Getting Started in R

S.A. Bashir

February 26, 2003

Contents

Preface

This "Getting Started in R" is aimed at absolute beginners in R with some basic statistical understanding. The idea is to read and follow the examples by typing them in an interactive R session. There are some exercises along the way to aid learning but I would recommend that the user experiment with commands by, for example, trying different options to those stated. This experimentation is an important part of learning R using this manual.

This guide does not aim to give detailed technical description of the workings of R or the logic behind it. There is plenty of other literature that already does that. It is hands-on approach to learning R.

If you would like some further reading or information, the R website has an up to date list of references. The website address is:

`www.r-project.org`

If you have any comments please let me know.

Saghir Bashir (`saghir@sbtc.ltd.uk`)

The latest version of this document can be found on:

`www.sbtc.ltd.uk/freenotes.html`

Font Conventions

These course notes use the following typographical conventions:

> `Constant width` to show the text to be typed, output from commands or the contents of files.

Acknowledgments

These notes started life as an introduction to Splus and were at the time based on, with permission, some notes by Barry Rowlingson. I am grateful to Barry for his permission.

I would like to acknowledge the R Development Core Team for their hard work in producing R.

1 Introduction

1.1 What is R?

R is an programming environment for data analysis, calculation and graphics. In summary it's main features are

- data handling and storage facility

- operators for matrix (and array) manipulation

- data analysis tools

- graphical facilities

- a programming language

1.2 R and Splus

For the purpose of this manual R and Splus should work in an identical way.

1.3 Starting and quitting R

To start R in the the Windows environment double click on the R icon. A new screen will appear containing one window (which lists information about the version number, license and getting started). The last line to appear will be '>', a standard prompt to indicate that R is expecting a command.

```
R : Copyright 2001, The R Development Core Team
Version 1.3.0  (2001-06-22)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for a HTML browser interface to help.
Type 'q()' to quit R.

>
```

To quit R, type q() followed by enter at the prompt:

```
> q()
```

A prompt asking whether to `Save workspace` will appear. Answering `Yes` will save all *objects* (variables) that were created during the session. Next time R is started from the same directory, the saved objects will be available for use. For the purposes of this course it is best not to save workspace to prevent confusing objects between sessions.

1.4 On-line Help

The on-line help gives useful information. Getting used to using it and understanding the help will make it easier to use R. The on-line help can be accessed in HTML format by typing:

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

A keyword search is possible using the *Search Engine and Keywords* link.

There is text help from within R using the function `help()` or `?`. For example, the following two commands result in the same thing.

```
> help(t.test)
> ?t.test
```

To do a keyword search use the function `apropos()`. For example:

```
> apropos("test")
[1] "chisq.test"  "print.htest" "prop.test"   "t.test"
```

Note that you need to put the keyword in double quotes (``keyword'').

1.4.1 Help Examples

To run the examples at the end of the help for a function, use the `example()` function. For example:

```
> example(t.test)

t.test> t.test(1:10, y = c(7:20))

        Welch Two Sample t-test

data:  1:10 and c(7:20)
t = -5.4349, df = 21.982, p-value = 1.855e-05
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -11.052802  -4.947198
sample estimates:
mean of x mean of y
      5.5       13.5
```

```
t.test> t.test(1:10, y = c(7:20, 200))

        Welch Two Sample t-test

data:  1:10 and c(7:20, 200)
t = -1.6329, df = 14.165, p-value = 0.1245
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -47.242900   6.376233
sample estimates:
mean of x mean of y
  5.50000  25.93333
```

1.4.2   Exercise

Use the help system to find help on the following topics:

1. Analysis of Variance (ANVOA)

2. Trigonometric Functions (e.g., `cos`)

3. Mean and median

4. Generalised Linear Models

1.5   General comments

1.5.1   Case Sensitivity

R is case sensitive. `D` and `d` are different symbols and refer to different variables. Similary `sag`, `Sag` and `SAG` refer to three different variables.

1.5.2   R Commands

R commands are separated by a semi-colon (';') or by a newline. To put comments in your code use a hash ('`#`') and everything from the hash to the end of the line will be regarded as a comment.

If a command is not complete at the end of a line then R will issue the following prompt (by default):

```
+
```

on second and subsequent lines until the command syntax is correct. To break out this, type `CTRL + c` (press the Control Key and 'C' at the same time).

1.5.3   Recalling previous commands

To recall a previously typed commands use the up arrow key (↑). To go between previously typed commands use the up and down arrow (↓) keys. Once a command is recalled, it can be modified/corrected using the left (←) and right arrow keys (→).

2   Maths in R

In this section R will be used to do simple arithematic.

2.1   Simple arithmetic

Whatever is typed at the prompt is evaluated, and the result is printed.

```
> 2+3            # Addition
[1] 5            # Answer labelled with [1]

> 2*4 + 7        # Multiplication and addition
[1] 15           # Multiplication first

> 10/3           # Division
[1] 3.333333

> 7^2            # Squaring ...
[1] 49

> 5**3           # ** to the power of...
[1] 125
> 5^3            # or we can use ^
[1] 125

> (56-12)/4 - 7*(84/12-3) # more complicated...
[1] -17
```

Standard functions that are found on a scientific calculator are available in R, for example:

```
> sqrt(2)        # Square root
[1] 1.414214

> sin(3.14159)    # sin(Pi radians) is zero
[1] 2.65359e-06   # and this is close...
```

It also provides `pi` (π) as a constant.

```
> sin(pi)
[1] 1.224606e-16  # much closer to zero...
```

Here is a short list of some of the arithmetic functions in Splus:

Name			**Operation**
`sqrt`			square root
`abs`			absolute value
`sin`	`cos`	`tan`	trigonometric functions (radians)
`asin`	`acos`	`atan`	inverse trigonometric functions
`sinh`	`cosh`	`tanh`	hyperbolic functions
`asinh`	`acosh`	`atanh`	inverse hyperbolic functions
`exp`	`log`		exponential and natural logarithm
`log10`			common logarithm
`gamma`	`lgamma`		gamma function and its natural log

These functions can be nested and combined to make more complex expressions:

```
> sqrt(sin(45*pi/180))
[1] 0.8408964
```

2.1.1 Exercise

Calculate the following in Splus

1. $(12^3 - 45)/4 + 4 \times (72/2.34 - 3)$

2. $log_e(72)$

3. $log_{10}(72)$

4. $e^{1.45} - 2.612$

5. $\cos(\frac{\pi}{3})$

2.2 Storing values

A value can be stored in a named variable by assigning it with the `<-` or `_` (underscore) symbols, for example:

```
> x <- 5        # assigns 5 to x
> x             # type x to print it's value
[1] 5

> y _ sqrt(9)   # assign the square-root of 9 to y
> y
[1] 3
```

In fact you can make assignments using `->`, for example:

```
> 7 -> z      # assigns 7 to z
> z
[1] 7
```

This could be interpreted as z is assigned 7.

