The R statistical package: A tutorial

Daniel Walsh

2015

R is a free, open source, programming environment for data analysis and graphics. R is available for all computing platforms and can be downloaded, along with many other specialised user contributed packages, from `http://cran.stat.auckland.ac.nz/`. More information about learning R is available from the `http://www.r-project.org/`, the main R website. The searchable R help mailing list, `http://cran.r-project.org/search.html`, is an extremely useful resource.

1 History

R was first developed at the University of Auckland by Ross Ihaka and Robert Gentlemen as an efficient version of the S language. Since then it has become a large, successful, open source project, which many of the world's leading statisticians contribute to. It is a high quality product suitable for: statistical data analysis; production of detailed graphics; stochastic simulation; and programming customised algorithms. Moreover it is free. R does not inhibit the user with restrictive, expensive licenses; one can install R on as many computers as one wishes without loss of any functionality, and one will be able to use R in any future employment. There is a moderate learning curve involved; however, the benefits of learning R quickly outweigh any costs.

2 Basics

In this section we cover the basics of using R.

2.1 Starting

To start on the University Lab computers, select `Start` → `All Programs` → R → `R i386 3.12.exe` (the number may be different). If you are working on a Linux machine or an Apple computer which has R installed on it you can simply type R at the command line prompt.

2.2 Working directory

Once R is started you need to set your *working directory*. This is the default directory where your R session will be saved; where files will be written to; and where R will look to read data files and functions. On a windows machine, select File → Change dir..., and set the working directory to a location in your My Documents directory. If you started R from the command line, your current directory will be your working directory.

2.3 Libraries

Some R functions and data are not loaded by default upon start up, but are stored in *libraries* instead. To load these functions, we use the library command. For example, to load the lattice graphics package we type:

```
> library(lattice)
```

2.4 User contributed packages

R has hundreds of packages written for it by R users from around the world. One can use the install.packages() command to install these packages. A list of available packages can be found at: http://cran.stat.auckland.ac.nz/web/packages.

For example, to install the maps() package, one should just type:

```
> install.packages("maps")
```

It will request that you select a download site, then download the package and install it. (If the command doesn't work, try using the installing via the Packages & Data menu.) To use the package (each time you start R), simply type:

```
> library("maps")
```

Unfortunately, due to firewalls and other security precautions, user contributed packages either cannot be installed on University computers, or they must be installed each session.

2.5 Calculator

R can be used as an calculator. The assignment operator is represented by = or <-. Commands on the same line can be separated by a semi-colon.

```
> 1 + 1
[1] 2
> a = 4; b <- 3
```

```
> b/a
[1] 0.75
```

2.6 Editing Commands

The "up" and "down" arrows scroll back and forth through previous commands, while the "left" and "right" arrows moves the cursor along the current command. Copy, cut, and paste com-

mands are the standard `Control-C`, `X`, `V` in Windows, and `Apple-C`, `V`, `X` on Macs. Emacs style commands are also available.

`Control-A`	(Start) Return cursor to start of the line.
`Control-B`	(Back) Move cursor backward (Left Arrow).
`Control-D`	(Delete) Delete to the left of the cursor.
`Control-E`	(End) Move cursor to the end of the line.
`Control-F`	(Forward) Move cursor forward (Right Arrow).
`Control-H`	(Backspace) Delete to the right of the cursor.
`Control-K`	(Kill) Clear line after cursor.
`Control-N`	(Next) Next command (Down Arrow).
`Control-P`	(Previous) Previous command (Up Arrow).

2.7 Objects

When a variable is assigned a value it exists in memory as an "object". We can list all objects created with the `ls` command.

```
> ls()

[1] "a" "b"
```

We can remove objects using the `rm` command.

```
> A <- 100
> ls()

[1] "a" "A" "b"
```

```
> rm(A)
> ls()

[1] "a" "b"
```

2.8 Quitting

To quit `R` select `File` → `Exit`. `R` will ask us if we want to save the workspace image; this means it will save all the objects we've created in a file called `.RData` in our working directory that we can open later, and a text file listing of the `R` commands typed during the session called `.Rhistory`. Clicking on the `.RData` file will open `R` and restore our previous `R` session.

