only one vector is passed into `plot()` Now use this function to plot 50 standard normal observations; what is the range of the y-axis?

A standard normal distribution has mean 0 and standard deviation 1. You can specify different values for a normal distribution, e.g.

```
> plot(rnorm(50,mean=10,sd=0.5))
```

Play about with these arguments noting the differences in y-axis between the plots.

Notice that every time you enter a plot command, the graphics window is cleared and a new graphic is drawn, as opposed to adding to or changing the existing plot. This is because `plot` is a "high-level plot" command.

Other types of high-level plot use different commands like those given below. If you are interested in what the datasets mean then read-up on them using the `help()` command.

3.2 Bar Charts

Bar charts can be obtained for qualitative data (usually a factor variable). The argument supplied should be a vector of the heights or the frequencies for each bar. The classic bar chart uses one variable, but you can plot two variables simultaneously using stacked or grouped bar plots.

The following example uses the `VADeaths` dataset which is directly available in R. The dataset contains information about the death rates in the US state ov Virginia. It is stored as a matrix with row and colum names. Quickly read the help file, before plotting some bar charts.

```
> help(VADeaths)
> VADeaths
> class(VADeaths)
> barplot(VADeaths["50-54",])
> barplot(VADeaths)
> barplot(VADeaths,beside=TRUE)
> t(VADeaths)
> barplot(t(VADeaths))
```

N.B. In the workshop last week you used the `data()` to access the `iris` dataset. In the past this was necessary, but recent versions it is optional. Using data `data()` puts the dataset into your working directory, and so makes the object visible in your Environment window.

3.3 Histograms

At first sight a histogram looks like a bar chart, but the difference is that in a histogram the variable is numeric (not factor). The first argument to `hist` should be a numeric vector containing the individual observations.

```
> hist(rnorm(1000))
> hist(rexp(1000,4))
```

Question: What do these two histograms show?

Experiment with the arguments `breaks=n`, where `n` is a number of your choice, and `prob=TRUE`, observing how the resulting histograms differ.

3.4 Boxplots

A boxplot (a.k.a. box-and-whisker plot) is used to compare a numeric variable across several groups in a data frame. The exact format of a boxplot differs between packages; in R the box is defined by the lower quartile, median and upper quartile, while the "whiskers" extend to the data point that is no more than 1.5 times the interquartile range from the box. Any observation outside the whiskers is defined as an *extreme value* (not necessarily an outlier!) and is shown as a point.

```
> ?InsectSprays
> boxplot(count ~ spray, data = InsectSprays)
> boxplot(count ~ spray, data = InsectSprays,
+ col = "lightblue",pch=16)
```

Notice the strange use of the tilde sign (~) in: `count~sprays`. This is called an R *model formula*. Model formulae are common in model fitting, and the details are not important now; you will come accross them later in the course. For now, read the formula as "`count` *depends on* `spray`" using the data frame `InsectSprays`.

3.5 Quantile plots

A quantile plot (or Q-Q plot) allows the visual assessment of how well a continuous variable fits a known distribution (usually the normal distribution). `qqnorm(x)` plots the ordered values of `x` against quantiles of a standard normal distribution.

In an "ideal" plot the points fall on a a diagonal straight line. To see how close to a straight line the points are you can use `qqline()`

What is a quantile? Choose a value p between 0 and 1. The p-th quantile of a data-variable is the value, whereby a proportion p of the values are less than or equal to the quantile. It's easier to explain with an example:

Example

```
> sum(iris$Petal.Length<5.5)
[1] 122
> sum(iris$Petal.Length<5.5)/length(iris$Petal.Length)
[1] 0.8133333
```

So 122 flowers had a petal length of 5.5 cm or less which is the proportion $p = 0.813$. The 0.813 quantile for petal length is 5.5cm.

Example of a quantile plot We will produce a quantile plot of petal length for the *virginica* flowers. First we need to find `which` rows of the data frame are of this species.

```
> elem<-which(iris$Species=="virginica")
> elem
> qqnorm(iris$Petal.Length[elem])
> qqline(iris$Petal.Length[elem])
```

The points in the middle of the diagram fit the reference line very well. The points on the left hand side of the diagram are all a bit above the reference line, and to a lesser extent on the right hand side too. In practice the "tails" of a Q-Q plot rarely lie very close to the reference line and this plot is quite good. We can conclude that the petal lengths for virginica irises roughly fit a normal distribution.