Now it is possible to do arithmetic with x, y and z, for example:

```
> (x * y) + z
[1] 22

> y^x - z + 6
[1] 242

> w <- x+y
> w
[1] 8
```

Notice that if you type an assignment it does not print anything, but if you just type an expression the result is printed.

Variable names must start with a letter, and may contain letters, numbers and dots. Upper and lower case are different.

```
> abc <- 123      #
> Abc <- 456      # abc, Abc and ABC are different.
> ABC <- 789      #

> abc
[1] 123
> Abc
[2] 456
> ABC
[1] 789

> abc.de2 <- 543  # dot notation and numbers in a variable name.
> abc.de2
[1] 543

4abc <- 98        # variable names cannot start with a number.
Error: syntax error
```

2.3 Logical Values

R enables computation with Boolean, or Logical variables. These take on either the value True or False. You can use conditional tests to generate these values:

```
> x <- 32
```

```
> x > 16        # Is x greater than 16?
[1] T           # Yes it is (True).
> x <= 16       # Is x less than equal to 16?
[1] F           # No it is not (False).
```

Logical values can be stored in variables in the same way as numeric values:

```
> tf <- x>16
> tf
[1] T
```

2.4   Managing Objects

To list the objects (e.g., variables, data, functions) that you have created simply type `ls()`. For example,

```
> height <- 1.78
> weight <- 83
> age <- 26
> job <- "Builder"
> ls()
[1] "age"    "height" "job"    "weight"
```

To search for objects which contain given characters, use the `pattern` option (abbreviated to `pat`).

```
> ls(pat="h")
[1] "height" "weight"
```

To restrict the search to objects that start with this character, type:

```
> ls(pat="^h")
[1] "height"
```

To delete an object use the `rm()` functions.

```
> rm(age)
> ls()
[1] "height" "job"    "weight"
```

3   Vectors

All the values presented so far have been scalars. R can handle vectors which are a combination of scalars in a single structure.

3.1 Creating Simple Vectors

To create a simple vector, use the `c()` (combine function) function. Try

```
> x <- c(2,4,6,10,11)    A sequence of numbers.
> x                      To see the outcome...
[1]  2  4  6 10 11
```

3.2 Sequences

You can use the notation `#1:#2` to generate a vector that consist of a sequence where `#1` and `#2` are two integer numbers.

```
> xv <- 1:10
> xv
 [1]  1  2  3  4  5  6  7  8  9 10
> yv <- 40:1
> yv
 [1] 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26
[16] 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11
[31] 10  9  8  7  6  5  4  3  2  1
```

The last example (`yv`) shows that is a vector is too long for one line it simply continues on the following line (using as many lines as are necessary). Further in the square brackets it shows the position at which that line begins (e.g., `[16]` mean that the first values on this line is the sixteenth one).

3.3 Generating Sequences

3.3.1 Regular Sequences

R can be used to generate vectors sequences. Simple regular sequences have already been presented above and are generally of the form:

```
> 1:12
 [1]  1  2  3  4  5  6  7  8  9 10 11 12
> 10:1
 [1] 10  9  8  7  6  5  4  3  2  1
> 4:18
 [1]  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
```

To generate a sequence of repeating sequences use the `rep()` functions, for example:

```
> r1 <- rep(c(1:4), 3)
> r1
 [1] 1 2 3 4 1 2 3 4 1 2 3 4
```

```
> r2 <- rep(c(1.2, 6.1, 3.2, 5.5), 2)
> r2
[1] 1.2 6.1 3.2 5.5 1.2 6.1 3.2 5.5
```

The first sequence repeat 1, 2, 3, 4 3 times. The second sequence repeats a list of numbers (1.2, 6.1, 3.2, 5.5) twice.

To generate a sequence of real numbers use the function seq(), for example:

```
> seq(4, 6, .25)  # from 4 to 6 by increments of 0.25.
[1] 4.00 4.25 4.50 4.75 5.00 5.25 5.50 5.75 6.00
```

The following examples produces the same sequence

```
> seq(length=9, from=4, to=6)
[1] 4.00 4.25 4.50 4.75 5.00 5.25 5.50 5.75 6.00
> seq(from=4, to=6, by=0.25)
[1] 4.00 4.25 4.50 4.75 5.00 5.25 5.50 5.75 6.00
```

To understand the seq() function type the following:

```
> seq(7)
[1] 1 2 3 4 5 6 7
> seq(7, 10)
[1]  7  8  9 10
> seq(7, 10, 0.5)
[1]  7.0  7.5  8.0  8.5  9.0  9.5 10.0
> ?seq

> rep(seq(1, 3, 0.4), 2)  # Combining rep() and seq()
 [1] 1.0 1.4 1.8 2.2 2.6 3.0 1.0 1.4 1.8 2.2 2.6 3.0
```

3.3.2   Random Sequences

R can generate a random sequence from a number of probability density functions. The general format for generating such sequences is: r*density*(num, p1, p2, ...) where *density* is the probability density function, num is the number of values to generate and p1, p2, ... are the values needed to generate from the density function.

To generate a 40 values from a Guassian (Normal) distribution with mean 6 and standard deviation 2.3, type

```
> normal40 _ rnorm(40, 6, 2.3)
> normal40
 [1]  6.834486  2.448428  9.872630  6.843854  3.523625  5.836164
 [7]  2.467408  4.031621  7.117638  4.505798  1.992775  2.245854
[13]  6.500079 10.027989 11.256062  6.790967  7.470746  6.410076
```

```
[19]  6.319011  9.647313  2.671094  7.659652  7.523609  4.191818
[25]  3.786023  4.156107  5.677170  3.828295  5.112820  7.847509
[31]  5.939255  7.163945  4.771867  7.406695  3.807607  3.809296
[37]  3.097054  7.163667  7.019444  8.740871
```

The vector `normal40` can be used as data in R, for example,

```
> mean(normal40)
[1] 5.837908
> sd(normal40)
[1] 2.343080
> summary(normal40)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.993   3.824   6.129   5.838   7.225  11.260
```