We can also quit using the `q` command.

```
> q()
> q(save="yes")
```

```
> q(save="no")
```

2.9 Saving and Restoring the Workspace

To restart `R` and open the previously saved workspace, we can double click on the `.RData` file, or run `R` from the Start Menu, change the working directory, and then load the old R session, by using the `File` → `Load workspace...` menu, or by using the `load` command.

```
> load("C:/Documents and Settings/User/My Documents/.RData")
```

We can save the workspace without quitting by using the `save.image` command. Alternatively we can save selected objects with the `save` command.

```
> a <- 1; b <- 2; d <- 3
> save.image()
```

```
> save(a,d,file="AD.RData")
```

2.10 Function Syntax

All R commands are R functions; functions have the following syntax:

```
result <- function(arg1,arg2,...,
                   optional_arg1=value1,optional_arg2=value2,...)
```

The optional arguments do not have to be supplied, as they have default values. For example, log returns the logarithm for the natural base $e = 2.718282$.

```
> log(100)
[1] 4.60517
> log(100, base=exp(1))
```

```
[1] 4.60517
> log(100, base=10)
[1] 2
```

If the function is run without the parentheses (), the function call (i.e. function code) is returned.

```
> log
function (x, base = exp(1))
if (missing(base)) .Internal(log(x)) else .Internal(log(x, base))
<environment: namespace:base>
```

2.11 Help

Help menu are available in the Windows version from the Help menu, including manuals in PDF format. From the command line, one can access the main help menu via:

```
> help.start()
```

Help for a particular command can be accessed via the help or ? commands.

```
> help("sqrt")
> ?"sqrt"
```

Note that for most functions, ?"function" and ?function both display the help file for function, however some functions e.g. if and for, the quotation marks are needed.

```
> ?"if"
> ?"for"
```

2.12 Comments

Comments are denoted by the # symbols. Anything after a # symbols is ignored by R.

```
> ## This is a comment
> A <- 3; # A is assigned the value 3
```

2.13 Demo

There are several demonstrations available, which can be listed by using the `demo()` command.

```
> demo(graphics)
> demo(image)
```

2.14 Calculator Functions

Many standard functions are built-in.

```
> a <- 3
> b <- 4
> a^2

[1] 9

> (b * a^3)/a

[1] 36

> sqrt(16)

[1] 4

> pi

[1] 3.141593

> round(pi,3)

[1] 3.142

> ceiling(pi)

[1] 4

> cos(pi)

[1] -1

> tan(pi/4)

[1] 1

> log(10)

[1] 2.302585

> exp(2)

[1] 7.389056

> abs(-103)

[1] 103
```

3 Understanding R Objects

Using `R` involves working with objects. There are different *classes* of objects including: `character`, `integer`, `numeric`, `vector`, `matrix`, `array`, `data.frame`, `list`, `lm` (linear model), and `NULL` (empty object). An object may belong to several classes at once; e.g. consider the object `x` created below:

```
> x <- 1:6
> x

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

Here we have used the colon operator to create a vector containing the integers from 1 to 6. We can use the function `is()` to display all the classes a variable belongs to, and we can use the function `class()` to display the main class.

```
> is(x)

[1] "integer"             "numeric"             "vector"
[4] "data.frameRowLabels"
```

```
> class(x)

[1] "integer"
```

Thus, `x` is a numeric vector, containing integers.

3.1 Vectors

Vector are one of the basic object types in `R`. Vectors elements are accessed by square brackets, `[]`. Vectors can be created by combining elements with the `c()` function. Vector can also be created with the `seq` (sequence) function or with the colon (`:`) operator (as above).

```
> x <- 11:16
> x

[1] 11 12 13 14 15 16

> x[3:6]

[1] 13 14 15 16

> x[-1]

[1] 12 13 14 15 16

> c(1,3,5)

[1] 1 3 5

> x[c(1,3,5)]

[1] 11 13 15

> -c(1,3,5)

[1] -1 -3 -5

> x[-c(1,3,5)]

[1] 12 14 16

> x[c(1,1)]

[1] 11 11
```

Notice how we can access elements of the vector using the square brackets and a vector of indices, and how we can remove elements with negative indices.

The `seq()` function is an easy way to create a vector of evenly spaced numbers. It requires a starting point, an ending point, and either the final number of elements, or the spacing between elements.