Repeat the Q-Q plot for all three species of iris. You should now see that the petal lengths for all 150 flowers do not follow a normal distribution.

3.6 Plotting Multivariate Data

The `matplot()` function plots a each column of a matrix or data frame as a different series

```
> data(USArrests)
> help(USArrests)
> matplot(USArrests,type="l")
```

The x-axis here is state ordered alphabetically. It might make more sense to order one variable by increasing value (in the following example number of assaults per 100 000 population) and reorder all series accordingly

```
> USPopRank<-order(USArrests$Assault)
> matplot(USArrests[USPopRank,],type="l")
```

3.7 Pairwise plots

This plots each pairwise combination of variables in a data frame as below

```
> pairs(USArrests)
```

Can you suggest a problem with this type of plot when there are a large number of variables?

3.8 Condition Plots

A condition plot creates several “panels” of scatter plots. The panel in which each point is plotted is defined be a third variable called the conditioning variable. That third variable can be a factor variable, or a partition of a continuous variable.

```
> plot(iris$Sepal.Length,iris$Petal.Length)
> coplot(Petal.Length~Sepal.Length | Species,data=iris)
```

Notice again the formula notation using the Tilda. This time read the formula
`Petal.Length~Sepal.Length | Species` as
“`Petal.Length` *depends on* `Sepal.Length` *given* `Species`.”

An example of conditioning on a continuous variable is given by

```
> data(swiss)
> names(swiss)
[1] "Fertility"        "Agriculture"      "Examination"      "Education"
[5] "Catholic"         "Infant.Mortality"
> coplot(Fertility~Education|Agriculture, data=swiss,overlap=0)
> coplot(Fertility~Education|Agriculture, data=swiss,overlap=0.5)
> coplot(Fertility~Education|Agriculture, data=swiss,overlap=0.5,
+ panel=panel.smooth)
```

The panels are ordered bottom left (smallest values of Agriculture), bottom middle, bottom right,. . ., top right (largest values of Agriculture).
Question: What do you think the arguments `overlap` and `panel=panel.smooth` do?

There are many different types of condition plots available from the R library called *lattice*.

3.9 The `plot()` function for different classes

There are actually many different versions of the `plot()` function. When `plot` is called, the class of the first object is considered. This allows a graphic particularly suited to this type of data to be plotted. Suppose that object is of class `"data.frame"`, then a function named `plot.data.frame()` will be called, and a pairs plot will be plotted.

If instead the object has class `"lm"` (an object produced by fitting a linear model to the data), then `plot.lm()` is called and diagnostic plots will be produced.

If the class of an object does not match up with any R function of the form `plot.something()`, then `plot.default()` is used, which is the function you have been using so far.

```
> class(swiss)
[1] "data.frame"
> plot(swiss)
> data(state)
> class(state.division)
[1] "factor"
> plot(state.division)
> class(xvalues)
[1] "integer"
> plot.default(xvalues,quadx)
```

4 Enhancing and Adapting Graphics

Often we wish to customise a plot in some way. There are generally 3 different ways to do this:

- by using arguments supplied to the high-level plotting function,
- by using other (specific) functions after the high-level plot has been created or
- by changing the default settings using the function `par()`.

4.1 Arguments

There are many different arguments to enhance graphics. You are not expected to learn all of them in one go. Just trying out different things is a good way to get to know the possibilities.

Arguments that can be specified in most high-level plot functions include:

`main`	main title for the plot
`sub`	subtitle at bottom of the plot
`xlab`	label for x-axis
`ylab`	label for y-axis
`xlim`	the (approximate) minimum and maximum value for the x axis
`ylim`	the (approximate) minimum and maximum value for the y axis
`col`	the plotting colour
`col.axis`, `col.lab`, `col.main`, `col.sub`	colour for axes, labels, etc.
`lty`	line type; `lty=1` gives a solid line, other values are various combinations of dash/dotted
`pch`	plotting symbol; can be a number or a character (see below)
`log`	plot the axes using a log scale (one of `"x"`,`"y"`,`"xy"` or `""`)
`cex`	a number giving relative size of text (character expansion)
`axes`	`TRUE` or `FALSE`, if `FALSE` then no axes are displayed