Some examples of generating random sequences from other distributions are presented below:

```
> rpois(20, lambda=5)            # Poisson(5)
 [1] 8 6 4 3 3 5 4 3 2 9 0 4 8 5 2 5 8 5 1 6


> rbinom(20, size=18, prob=0.3)  # Binomial(18, 0.3)
 [1] 4 4 7 1 6 5 4 5 5 4 5 8 4 7 7 5 2 8 4 5


> runif(20, min=3, max=6)        # Uniform(3, 6)
 [1] 4.149455 3.575531 3.501311 3.815135 3.525723 4.858564
 [7] 3.827195 3.360773 5.460830 5.724241 3.817351 5.755735
[13] 4.232872 4.367516 5.110012 3.960176 3.340431 4.707575
[19] 5.616004 4.341320


> rexp(20, rate=1.5)             # Exponential(1.5)
 [1] 0.61952785 0.92581617 0.85785772 0.53322649 0.37195255
 [6] 0.84596779 0.38540842 0.27079439 0.05713503 0.92850265
[11] 0.23248135 0.46429596 0.12284165 0.61804739 0.14611095
[16] 1.13248605 1.15375220 0.73407235 0.04107131 0.54812605
```

The function `sample()` generates random permutations of data or a random sample from a given vector. The first arguement is a vector or a positive integer and the second the size of the sample. For example:

```
> sample(12)       # Sampling without replacement.
 [1]  9  5  6  3  7  1  2  8 10 11  4 12


> sample(12, 5)    # Sampling 5 values without replacement.
[1] 10 12  6  3  5


> sample(12, replace=TRUE)      # Sampling with replacement.
 [1] 11  8  6 12 10  4  5  8  4  5  1  1
```

```
> sample(12, 8, replace=TRUE)
+        # Sampling 8 values with replacement.
[1]  6  3  7  9  5  8 10  7


> sample(c("Heads", "Tails"), 6, replace=TRUE)
+        # Simulating 6 coin tosses.
[1] "Heads" "Heads" "Tails" "Heads" "Tails" "Heads"
```

3.3.3   Exercises

1. Generate the following sequences

   ```
   [1]  8.0  8.2  8.4  8.6  8.8  9.0  9.2  9.4  9.6  9.8 10.0
   [1] 10.0  9.8  9.6  9.4  9.2  9.0  8.8  8.6  8.4  8.2  8.0
   ```

2. Generate 80 values from a standard normal distribution.

3. Generate 100 values from a binomial distribution of size 23 and probability 0.25.

4. Generate 100 values from a log normal distribution with mean 12 and standard deviation 2.

3.4   Vector Arithmetic

We can manipulate vectors in a similar manner to scalars. However care must be take when doing such things as the results may not be the desired ones.

```
> x <- 12:1        # x is a sequence .....
> x
 [1] 12 11 10  9  8  7  6  5  4  3  2  1
> x*2              # multiply x by 2 ...
 [1] 24 22 20 18 16 14 12 10  8  6  4  2
> x*x              # square x....
 [1] 144 121 100  81  64  49  36  25  16   9   4   1
```

If the vectors are of different length we can get some strange answers for example try the following:

```
> x <- 1:10
> y <- c(1,3)
> x*y
 [1]  1  6  3 12  5 18  7 24  9 30
```

Here Splus has repeated y five times and then multiplied by x

x	1	2	3	4	5	6	7	8	9	10
y	1	3	1	3	1	3	1	3	1	3
x × y	1	6	3	12	5	18	7	24	9	30

In this example the `length` of `y` is a factor of the `length` of `x` (type `length(x)` and `length(y)`). However if it is not a factor then a warning message is issued as follows:

```
> x <- 1:10
> y <- c(1,2,3)
> x + y
Warning in x + y : longer object length
        is not a multiple of shorter object length
 [1]  2  4  6  5  7  9  8 10 12 11
```

Note that the operation has been completed (see the line after `x + y`. Can you see what has happened?

Logical operations on a vector produce a vector of True and False values, for example

```
> x > 5
 [1] F F F F F T T T T T
```

3.5 Extracting Elements of a Vector

If we are interested in only extracting a subset of the vector then we can do this using square brackets `[ ]`.

```
> x <- c(1:10)*2
> x
 [1]  2  4  6  8 10 12 14 16 18 20
> x[6]           # extracting the 6th value...
[1] 12           # which is 12

> x[2:6]         # extracting values 2 through 6 inclusive.
[1]  4  6  8 10 12

> x[c(1,7,9)]    # extracting the 1st, 7th and 9th values.
[1]  2 14 18

> y <- x[c(1,5,8)]  # assigning a subset of x to y.
> y
[1]  2 10 16
```

Here are some examples to try

```
> x[9:6]             #  reverse order...
[1] 18 16 14 12

> x[c(1:3, 8:10)]    #  two distinct ranges...
[1]  2  4  6 16 18 20
```

```
> x[c(1,2,3,1,2,3,1,2,3,1,2,3)]   # repetition of the index...
 [1] 2 4 6 2 4 6 2 4 6 2 4 6

> x[c(8,2,5,10)]     # any order you please....
[1] 16  4 10 20
```

As mentioned above you can assign subsets to a vector

```
> y <- x[c(1:4,3,9)]
> y
[1]  2  4  6  8  6 18
```

3.6 Negative Numbers

If you use a negative subscript in you selection procedure, the corresponding numbered
element is not included in the return vector.

```
> x <- c(3,6,9,12,15,18,21)
> x
[1]  3  6  9 12 15 18 21

> x[-4]           # exclude the 4th element...
[1]  3  6  9 15 18 21

> x[c(-4,-6)]     # exclude 4th and 6th elements...
[1]  3  6  9 15 21
```

If you try to mix negative and positive numbers then you get the following error

```
> x[c(3,-4)]
Error: only 0's may mix with negative subscripts
```

3.7 Logical Subscripts

Logical values can be used as subscripts. A True value selects that element, and a
False value does not select it:

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

> gt5 <- x > 5
> gt5
 [1]  F  F  F  F  F  T  T  T  T  T

> x[gt5]
[1]  6  7  8  9 10
```

This can all be done in one line, however:

```
> x[x > 5]
[1]  6  7  8  9 10
```

This looks a bit strange to most people at first, seeing x inside a subscript of itself, but when you remember that x > 5 is just a vector of True and False, it starts to make sense.

3.8   More Functions

There are functions that operate on vectors and return useful information:

```
> x <- 5:14

> length(x)       # Number of elements in x
[1] 10

> max(x)          # Largest value in x
[1] 14

> min(x)          # Smallest value in x
[1] 5

> sum(x)          # Sum of all the values in x
[1] 95

> prod(x)         # The product of all the values in x
[1] 3632428800

> mean(x)         # The mean of the all values in x
[1] 14.5

> range(x)        # Range of vector x
[1]   5 14

> var(x)          # The variance of x
[1] 35

> sd(x)           # The standard deviation of x
[1] 3.027650

> sqrt(var(x))    # The square root of the variance (sd)
[1] 3.027650
```

Notice that `range(x)` produces a vector of length two, whereas the other functions produce a scalar. Arithmetic operations on logical values work with True being equal

to one, and False being equal to zero. You can count the number of True values in a vector by using `sum(x)`

```
> x <- 1:10
> x>7
 [1] F F F F F F F T T T
> sum(x>7)
[1] 3
```

This tells us there are three values in the vector x that are greater than 7.

The following functions action on the whole vector.