```
> seq(0,1,by=0.2)

[1] 0.0 0.2 0.4 0.6 0.8 1.0

> seq(0,1,length.out=3)

[1] 0.0 0.5 1.0

> seq(0,10,by=3.1)

[1] 0.0 3.1 6.2 9.3
```

There are many functions that act on vectors, including: `diff` (differences between adjacent elements); `rev` (reverses a vector); `length` (number of elements in a vector); `sort` (sorts the elements of a vector); `mean` (averages the elements of a vector); and `sum` (adds the elements of a vector).

```
> y <- c(5,7,18:20)
> y

[1]  5  7 18 19 20

> z <- (4:8)^2
> z

[1] 16 25 36 49 64

> diff(z)

[1]  9 11 13 15

> y

[1]  5  7 18 19 20

> rev(y)
```

```
[1] 20 19 18  7  5
```

```
> w = c(3,2,4,1,5)
> w
```

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

```
> length(w)
```

```
[1] 5
```

```
> sort(w)
```

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

```
> mean(w)
```

```
[1] 3
```

```
> sum(w)
```

```
[1] 15
```

An important, and time saving, vector function is `rep()`. The `rep()` function allows one to replicate each element in a vector a certain number of times (given by the `each` argument), and replicate the whole vector a certain number of times (given by the `times` argument).

```
> rep(c(0,1), each=2, times=3)
 [1] 0 0 1 1 0 0 1 1 0 0 1 1
```

```
>  rep(c(0,1), each=3, times=2)
 [1] 0 0 0 1 1 1 0 0 0 1 1 1
```

Vectors of the same length can be combined into a matrix via the `rbind` (row bind) and `cbind` (column bind) commands.

```
> a <- 1:3
> b <- 6:4
```

```
> A <- rbind(a,b)
> B <- cbind(a,b,a)
```

```
> A
```

```
  [,1] [,2] [,3]
a    1    2    3
b    6    5    4
```

```
> B
```

```
     a b a
[1,] 1 6 1
[2,] 2 5 2
[3,] 3 4 3
```

Both `A` and `B` are of the class `matrix`.

```
> class(A)
```

```
[1] "matrix"
```

```
> class(B)
```

```
[1] "matrix"
```

To access elements of a matrix we use square brackets, as with a vector, but supply two arguments: the first referring to the rows, and the second referring to the columns.

To access the first of row of the matrix `B`, we use `B[1,]`; to access the first column of `B`, we use `B[,1]`.

```
> B[1,]
```

```
a b a
1 6 1
```

```
> B[,1]
```

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

We can also subset the rows and columns at the same time.

```
> A[,c(1,3)]

  [,1] [,2]
a    1    3
b    6    4
```

```
> B[c(1,3),c(2,3)]

     b a
[1,] 6 1
[2,] 4 3
```

To reshape a long vector into matrix we use the `matrix()` command. We must be careful to specify whether the matrix should be formed by row or by column.

```
> x = 1:12
> Mrow <- matrix(x, byrow=TRUE, ncol=4)
> Mcol <- matrix(x, byrow=FALSE, ncol=4)
```

```
> Mrow

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

```
> Mcol

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

3.2 Data frames

A data frame is an `R` object that contains vectors; the vectors are stored vertically in a matrix like structure, and can be referred to by the name of the column. The main advantage of a data frame is that the variables in a data frame do not all need to be the same type; e.g. some variables can be of class `numeric`, and some variables can be of class `character`.

We can create a data frame object using the `data.frame()` function. This data frame contains two small vectors, the first of which is named `X`, and the second `Y`.

```
> x = 1:4
> y = 5:8
> Data <- data.frame(X=x, Y=y)
> Data

  X Y
1 1 5
2 2 6
3 3 7
4 4 8

> class(Data)

[1] "data.frame"

> names(Data)

[1] "X" "Y"
```

We can access the original vectors in the following way:

```
> Data$X

[1] 1 2 3 4
```

```
> Data$Y

[1] 5 6 7 8
```

If we wish to change the variables names we can do so by assigning an argument to the `names()` function.

```
> names(Data) <- c("Var1", "Var2")
```

```
> Data$Var1
[1] 1 2 3 4
```

```
> Data$Var2
[1] 5 6 7 8
```

We can access the variables directly if we attach the data frame. To do this, we use the `attach()` command.