For example,

```
> xvalues<- -5:5
> quadx<-xvalues^2
> plot(xvalues,quadx,type="b",main="this is a title",
+ sub="this is the subtitle",xlab="x", ylab="x squared",col.main="green",
+ col="blue",lty=2,pch="*",cex=3)
> plot(xvalues,quadx,xlim=c(-10,10))
> plot(xvalues,quadx,xlim=c(-10,10),axes=FALSE)
```

There are also some arguments that can only be used with a specific type of high-level plot. Examples of this are `beside` for `barplot()`, and `breaks` for `hist()`.

The value of `pch` can be a character, usually `"o"`, `"+"`, `"*"` or `"."`, or it can be a number. Figure 2 shows which number corresponds to which symbol.

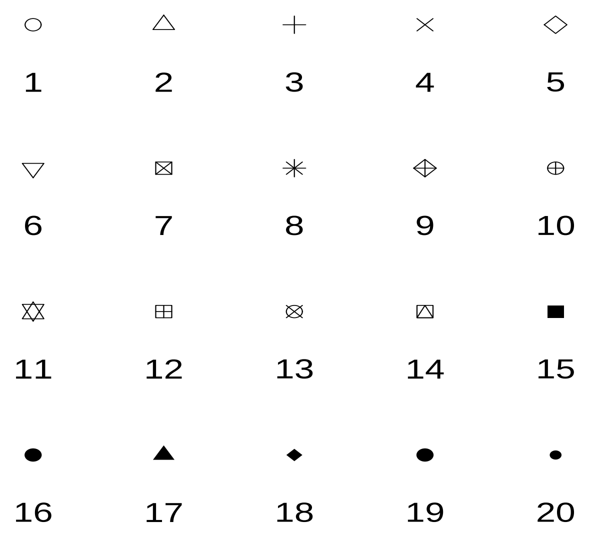

Figure 2: The plotting symbols with `pch=n` for `points()` (or `plot(...,type="p"))`.

A function that is useful in conjunction with plot commands is `jitter()`. This adds a small amount of noise to the vector argument. When applied to a plot, it can prevent overlaying points that have very similar values.

```
> ?morley
> plot(morley$Expt,morley$Speed)
```

It is difficut to assess how many observations are in ‘Experiment 5’. We can be sure there is no *overplotting* by using `jitter()`.

```
> plot(jitter(morley$Expt),morley$Speed)
```

There are other ways to do this, the `symbols()` function will plot symbols whose radii are specified by a vector. Another method is to use sunflower plot, which adds petals to multiple observations:

```
> sunflowerplot(morley$Expt,morley$Speed)
```

Exercise

1. Define an object that is the exponential of `xvalues` using the function `exp()`.
2. Plot that object against `xvalues` using a plotting character of a + sign, and joined by lines.
3. Now do the same plot with the y-axis on the log scale.

4.2 Adding to existing plots

Some functions that *add graphics to an existing diagram* are:

`points()`	like `plot` with `type="p"` but adds to the existing plot
`lines()`	like `plot` with `type="l"` but adds to the existing plot
`matpoints()`	like `matplot` with `type="p"` but adds to the existing plot
`matlines()`	like `matplot` with `type="l"` but adds to the existing plot
`title()`	adds title, subtitle and/or labels etc.
`axis()`	draw the axes; useful if you want to specify different tick marks or numbers on the axis
`abline()`	draws a straight line
`text()`	adds text at a specified point
`legend()`	adds a legend to the plot
`qqline()`	is used specifically for Q-Q plots to add a reference diagonal line to the plot.

The `mtcars` dataset contains data on different North American cars. Find out about this dataset and what the variables are by reading the help page:

```
> ?mtcars
```

We can easily plot the fuel economy against the weight of the car using:

```
> plot(mtcars$wt,mtcars$mpg)
```

We can also show how the number of cylinders in the engine relates to the fuel economy and weight of car.