```
> y <- c(-3.72, 11.56, 14.57, 19.65, -4.41, 15.41,
+ 25.79, 6.21, 9.84, 12.92)

> round(y, 1)       # Round y to one decimal place.
 [1] -3.7 11.6 14.6 19.6 -4.4 15.4 25.8  6.2  9.8 12.9

> trunc(y)          # Take the integer part of y.
 [1] -3 11 14 19 -4 15 25  6  9 12

> ceiling(y)        # Round up to the nearest integer.
 [1] -3 12 15 20 -4 16 26  7 10 13

> rev(y)            # Reverse the order of y.
 [1] 12.92  9.84  6.21 25.79 15.41 -4.41 19.65 14.57 11.56 -3.72

> sort(y)           # Sort y in increasing order.
 [1] -4.41 -3.72  6.21  9.84 11.56 12.92 14.57 15.41 19.65 25.79

> rev(sort(y))      # Sort y in decreasing order.
 [1] 25.79 19.65 15.41 14.57 12.92 11.56  9.84  6.21 -3.72 -4.41

> rank(y)           # Rank elements of y.
 [1]  2  5  7  9  1  8 10  3  4  6

> cumsum(y)         # Calculate the cumulative sum of y.
 [1]  -3.72   7.84  22.41  42.06  37.65  53.06  78.85
 [8]  85.06  94.90 107.82

> cumprod(y)         # Calculate the cumulative product of y.
 [1] -3.720000e+00 -4.300320e+01 -6.265566e+02 -1.231184e+04
 [5] 5.429520e+04  8.366891e+05  2.157821e+07  1.340007e+08
 [9] 1.318567e+09  1.703588e+10
```

3.9   Missing Values

Missing values are coded as `NA` in R. For example,

```
> missx <- c(2, 5, 7, NA, 4, 5, 2)

> missx
[1]  2  5  7 NA  4  5  2

> missx[2]
[1] 5

> missx[4]
[1] NA
```

The fourth element of `missx` is missing. Statistical functions will return a missing value (`NA`) if a vector contains any missing values. To overcome this we have to use the `na.rm=T` option.

```
> median(missx)
[1] NA
> median(missx, na.rm=T)
[1] 4.5

> min(missx)
[1] NA
> min(missx, na.rm=T)
[1] 2
```

Try the same for other statitical functions, e.g., `mean()`, `var()`, `prod()`, etc.

3.10   Exercise

1. Create the following vector sequences:

```
 1   4   7 10 13 16 19 22 25 28

1.0 1.2 1.4 1.6 1.8 2.0 1.0 1.2 1.4 1.6 1.8 2.0

10.0  9.5  9.0  8.5  8.0  7.5  7.0

1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4

4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1

 1   4   9  16  25  36  49  64  81 100

 1   4   9 16 25   6 14 24 36 50
```

2. For each of the sequences above calculate the mean, median, variance and the range.

3. Generate 15 values from a uniform distribution between 0.7 and 1.

4. Generate 15 values from a uniform distribution between 0 and 0.3.

5. Generate 28 values from a uniform distribution between 1.5 and 7.5.

6. Generate 15 values from a uniform distribution between 0 and 1. Sort these values in increasing order and then generate a vector with the cumulative product.

7. Using the functions `runif()` and `trunc()` simulate 20 die throws.

8. Generate 30 values from a Normal distribution with mean 3.4 and variance 5.25 rounded to 2 decimal places. Calculate the median, range, and the upper and lower quartiles.

9. Generate 30 values from a Poisson distribution with lambda 5.

10. Generate 20 values from a Binomial distribution of size 15 and probability of success 0.3.

11. Generate a randomisation list of size 150 for three treatment groups "Placebo", "Drug A" and "Drug B".

12. Generate 100 values from a standard normal distribution and count the number of values greater than 1.96. Extract these into a vector called signif.100. Do the same again but this time generate 2000 values.

4 Matrices

Splus lets you store data in a two-dimensional matrix. You use the matrix function:

```
> x <- matrix(c(2,3,5,7,11,13),ncol=2)
> x
     [,1] [,2]
[1,]    2    7
[2,]    3   11
[3,]    5   13
```

You give matrix a vector of the values, and you need to specify either `ncol` or `nrow` to tell the function the size of the matrix.

Can you see the labels on the matrix rows and columns? These are like the labels you see when printing a vector. They also tell us how to extract parts of a matrix. You use square brackets with two comma-separated values:

```
> x[2,1]    # One element of x
[1] 3
> x[2,2]    # Another element
[1] 11
```

If you leave out one number, you get the whole row or column:

```
> x[,1]      # The first column
[1] 2 3 5
> x[3,]      # The third row
[1]  5 13
```

Notice that returning a single row or column produces a vector. You can extract sub-matrices from a matrix by specifying a vector as one of the indices:

```
> x[2:3,]             Rows 2 and 3
     [,1] [,2]
[1,]    3   11
[2,]    5   13        labels start at 1 again
```

You can specify a vector for the rows and columns subscripts to get a piece of the original matrix:

```
> x <- matrix(1:16,ncol=4)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16
> x[c(1,4),c(3,4)]        Rows 1 and 4,
     [,1] [,2]            Cols 3 and 4
[1,]    9   13
[2,]   12   16
```

If you can understand that example, then you understand the concepts of matrix in Splus.

4.1 Matrix assignments and construction

By using subscripts you can change values in a matrix:

```
> mx <- matrix(seq(0, 95, length=20), ncol=5)

> mx                 #  before...
     [,1] [,2] [,3] [,4] [,5]
[1,]    0   20   40   60   80
[2,]    5   25   45   65   85
[3,]   10   30   50   70   90
[4,]   15   35   55   75   95
```

```
> mx [3,1] <- -12

> mx                     # ... after
> mx
     [,1] [,2] [,3] [,4] [,5]
[1,]    0   20   40   60   80
[2,]    5   25   45   65   85
[3,]  -12   30   50   70   90
[4,]   15   35   55   75   95
```

You can replace whole rows or columns too:

```
> mx[,2] <- c(3, 6, 9, 12)
> mx
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    3   40   60   80
[2,]    5    6   45   65   85
[3,]  -12    9   50   70   90
[4,]   15   12   55   75   95
> mx[4,] <- c(-1, -2, -3, -4, -5)
> mx
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    3   40   60   80
[2,]    5    6   45   65   85
[3,]  -12    9   50   70   90
[4,]   -1   -2   -3   -4   -5
```

As with vector arithmetic, the assigned value is repeated to fill out the matrix section if it is not long enough:

```
> mx[,1] <- c(0.5, 0.9)
> mx
     [,1] [,2] [,3] [,4] [,5]
[1,]  0.5    3   40   60   80
[2,]  0.9    6   45   65   85
[3,]  0.5    9   50   70   90
[4,]  0.9   -2   -3   -4   -5
```

Extra rows and columns can be added to a matrix using the `rbind()` and `cbind()` functions. These add extra rows and columns respectively:

```
> cbind(mx, c(0.1, 0.2, 0.3, 0.4))     # Add an extra column.
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]  0.5    3   40   60   80  0.1
[2,]  0.9    6   45   65   85  0.2
[3,]  0.5    9   50   70   90  0.3
[4,]  0.9   -2   -3   -4   -5  0.4
```

```
> rbind(mx, )
> rbind(mx, 100:104)                        # Add a an extra row.
      [,1] [,2] [,3] [,4] [,5]
[1,]   0.5    3   40   60   80
[2,]   0.9    6   45   65   85
[3,]   0.5    9   50   70   90
[4,]   0.9   -2   -3   -4   -5
[5,] 100.0  101  102  103  104


> cbind(mx,mx)     # Add the matrix to itself (column wise).
    [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]  0.5    3   40   60   80  0.5    3   40   60    80
[2,]  0.9    6   45   65   85  0.9    6   45   65    85
[3,]  0.5    9   50   70   90  0.5    9   50   70    90
[4,]  0.9   -2   -3   -4   -5  0.9   -2   -3   -4    -5


> rbind(mx, mx)     # Add the matrix to itslef (row wise).
      [,1] [,2] [,3] [,4] [,5]
[1,]  0.5    3   40   60   80
[2,]  0.9    6   45   65   85
[3,]  0.5    9   50   70   90
[4,]  0.9   -2   -3   -4   -5
[5,]  0.5    3   40   60   80
[6,]  0.9    6   45   65   85
[7,]  0.5    9   50   70   90
[8,]  0.9   -2   -3   -4   -5
```

You can add rows or columns to the middle of a matrix with clever use of subscripts
and `rbind()` or `cbind()`, respectively:

```
>  cbind(mx[,1:3], c(91,92,93,94), mx[,4])
      [,1] [,2] [,3] [,4] [,5]
[1,]  0.5    3   40   91   60
[2,]  0.9    6   45   92   65
[3,]  0.5    9   50   93   70
[4,]  0.9   -2   -3   94   -4
```

The function `t()` tranposes a matrix (switched the rows and columns).

```
> mx
      [,1] [,2] [,3] [,4] [,5]
[1,]  0.5    3   40   60   80
[2,]  0.9    6   45   65   85
[3,]  0.5    9   50   70   90
[4,]  0.9   -2   -3   -4   -5
> t(mx)
      [,1] [,2] [,3] [,4]
[1,]  0.5  0.9  0.5  0.9
```

```
[2,]  3.0  6.0  9.0 -2.0
[3,] 40.0 45.0 50.0 -3.0
[4,] 60.0 65.0 70.0 -4.0
[5,] 80.0 85.0 90.0 -5.0
```

4.2 Matrix Arithmetic

Functions on matrices work the same as on vectors, in general. They work on each
element in turn. This includes the multiplication operator `a * b` which multiplies each
element of a with the corresponding element of b. To do proper matrix multiplica-
tion, use `a %*% b`. The matrices **must** have the correct dimensions to be multiplied
together.

```
> my <- matrix(rep(1,4), ncol=2)
> my
     [,1] [,2]
[1,]    1    1
[2,]    1    1

> mz <- matrix(1:6, ncol=2)
> mz
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

> mz %*% my
     [,1] [,2]
[1,]    5    5
[2,]    7    7
[3,]    9    9
```

The size of a matrix is returned by the `dim()` function:

```
> dim(mx)
[1] 4 5        # number of rows then the number of columns.
```

The inverse of a matrix is computed by the `solve()` function. The matrix must be
square and not singular:

```
> x <- matrix(0:3,ncol=2)
> x                     # make a square matrix
     [,1] [,2]
[1,]    0    2
[2,]    1    3
> solve(x)             # whats the inverse?
     [,1] [,2]
```

```
[1,] -1.5    1
[2,]  0.5    0
> x %*% solve(x)        # test:  x * inv(x) = I
     [,1] [,2]
[1,]    1    0
[2,]    0    1          # 2x2 Identity
```

5 Data frames

A data frame is very much like a matrix, except it is designed for storing statistical or experimental data. Each row represents a unit, and each column a collection of measurements on the units. Each column can store a different type of data, such as numeric or character. The columns can also have names much like a list.

5.1 Reading data from files

So far, all data has been entered in R using the keyboard or generated using R functions. In reality, data is usually too large to type in and held in files. The function `read.table()` is used to read a file into a data frame. It will look at the file and determine if the data are numeric or character, and will also check for the presence of labels above the columns.

The file *teachers.dat* contains the following data:

```
age sex children salary weight height
47 Male    0 34377 94 1.72
48 Male    1 14502 79 1.86
40 Male    4 24899 88 1.49
61 Male    0 52168 79 1.80
41 Male    2 40344 88 1.84
35 Male    1 30512 94 1.68
57 Male    2 31845 85 1.70
54 Male    1 30447 90 1.54
51 Male    5 51862 88 1.71
43 Male    0 29632 90 1.84
50 Male    5 25663 89 1.82
53 Female 1 31956 83 1.57
53 Female 3 44864 66 1.45
55 Female 1 30301 64 1.47
45 Female 2 33190 72 1.54
57 Female 3 49778 88 1.38
50 Female 3 31546 75 1.47
46 Female 2 30157 65 1.60
44 Female 3 36358 72 1.59
56 Female 5 37554 72 1.49
```

Note that the first line has the names of the variables.

To read in this data type the follow:

```
> teachers <- read.table("teachers.dat", header=T, as.is=T)
```

The argument `as.is=T` stops R from trying to convert character data to factor objects (see Section 5.4). Without this argument, the `sex` column would be a factor with two levels (i.e., "Male" and "Female"). The arguement `header=T` informs R that the first line of the data file contains the names of the variables.

To look at the data we have just read in:

```
> teachers$age
 [1] 47 48 40 61 41 35 57 54 51 43 50 53 53 55 45 57 50 46 44 56

> teachers$sex
NULL

> teachers$gender
 [1] "Male"   "Male"   "Male"   "Male"   "Male"   "Male"
 [7] "Male"   "Male"   "Male"   "Male"   "Male"   "Female"
[13] "Female" "Female" "Female" "Female" "Female" "Female"
[19] "Female" "Female"

> teachers
   age    sex children salary weight height
1   47 Male         0  34377     94   1.72
2   48 Male         1  14502     79   1.86
3   40 Male         4  24899     88   1.49
 ... Output omitted ...
```

The names of variables in a data frame can be changed by typing:

```
> names(teachers)      # Before
[1] "age"      "sex"      "children" "salary"   "weight"
[6] "height"

> names(teachers) <- c("age", "gender", "children", "income",
+ "weight", "height")

> names(teachers)      # After
[1] "age"      "gender"   "children" "income"   "weight"
[6] "height"
```

5.2   Some notes on data frames

To use variables in a data frame we have been using the form `teachers$varname`. By using the function `attach()` we can refer to the variables by name. The function `detach` undoes the `attach()`.