```
> attach(Data)
```

```
> Var1
[1] 1 2 3 4
```

```
> Var2
[1] 5 6 7 8
```

To detach the data frame we use the `detach()` command.

```
> detach(Data)
```

```
> Data$Var1
[1] 1 2 3 4
```

```
> Var1
Error: object "Var1" not found
```

There are times when it is useful to convert a data frame into a matrix; we can do this with the `as.matrix` function.

```
> M <- as.matrix(Data)
> M
     Var1 Var2
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
> class(M)
[1] "matrix"
```

4 Reading data files into R

Most data sets we shall consider in this course are in *tabular* form: this means that each variable is a column, each row is an observation, columns are separated by white space, and each column or row may have a name. If a data set of this form is stored in a plain ASCII text file, then we can read this file into R using the `read.table()` command, and store it in a data frame.

As an example, consider the `pop.txt` data set, available from the 161.220 course website `http://www.massey.ac.nz/~a161220`.

```
> pop <- read.table("http://www.massey.ac.nz/~a161220/pop.txt",
                    header=TRUE)
```

```
> pop
```

```
     agegroup nzmale90 nzfemale90 nzmale00 nzfemale00 usmale90 usfemale90 usmale00 usfemale00
1   preschool     1410       1347     1450       1383     9646       9204     9639       9227
2      school     4102       3928     4263       4066    27197      25857    30555      29043
3     working     8096       8108     9732       9655    62702      63009    68664      69599
4 nearretired     2020       2079     2204       2277    14464      16705    15778      17660
5      senior      994       1512     1311       1856     8053      13111    10139      15261
```

We have used the argument `header=TRUE` to indicate that the first row of the data file contains the variable names, and we have stored the data file in the data frame `pop`.

This data set is interesting for the fact that the first column is actually a description of the observation, in other words, a row name. If the data set is of this form, we would usually want the first column to be the row name, not just another variable. To do this we use the `row.names=1` option, which indicates that first column contains the row names.

```
> pop <- read.table("http://www.massey.ac.nz/~a161220/pop.txt",
                    row.names=1, header=TRUE)
```

```
> pop
```

```
            nzmale90 nzfemale90 nzmale00 nzfemale00 usmale90 usfemale90 usmale00 usfemale00
preschool       1410       1347     1450       1383     9646       9204     9639       9227
school          4102       3928     4263       4066    27197      25857    30555      29043
working         8096       8108     9732       9655    62702      63009    68664      69599
nearretired     2020       2079     2204       2277    14464      16705    15778      17660
senior           994       1512     1311       1856     8053      13111    10139      15261
```

```
> row.names(pop)
```

```
[1] "preschool"   "school"      "working"     "nearretired" "senior"
```

The `row.names()` function reveals the row names of the data frame. We can also access data in a data frame by using the matrix subscripts.

```
> pop$nzmale90
```

```
[1] 1410 4102 8096 2020  994
```

The `attributes()` command is a useful function that allows us to summarise the attributes of a given data set.

```
> attributes(pop)
```

```
$names
[1] "nzmale90"   "nzfemale90" "nzmale00"   "nzfemale00" "usmale90"
[6] "usfemale90" "usmale00"   "usfemale00"

$class
[1] "data.frame"

$row.names
[1] "preschool"   "school"      "working"     "nearretired" "senior"
```

4.1 R data sets

R has many data sets built in; one can access them through the `data()` command. For instance, to load the `anscombe` data set we type:

```
> data(anscombe)
```

A description of this data set is available via the help command:

```
> ?anscombe
```

Some data sets are stored in library that are not loaded by default. To access these data sets, one must supply the library name via the `package` option:

```
> data(singer, package="lattice")
```

5 Output

Apart form cutting and pasting from the R window there are a couple of ways of writing R output directly to a file

5.1 sink()

To send output straight to a file we can use the `sink()` command. Below, we fit a simple linear regression model (using the `lm()` function) for the first set of data in the `anscombe` data set; all the output is sent to the file `Rout.txt` which will be located in the working directory. The use of `sink()` ends the sink commands and closes the output file.