First define logical vectors to identify the cars that have four, six and eight cylinders respectively.

```
> fourcyl<-mtcars[,"cyl"]==4
> sixcyl<-mtcars[,"cyl"]==6
> eightcyl<-mtcars[,"cyl"]==8
```

Now plot each subset separately. Look at the plot **after typing each command** so you can see what each command does.

```
> plot(mtcars[,"wt"],mtcars[,"mpg"],type="n")
> points(mtcars[fourcyl,"wt"],mtcars[fourcyl,"mpg"],pch="+",col="blue")
> points(mtcars[sixcyl,"wt"],mtcars[sixcyl,"mpg"],pch="o",col="red")
> points(mtcars[eightcyl,"wt"],mtcars[eightcyl,"mpg"],pch="x",col="magenta")
```

Note that we used the argument `type="n"` in the plot function. This was to set up the plotting area to match the entire dataset. We then used `points()` to plot each subset separately.

Question: What would have happened if you had omitted the first `plot(...,type="n")` function and instead used
`plot(mtcars[fourcyl,"wt"],mtcars[fourcyl,"mpg"],pch="+",col="blue")`
directly?

Now we can add a line that shows the median fuel economy, and a legend

```
> abline(h=median(mtcars[,"mpg"]),col="green")
> legend(1.5,14.0,legend=c("Four Cylinder","Six Cylinder","Eight Cylinder"),
+ col=c("blue","red","magenta"),pch=c("+","o","x"))
```

`abline()` can take several different arguments: `h` specifies a horizontal line, `v` specifies a vertical line, `a, b` specify the intercept and slope and `coeff` specifies a vector of length 2 containing the intercept and slope.

The first two arguments of `legend()` are the x and y coordinates of the top left hand corner of the legend, the argument `legend` is a vector of labels, and `pch` is a vector of plotting symbols. `lty` should be used instead of `pch` when adding a legend to a line plot.

4.3 Using Colours

The choice of colours available in R is extensive, the full list can be found using the

```
> colors()
```

command.

There are some commands which make using colours easier. `palette()` is used to assign colours to the values 1 to 7 (or to whatever number you want). With no argument `palette()` returns the current values, with an argument the colour assignment is changed. The default colours for a grouped bar chart are grey levels. Below we change the colours of the bars using the default colours and then *rainbow* colours. Finally the palette is returned to the default settings.

```
> barplot(VADeaths,beside=TRUE)
> palette()
> barplot(VADeaths,beside=TRUE,col=1:5)
> palette(rainbow(7))
> palette()
> barplot(VADeaths,beside=TRUE,col=1:5)
> palette("default")
> palette()
```

4.4 An Example of Using `axis()`

The following example places the axes through the origin.

```
> plot(xvalues,quadx,axes=FALSE)
> axis(side=1,pos=0,at=c(-5,0,5))
> axis(side=2,pos=0,at=c(0,5,10,15,20,25),
+  labels=c("","5","10","15","20","25"),las=1)
> box()
```

Careful that last axis argument is “las equals one” (label axis style).

The `axes=FALSE` argument means that no axes are printed. In the command `axis()`, `side` takes the value 1,2,3 or 4 for bottom, left, top and right axis respectively. The `pos=0` statement places the axis at zero, whereas normally it is at the minimum value of y (or x). On the y-axis the `"0"` label would normally interfere with the x axis, so we change it’s label to be `""`.
Question: what would happen if you just specified `at=c(5,10,15,20,25)`?
The `las=1` argument tells R to place the labels horizontally (not the default, which is parallel to the axis).

4.5 The `par()` Function

The `par()` function allows you to set many graphics parameters, and those settings will remain until you finish the session, or until you reset the parameters. Many of the arguments that you have already met when specifying them in plotting commands can be changed permanently, by using them as arguments to `par()`. These include:
`col, col.axis, col.lab, col.main, col.sub, lty, pch, log, cex, axes` and `las`. Others arguments include:
`bg`, which specifies the changes the background colour,
`font` allows bold and italic characters,
and `mfrow`.

The latter parameter allows you to specify multiple plots in the same window. For example,

```
>par(mfrow=c(1,2))
```

will put two plots side by side, i.e. using one row and two columns. To see the effect create any two plots, such as:

```
> plot(mtcars[,"wt"],mtcars[,"mpg"])
> plot(xvalues,quadx,type="b")
```

Figure 1 has a 3 by 2 layout, which was specified using

```
> par(mfrow=c(3,2))
```

To reset the layout to just one plot on the page use

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

5 Graphics Devices

The window in which you see the plot is called a Graphics Device. Specifically it is an *RStudio graphics device* `RStudioGD` which was started when you entered the first high level plot command. You can open up a new device with the command

```
> RStudioGD()
```

Notice that the new device is labelled "ACTIVE". Enter a simple graphics command to see that the plot appears on the new (active) device.