```
> teachers$weight
 [1] "94" "79" "88" "79" "88" "94" "85" "90" "88" "90" "89" "83"
[13] "66" "64" "72" "88" "75" "65" "72" "72"
> weight
Error: Object "weight" not found

> attach(teachers)  # Attach the data frame.
> weight            # Typing the variable name will now work...
 [1] "94" "79" "88" "79" "88" "94" "85" "90" "88" "90" "89" "83"
[13] "66" "64" "72" "88" "75" "65" "72" "72"

> teachers$weight   # So does the old format.
 [1] "94" "79" "88" "79" "88" "94" "85" "90" "88" "90" "89" "83"
[13] "66" "64" "72" "88" "75" "65" "72" "72"

> detach(teachers)  # Detach the data frame.
> weight            # No longer works.
Error: Object "weight" not found
```

To check whether or not an object is a data frame we used the `is.data.frame()` function:

```
> is.data.frame(teachers)
[1] TRUE

> data <- cbind(c(1:10), rnorm(10))
> data
      [,1]          [,2]
 [1,]    1  0.54280499
 [2,]    2 -0.01827376
 [3,]    3  0.75249857
 [4,]    4  0.82214771
 [5,]    5 -0.96701648
 [6,]    6 -1.22995568
 [7,]    7  1.01985401
 [8,]    8 -1.93996275
 [9,]    9  1.00902315
[10,]   10 -0.87872980
> is.data.frame(data)
[1] FALSE
```

If the function returns `TRUE` then it is a data frame.

5.3   Writing data to a file

The function `write.table()` writes data to a file, for example:

```
demog <- cbind(teachers$age, teachers$weight, teachers$height)
```

```
> write.table(demog, "demog.dat")
```

writes the data in `demog` to the file *demog.dat*.

5.4   Factors

Factors are category objects (R's way of storing categorical variables). Suppose you have an experiment with six subjects, some of which are given treatment "a", some treatment "b", and some treatment "c". To store this information you can create a factor object:

```
> treat <-  factor(c('a','b','b','c','a','b'))
> treat
[1] a b b c a b
Levels:  a b c
```

Here the factor function has converted the character vector into a factor object. The resulting print of treat may look like a character object, but it is not. Notice that there are no quote marks around the letters. Compare with:

```
> treat.char <- c('a','b','b','c','a','b')
> treat.char
[1] "a" "b" "b" "c" "a" "b"
```

You can get a vector of the different categories in a factor with the levels function:

```
> levels(treat)
[1] "a" "b" "c"
```

This gives us a list of the unique categories in the factor.

Suppose we have the responses to the experiment in a vector:

```
> response <- c(10,3,7,6,4,5)
> rbind(treat, response)
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    2    2    3    1    2
[2,]   10    3    7    6    4    5
```

Note that `treat` is coded as `1`, `2` and `3` for treatment levels `a`, `b` and `c`, respectively.

We can then find the responses for a particular treatment by subscripting:

```
> response[treat=="a"]
[1] 10  4
> response[treat=="b"]
[1] 3 7 5
```

Sometimes the levels of factors will have a natural ordering that we would want to make use of in statistical analysis. The function `ordered()` creates such ordered factors. These are identical to factor with the exception that the ordered factors are printed in the order of levels. Factors are printed in alphabetical order. When fitting linear models the contrasts used are different (always make sure you understand what constracts are being used).

5.4.1 Generating factor variables

The function `gl()` is used to generate regular series of factor variables. The best description is in the help. Type:

```
> ?gl

  Generate Factor Levels

      gl(n, k, length = n*k, labels=1:n, ordered=FALSE)

  Value:

      This function generates 'factor's by specifying the
      pattern of their levels.  The result has levels from
      '1' to 'n' with each value replicated in groups of
      length 'k' out to a total length of 'length'.  Labels
      for the resulting factor levels can be optionally spec-
      ified with the arguments 'labels' and the factor levels
      can be made ordered by specifying 'ordered=TRUE'.  'gl'
      is modelled on the GLIM function of the same name.

  See Also:

      the underlying 'factor(.)'.

  Examples:

      # First control, then treatment:
      gl(2,8, label=c("Ctnrl","Treat"))
      # 20 alternating 1s and 2s
      gl(2, 1, 20)
      # alternating pairs of 1s and 2s
      gl(2, 2, 20)

> # Run the examples to see what happens.
> example(gl)
```

Try the following and observe what happens.

```
> gl(4, 3)
```

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

> gl(4, 2, 16)
 [1] 1 1 2 2 3 3 4 4 1 1 2 2 3 3 4 4
Levels:  1 2 3 4

> gl(4, 1, 16)
 [1] 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4
Levels:  1 2 3 4

> gl(4, 1, 8, label=c("None", "Low", "Medium", "High"))
[1] None    Low     Medium High    None    Low     Medium High
Levels:  None Low Medium High
```

1. The function `expand.grid()` creates a `data.frame` with all combination of a list of factors. Create a `data.frame` called `trial` with all combinations of the following factors:

   - `treatment` with levels "Placebo" and "Drug".
   - `sex` with levels "Male" and "Female".
   - `weeks` with 12 levels.

   Hint: Use the `help()` and `example()` functions.

2. Add a column called `outcome` to `trial` with values generated from a normal distrubtion with mean 17 and standard deviation 4.3.

3. Calculate the mean, median, standard deviation and the interquartile range by treatment and by sex.

4. Calculate the mean, median, standard deviation and the interquartile range by treatment and sex. *Hint: Use function* `tapply()`.

6 Scripts and Functions

6.1 Scripts

If you have a particularly complicated piece of R code, and you do not want to have to type it all in again if you make a mistake (for example, the long-winded version of the `t-test` code), you can put all the lines in a file and read it in with the source function.

Edit a file called `dmean.R` which contains the following lines:

```
x1 <- rnorm(100)
m1 <- mean(x1)
x2 <- rnorm(100)
```

```
m2 <- mean(x2)
dm <- abs(m1-m2)
```

Now in R type `source("dmean.R")`:

```
> source("dmean.R")
> dm
[1] 0.01060566
```

The source function reads in the file and executes the lines within in the same way as if they had been typed at the keyboard. After running you will see that it has created `x1`, `x2`, `m1` , `m2` and `dm`. Type:

```
> ls()
[1] "dm" "m1" "m2" "mx" "x1" "x2"
```

6.2 Functions

If you find you have a process that you use a lot, perhaps one that calls several R functions, you can write your own function to do it.