```
> sink("Rout.txt")
> data(anscombe)
> summary(lm (y1 ~ x1, data=anscombe))
> sink()
```

The resulting output file (`Rout.txt`) looks like:

```
Call:
lm(formula = y1 ~ x1, data = anscombe)

Residuals:
     Min       1Q   Median       3Q      Max
-1.92127 -0.45577 -0.04136  0.70941  1.83882

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   3.0001     1.1247   2.667  0.02573 *
x1            0.5001     0.1179   4.241  0.00217 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.237 on 9 degrees of freedom
Multiple R-squared:  0.6665,        Adjusted R-squared:  0.6295
F-statistic: 17.99 on 1 and 9 DF,  p-value: 0.00217
```

5.2 write.table()

To write a matrix to a file we use the `write()` command. The

```
> x <- matrix(1:10,ncol=5)
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
> write.table(x, file="Mat.txt")
```

The resulting file contains the following output:

```
"V1" "V2" "V3" "V4" "V5"
"1" 1 3 5 7 9
"2" 2 4 6 8 10
```

This file can be read in back into `R` with the `read.table()` command.

```
> x = read.table("Mat.txt")
```

6 Creating graphs

There are many different ways to plot data. In this section, we will create a simple plot, and try to cover the most common techniques for changing, labelling, and annotating the plot.

First we will make a scatterplot of Anscombe's first data set, shown in Figure 1.

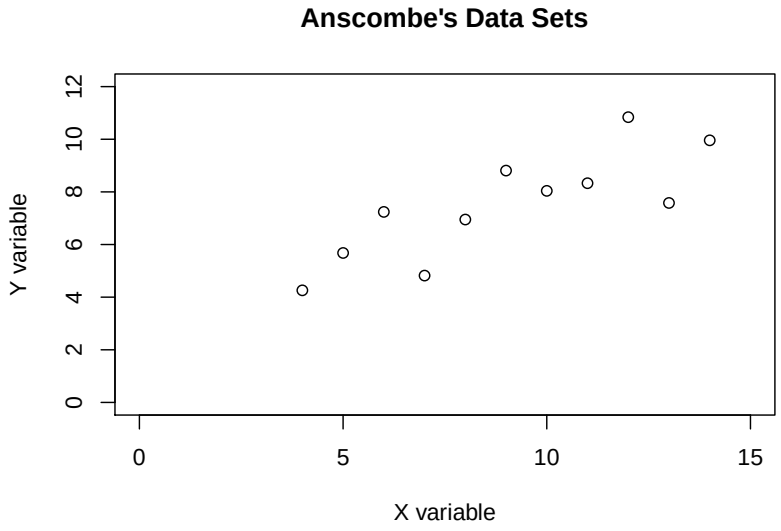

Figure 1: Basic scatterplot of Anscombe's first data set.

```
> data (anscombe)
> attach (anscombe)
> plot(x1, y1, xlim=c(0,15), ylim=c(0,12),
       xlab="X variable", ylab="Y variable",
       main="Anscombe's Data Sets")
```

We have used several options in creating this plot: `xlim=c(0,15)` specifies that the *horizontal* axis should be plotted from 0 to 15, regardless of the data; similarly `ylim=c(0,12)` specifies that the *vertical* axis should be plotted from 0 to 12, regardless of the data; the labels on the x and y-axis are set by `xlab` and `ylab` respectively; finally the main title is set by the `main` argument.

(Alternatively instead of producing the plot with `plot(x1, y1)` we could have used the `plot(y1 ~ x1)` form of the `plot()` command.)

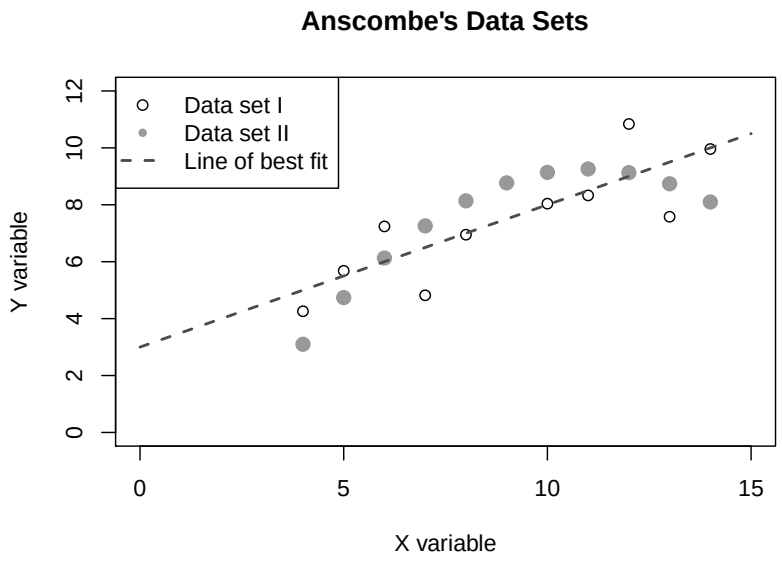

Figure 2: Scatterplot of Anscombe's first and second data sets; with line of best fit and legend added.

If we want to overlay one plot upon another, we can do so by using the `points()` command instead of `plot()` for the second plot.