`> dev.list()` returns a list of all the devices that are open
`> dev.set(2)` changes the active device to device 2. Enter another plot command to verify this.
`> dev.off()` try it.

You've just killed the graphics device in the RStudio window and the external window is now the current device. You can open a *new* device in the RStudio window, by again using

```
> RStudioGD()
```

You can save a graphics window using the *Export* button in the Plot window. As usual it is more convenient to use R commands to do this.

My preferred method is to finalise all the graphics commands, and then add the command

```
> pdf(file="Pathname/filename.pdf")
```

before the graphics commands followed by
`> dev.off()` which finalises the PDF file. Note that you do not see the graphics on the screen, they are sent directly to the `pdf` device. `pdf()` opens up a new graphics device but one that writes to a PDF file rather than to the screen. Then check the PDF file, as some times there are small differences in the RStudioGD display and the PDF display. I do this for each diagram using a different file name each time.

There is an option for `pdf()` called `onefile=TRUE` which allows you to put multiple diagrams in one file, there are other options to change the height, width and other properties of the PDF page.

There are similar commands for other types of graphics files, the most useful are `png()` and `jpeg()`.

General comments:

1. The graphics devices can be operating system dependent and can change over time. The important thing is to read the help files and experiment to get the fine tuning right.

2. As the name suggests `RStudioGD` is specific to RStudio. If you use R on its own, then the standard type of graphics window is called `x11` and a new window can be created using `x11()`. The differences between the two are only relevant for very advanced users.

6 Exercises

If you have time at the end of the Workshop try out these exercises.

(a) Use R to calculate the matrix multiplication $M_1 \times M_2$, when

$$M_1 = \begin{bmatrix} 1 & 2 & 5 \\ 3 & 1 & 6 \\ 7 & 2 & 8 \end{bmatrix} \qquad M_2 = \begin{bmatrix} 1 & -1 \\ -6 & 2 \\ 1 & 1 \end{bmatrix}$$

(b) (i) Create a vector called `theta` that takes values from zero to 2π and has length 20. Do this using the command `seq(from=??, to=??, length=??)` putting in the appropriate values (there is an object in R called `pi`). Check your code by typing:

```
> length(theta)
> range(theta)
```

(ii) Create a plot of $\sin(\theta)$ against θ using a red coloured triangles as symbols.[1] Overlay a plot of $\cos(\theta)$ against θ using green coloured diamonds.

(iii) Insert an appropriate legend to the current plot.

(iv) Repeat the above plot but now plotting $\sin(2\theta)$ against θ and $2\cos(\theta)$ against θ. You will need to change the axes to do this.

(v) Can you predict what you get if you plot $\sin(\theta)$ against $\cos(\theta)$? Check your answer by creating this plot in R.

(c) Create a sample of fifty normal $N(100, 25)$ random variables using

```
> nsamp<-rnorm(50,mean=100,sd=5)
```

Now create a Q-Q plot and a reference line for `nsamp`. Hopefully the points lie near to the line.

(d) Create a sample of fifty χ_1^2 random variables using

```
> chisamp<-rchisq(50,1)
```

Now create a Q-Q plot and a reference line for `chisamp`.

Not surprisingly the data does not fit a normal distribution.

By making Q-Q plots of powers of `chi`, eg.

```
> qqnorm(chi^2)
```

choose the power that you think makes the sample approximately normal.

(e) Recreate Figure 1.

(f) Recreate Figure 2 (difficult).

[1]The trigonometric functions in R expect the angle to be given in radians

7 Tidying up

Make sure your script file has sensible comments. At least one comment for each main section.

► Save the script file (source file) again: $\boxed{\text{Strg + S}}$ or *File > Save as*.

► Leave RStudio by typing the command:

```
> q()
```

When R asks you *Save workspace image ...?*, click on **Don't save**!

► Feierabend!