We will write function to calculate the geometric mean. Type the following into a file called `gmean.R`.

```
gmean <- function(x){
# geometric mean calculation
 prodx <- prod(x)
  n   <- length(x)
  gm <- prodx**(1/n)
  gm
}
```

Now use `source("gmean.R")`. If you do `ls()` now you will see that you have a new object called `gmean`. This is your new function object. Try calling it:

```
> gmean(1:10)
[1] 4.528729
```

Let us look at the function file in detail:

```
    gmean <- function(x){
```

This line is just an assignment - the object is called `gmean`, and it is going to be a function. The `(x)` specifies the arguments to the function - here there is just one argument, called `x`. The curly bracket then starts the code for the function.

```
    # geometric mean calculation
```

This is a comment. Anything after a `#` in Splus is a comment.

```
prodx <- prod(x)
n <- length(x)
gm <- prodx^(1/n)
```

This is the code for the geometric mean (in three lines). The result is now in the variable `gm`. Now we have to tell the function to return this value as its return value.

```
gm
}
```

The last thing evaluated by a function becomes its return value. We just put `gm` on a line, and then that becomes the return value.

If you type the name of a function without `()`, Splus prints the code of the function. Try that, type: `gmean`.

6.2.1 Exercise

1. Write a function that takes two vectors as arguments, `v1` and `v2`, and returns the difference between the medians of the two vectors. Call the vector `median.diff`.

2. Write a function that returns the mean of the values in a vector that are greater than the median. Use the `median()` function. Call your function `gtmean`.

7 Statistical Analysis

So far we have seen only seen some data manipulation and the calculation of summary statistics. R can do a multitude of statistical analysis including linear models (`lm`), generalised linear models (`glm`), analysis of variance (`aov` and `anova`), non linear models with mixed effects (`nlme` generalised additive models (`gam`), survival analysis (`survival`), time series analysis (`tseries`), multivariate analysis (`multiv`) and many more.

Some of these must be loaded as packages using the `library()` function.

7.1 Simple Linear Regression

We are going to use a simple example to illustrate some basic analysis in R. First will start by generating two vectors (variables) `x` and `y`.

```
> x <- 1:6
> y <- 2*x + rnorm(6)
> rbind(x, y)
          [,1]     [,2]     [,3]     [,4]     [,5]     [,6]
[1,] 1.000000 2.000000 3.000000 4.000000 5.000000  6.00000
[2,] 2.542805 3.981726 6.752499 8.822148 9.032984 10.77004
```

We want to investigate the relationship between x and y. Let us start by plotting x
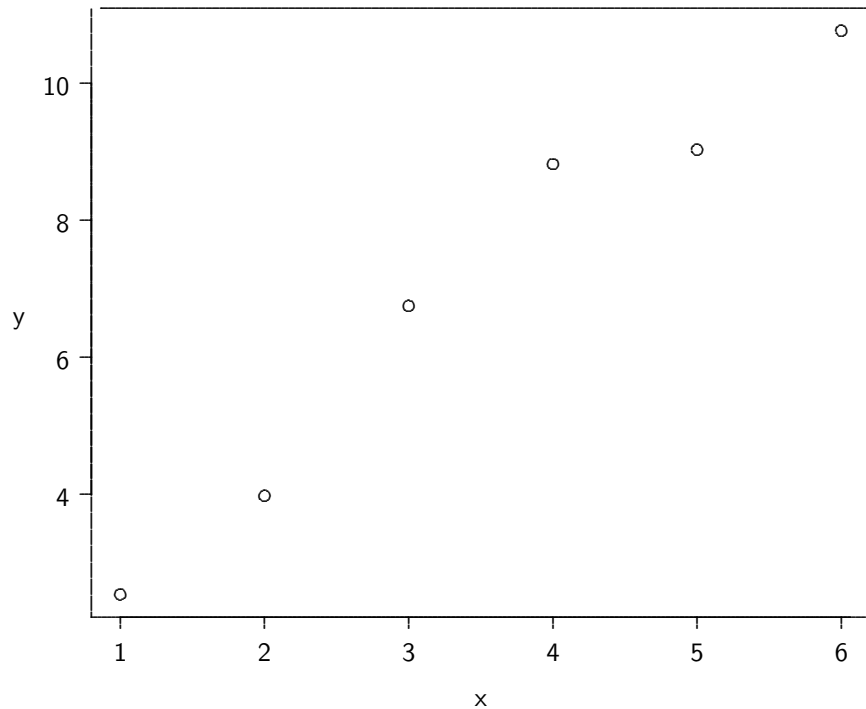again y.

```
> plot(x, y)
```


Figure 1: Plot of x against y

The plot indicates that there appears to be a linear relationship between x and y (even
if there is not in you case please continue with this example).

We will now use the `lm()` function to fit a simple linear regression model.

```
> lm(y~x)

Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
      1.148        1.667
```

Here we performed a linear regression of y on x. The intercept is `1.148` and the slope
parameter is `1.667`. Like any object in R we can save the results to another object,
say `modelxy`.

```
> modelxy <- lm(y~x)
```

If we type `modelxy` then this same results as above. Several functions allow us to display details about the statistical model. The `summary()` function displays more details about the model and the model fit.

```
> summary(modelxy)

Call:
lm(formula = y ~ x)

Residuals:
      1       2       3       4       5       6
-0.2724 -0.5008  0.6025  1.0047 -0.4518 -0.3822

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   1.1477     0.6661   1.723  0.15997
x             1.6674     0.1710   9.749  0.00062 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.7155 on 4 degrees of freedom
Multiple R-Squared: 0.9596,     Adjusted R-squared: 0.9495
F-statistic: 95.04 on 1 and 4 degrees of freedom, xp-value: 0.0006201
```

The residuals, coefficients and predicted values can be accessed using the `residuals()`, `coef()` and `predict()` functions, respectively.

```
> residuals(modelxy)
         1          2          3          4          5          6
-0.2723518 -0.5008482  0.6025065  1.0047380 -0.4518438 -0.3822007
>
> coef(modelxy)
(Intercept)          x
   1.147739    1.667418
>
> predict(modelxy)
        1         2         3         4         5         6
 2.815157  4.482574  6.149992  7.817410  9.484827 11.152245
```

You can plot the regression line using the function `abline()`.

```
> plot(x,y)
> abline(modelxy)
```