```
> points(x2, y2, pch=20, col=2, cex=2)
```

This code, adds the second Anscombe data set to the plot of the first; the points are plotted with plotting character 20, which is a solid circle; the colour or the points is set to 2 (which is red); and the character expansion parameter is set to 2, which mean the plotting symbols will be twice as large.

The resulting figure is shown in Figure 2. We have also added a line of best fit, and a legend. As we saw from the linear regression output in the previous section, the regression line has a slope of 0.5 and an intercept of 3. We could have added this line with the command `abline(3,0.5)`; however, we have done it in the following, more descriptive, manner.

```
> x0 <- 0:15
> y0 <- 0.5 * x0 + 3
> lines(x0,y0, lty=2, col=4, lwd=2)
```

This code calculates points that lie on the regression line, and then plots them using the `lines()` function; this is similar to `points()` in that is adds points to an existing plot, but it differs by plotting the lines that go through the points instead. In this case we have also set some graphics

parameters for how the lines are drawn: the line type is set to be dashed instead of solid (`lty=2`); the colour of the line is set to four, which is blue (`col=4`); and the line width is set to 2 (`lwd=2`).

To complete the figure we add a legend which describes each part of the plot.

```
> legend("topleft",
         c("Data set I", "Data set II", "Line of best fit"),
         pch=c(1,20,NA), lty=c(NA,NA,2), col=c(1,2,4),
         lwd=c(NA,NA,2))
```

This command adds a legend in the top left corner of the plot. The next argument is a vector containing three character strings, each of which is a separate line in the legend. The remaining arguments in the command specify the type of symbol to be placed next to the text. Note the use of the `NA` constant; this denotes missing data. In this case the missing data is the line information for the first two elements of the legend, and the plotting character information for the last element of the legend.

6.1 Reviewing plots

One can scroll through created plots by selecting the `History` → `Recording` menu from the plotting window, then use `Page Up` and `Page Down`.

6.2 Multiple plots

To plot more than one plot in the same figure, use `par()` (graphics parameters) and set the `mfrow` (multiple figures by row) option.

```
> par(mfrow=c(2,3))
> plot(x,y, main="Plot 1")
> plot(x,y, main="Plot 2")
> plot(x,y, main="Plot 3")
> plot(x,y, main="Plot 4")
> plot(x,y, main="Plot 5")
> plot(x,y, main="Plot 6")
> par(mfrow=c(1,1))
```

In this example, we have created a plotting matrix consisting of two rows and three columns; the plotting windows will be filled in by row.

To turn it off the effect of the multiple plotting window, one can simply close the plotting window, or use `par()` to reset the plotting matrix to be a single (one row by one column) figure:

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

To fill the plotting windows by column, we can use the `mfcol` option:

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

6.3 Graphics parameters

Although we shall not cover them here, many plotting options can be set using `par()` function; including size of margins, font types, the colour of axis labels etc.

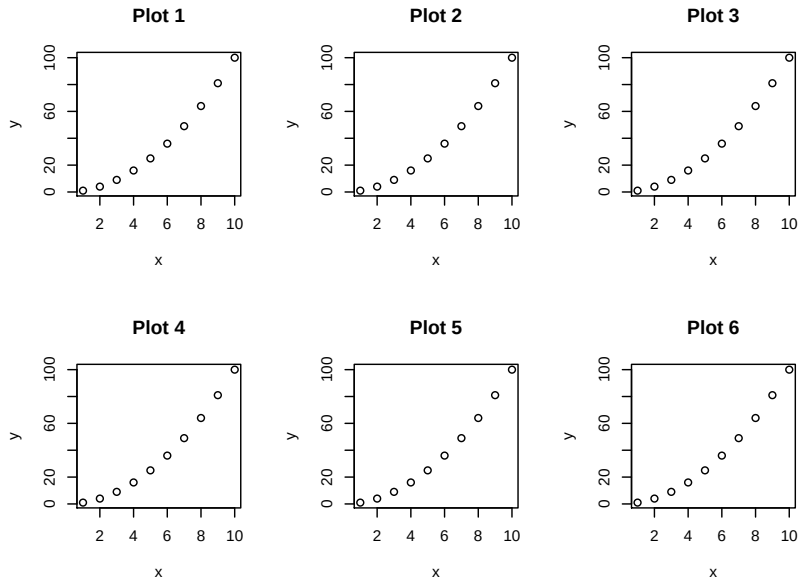

Figure 3: Multiple plots made in the same window using `par(mfrow)`.