The list of elements of the results of an analysis can found using the `names()` function.
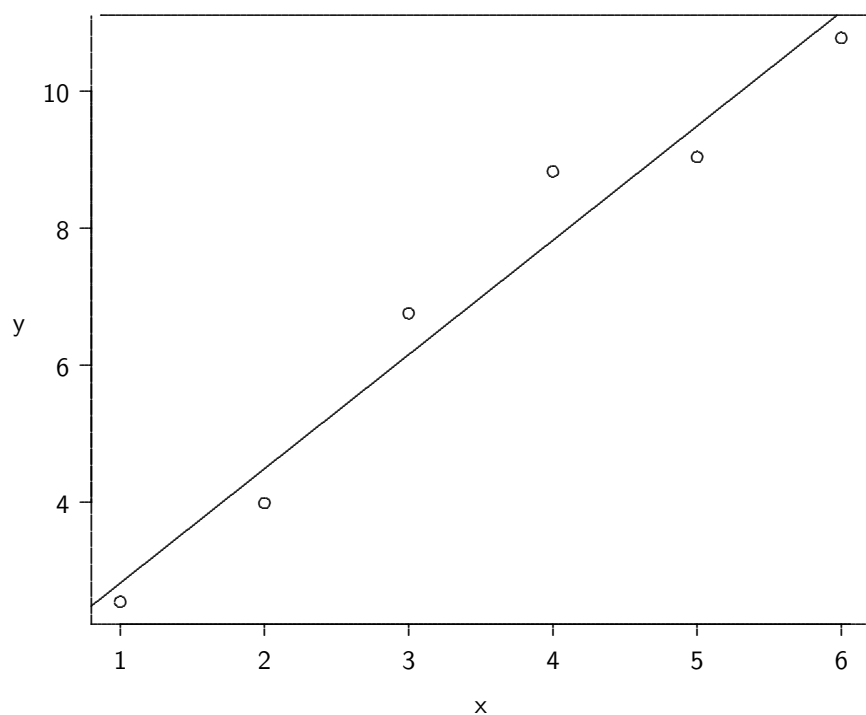
```
> names(modelxy)
```

Figure 2: Plot of x against y with the fitted line from a a simple linear regression.

```
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "xlevels"       "call"          "terms"         "model"

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

To extract the elements the follow notation can be used:

```
> modelxy$effects
(Intercept)           x
-17.1065037   6.9753085   0.6623385   0.9794860  -0.5621799  -0.5776208

> modelxy$df.residual
[1] 4

> modelxy["df.residual"]
$df.residual
[1] 4

> summary(modelxy)["r.squared"]
$r.squared

> summary(modelxy)$r.squared
```

```
[1] 0.959612
[1] 0.959612
```

8   Graphics

R graphics capabilities are very powerful and flexible. Given the range of possibilities, only the very basics are presented here.

8.1   Demonstration

R offer a demonstration of it's graphical capabilities. Type the following to see the demo:

```
> demo(graphics)
```

8.2   Simple Example

The following will plot the sine curve (see Figure 3).

```
> xrads <- seq(0, 2*pi, length=32)
> sinx <- sin(xrads)
> plot(xrads, sinx)
```

This shows us the default behaviour of the plot function - take two arguments, and plot a scatter plot of one against the other. Splus works out the axes for you, and labels them nicely too.

By using some optional arguments, we can make it join the dots up - try:

```
> plot(x,y,type='l')
```

This makes plot draw a line graph.

Once you have drawn a plot with plot you can add extra things to it. The functions lines and points add lines and points to a plot. Try the following (plots not shown):

```
> plot(xrads, sinx, type='l')    # lines first
> points(xrads, sinx)            # overlay points

> plot(xrads, sinx)              # new plot, points
> lines(xrads, sinx)             # add lines
```
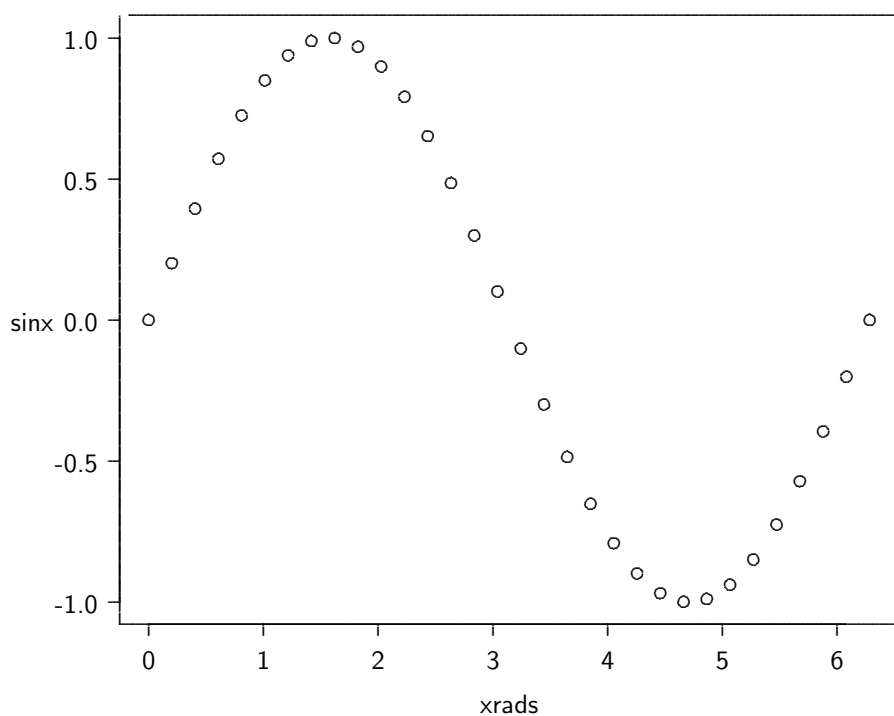
Now we can use the following to label the graph.

Figure 3: The sine curve - points only.

```
> plot(xrads, sinx,
+     main="Sine Curve",
+     xlab="Radians",
+     ylab="Sin(Radians)",
+     type="b")
```

The `main` option is to title graph, `xlab` and `ylab` label the x and y axes, `type=''b''` plots both lines and points.

To plot a horizontal line at zero and a vertical line at $\frac{\pi}{2}$, π and $\frac{3\pi}{2}$, type:

```
> abline(h=0, v=c(pi/2, pi, 3*pi/2))
```

8.3   Exercises

1. Plot a graph that shows three curves - `y=x`, `y=x**2`, and `y=sqrt(x)`, for x from 0 to 3. Plot a vertical line at 1, 2 and 3.

2. Plot a graph that shows `y=1/x` for `x` from 1 to 10 using a line and the points. Plot a horizontal line at 0.75.

3. Plot the sine and cosine curves on the same plot. For the sine curve use a line and for the cosine curve use points.