```
> help("par")
```

6.4 Saving pictures

Plots can be saved in various file formats, such as PDF (`.pdf`), JPEG (`.jpeg` or `.jpg`), PNG(`.png`) or postscript (`.ps`) by using the `File` → `Save as` menu.

One can also enclose the plotting function in the appropriate commands. For example, to save a simple figure as a PDF file, we use the `pdf()` function.

```
> x = 1:10
> y = x^2
> pdf(file="Fig.pdf")
> plot(x,y)
> dev.off()
```

The command `dev.off()` closes the file. To have saved the graphic as a JPEG file we use the `jpeg` command.

```
> x = 1:10
> y = x^2
> jpeg(file="Fig.jpg")
> plot(x,y)
> dev.off()
```

Similarly, the function `png()` creates PNG files, and `postscript()` create postscript files.

The PDF format is probably the most portable graphic format, with the highest quality output.

7 Generating random numbers

To generate a random number between zero and one in R, we use the `runif()` function. (The `set.seed()` function sets the random number seed allowing us to generate the same random numbers again.)

```
> set.seed(123)
> runif(n=5)
```

```
[1] 0.2875775 0.7883051 0.4089769 0.8830174 0.9404673
```

We can supply different parameters for the end-points of the uniform distribution.

```
> runif(n=5,min=4, max=8)
```

```
[1] 4.182226 6.112422 7.569676 6.205740 5.826459
```

Most other common statistical distributions are available in R as pre-defined functions, namely:

Beta	`rbeta()`	Log-normal	`rlnorm()`
Binomial	`rbinom()`	Logistic	`rlogis()`
Cauchy	`rcauchy()`	Multinomial	`rmulitnom()`
χ^2	`rchisq()`	Negative-Binomial	`rnbinom()`
F	`rf()`	Normal	`rnorm()`
Gamma	`rgamma()`	Poisson	`rpois()`
Geometric	`rgeom()`	Student's-t	`rt()`
Hypergeometric	`rhyper()`	Weibull	`rweibull()`

Note that the first parameter of all these functions is n, the number of realisations you require; the following parameters are distribution specific.

For every function `rdist()` there are three other functions, `ddist()`, `qdist()`, `pdist()` (e.g. for the normal distribution these functions are named `dnorm()`, `qnorm()`, `pnorm()`.)

The `ddist(x,...)` function return the height of the density curve at point x (i.e. $f(x)$). The `pdist(x,...)` function returns the value of the cumulative distribution function (i.e. $F(x)$) at point x. The `qdist(p,...)` function returns the value of quantile for a particular probability p.

8 Using RStudio

When we write R code it is a good idea to type it into a file (with a `.R` extension), and then send the code to R, rather than the other way around. We can use any simple text editor for this, e.g. Notepad, Wordpad, Emacs.

A more sophisticed enviroment is the RStudio program. RStudio is a is a free and open source integrated development environment (IDE) for R. RStudio contains a text editor, R console, file manager, plotting window, and help system.

After starting RStudio, `R` starts automatically. The `File` → `New` menu can be used to open a file for editing `R` code. This code can be sent to `R` by highlighting the code and pressing the `Run` button.

The working directory can be set with the `Session` → `Set Working Directory` → `Choose Directory...` menu.

Remember to save your code (`File` → `Save` before quitting.