

STATISTICS: AN INTRODUCTION USING R

By M.J. Crawley

Exercises

2. VECTORS AND LOGICAL ARITHMETIC

The screen prompt is an excellent calculator. All of the usual calculations can be done directly, and the standard order of precedence applies: powers and roots first, then multiplication and division, then finally addition and subtraction. Although there is a named function for square roots, **sqrt**, roots are generally calculated as fractional powers. So the cube root of 8 is

```
8^(1/3)
```

```
[1] 2
```

Use brackets as necessary to over-ride the precedence. For a t-test, where the value required is $\frac{|y_A - y_B|}{SE_d}$ and the numbers are 5.7, 6.8 and 0.38 you type

```
abs(5.7-6.8)/0.38
```

```
[1] 2.894737
```

All the mathematical functions you could ever want are here (see Table 1).

Integer quotients and remainders are obtained using the rather clumsy notation `%/%` (percent, divide, percent) and `%%` (percent, percent) respectively. Suppose we want to know the integer part of a division: say, how many 13's are there in 119

```
119 %/% 13
```

```
[1] 9
```

Now suppose we wanted to know the remainder (what is left over when 119 is divided by 13): in maths this is known as **modulo**

```
119 %% 13
```

```
[1] 2
```

Various sorts of rounding (rounding up, rounding down, rounding to the nearest integer) can be done easily. Take 5.7 as an example. The 'greatest integer' function is **floor**

```
floor(5.7)
```

```
[1] 5
```

and the 'next integer' function is **ceiling**

```
ceiling(5.7)
```

```
[1] 6
```

You can round to the nearest integer by adding 0.5 to the number then using **floor**. There is a built in function for this, but we can easily write one of our own to introduce the notion of function-writing. Call it *rounded*, then define it as a function like this

```
rounded<-function(x) floor(x+0.5)
```

Now we can use the new function

```
rounded(5.7)
```

```
[1] 6
```

Table 1. Mathematical functions.

Function	Meaning
log(x)	log to base e of x
exp(x)	antilog of x (2.7818^x)
log(x,n)	log to base n of x
log10(x)	log to base 10 of x
sqrt(x)	square root of x
factorial(x)	x!
choose(n,x)	binomial coefficients $n! / (x! (n-x)!)$
gamma(x)	Γx (x-1)! for integer x
lgamma(x)	natural log of gamma x
floor(x)	greatest integer < x
ceiling(x)	smallest integer > x
trunc(x)	closest integer to x between x and 0 trunc(1.5) = 1, trunc(-1.5) = -1 trunc is like floor for positive values and like ceiling for negative values
round(x, digits=0)	round the value of x to an integer
signif(x, digits=6)	give x to 6 digits in scientific notation
runif(n)	generates n random numbers between 0 and 1 from a uniform distribution
cos(x)	cosine of x in radians
sin(x)	sine of x in radians
tan(x)	tangent of x in radians
acos(x), asin(x), atan(x)	inverse trigonometric transformations of real or complex numbers.
acosh(x), asinh(x), atanh(x)	inverse hyperbolic trigonometric transformations on real or complex numbers
abs(x)	the absolute value of x, ignoring the minus sign if there is one

Generating sequences

The simplest way to generate a regularly spaced sequence of numbers is to use the colon operator like this:

```
1:7
```

```
[1] 1 2 3 4 5 6 7
```

For more complicated sequences, use the **seq** function in which the 3rd argument can be the step length (the increment or decrement you want to use): going up

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

```
[1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

or coming down

```
seq(5,-5,-1)
```

```
[1] 5 4 3 2 1 0 -1 -2 -3 -4 -5
```

The **along** option allows you to map a sequence onto an existing vector (to ensure equal lengths) or if you know how many numbers you want but you can't be bothered to work out the final value of a series, you can do this. Suppose we want a series of 20 numbers, starting at 5 and going down in steps of -1:

```
seq(from=5,by=-1, along=1:20)
```

```
[1] 5 4 3 2 1 0 -1 -2 -3 -4 -5 -6 -7 -8 -9 -10 -11  
-12 -13 -14
```

Generating repeats

The **rep** function replicates the input either a certain number of times or to a certain length. In the unlikely event that you wanted to get 5.3 repeated 17 times you would type

```
rep(5.3,17)
```

```
[1] 5.3 5.3 5.3 5.3 5.3 5.3 5.3 5.3 5.3 5.3 5.3 5.3 5.3 5.3  
5.3 5.3 5.3 5.3
```

The first parameter is the number and the second parameter is the repeat. More useful applications involve repeating sets of numbers, variable numbers of times. The general syntax is like this:

`rep(x, times)`

If `times` consists of a single integer, the result consists of the values in `x` repeated this many times. If `times` is a vector of the same length as `x`, the result consists of `x[1]` repeated `times[1]` times, `x[2]` repeated `times[2]` times and so on. The thing to remember is that the *length* of the second argument (the number of each repeat) has to match the length of the first.

`rep(1:6,2)`

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

This says repeat the whole series 1 to 6, twice. That's fine, but what if we wanted each of the numbers 1 to 6 repeated, say, 3 times. We need to write a list of six 3's explicitly like this:

`rep(1:6,rep(3,6))`

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

This emulates the old 'generate levels' function for those of you who grew up with GLIM. In R there is a direct copy of generate levels called `gl`. The third argument is the length of the whole vector (18 in this case)

`gl(6,3,18)`

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

```
Levels: 1 2 3 4 5 6
```

The most complex case arises when we want to repeat each element of the first list a different number of times. One very symmetric case might be if we wanted one 1, two 2's, three 3's and so on. This is

`rep(1:6,1:6)`

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

but more generally, we would like to be able to specify each repeat separately. We do that by using `concatenate, c`, to list each of the individual repeats in sequence. The list of numbers inside `c(...)` must match the length of the initial sequence:

`rep(1:6,c(1,2,3,3,2,1))`

This says that the 1 is repeated once, the 2 twice, the 3 and 4 thrice, the 5 twice and the 6 once:

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

Finally, we might want to customise both the sequence and the repeat. This is especially useful in generating repeated factor levels that are text:

```
rep(c("monoecious", "dioecious", "hermaphrodite", "agamic"), c(3, 2, 7, 3))
```

```
[1] "monoecious" "monoecious" "monoecious" "dioecious" "dioecious" "hermaphrodite"
[7] "hermaphrodite" "hermaphrodite" "hermaphrodite" "hermaphrodite" "hermaphrodite" "hermaphrodite"
[13] "agamic" "agamic" "agamic"
```

Table 2. Logical operations

Symbol	Meaning
!=	not equal
%%	the remainder of a division, modulo
%/%	the integer part
*	multiplication
+	addition
-	subtraction
^	to the power
/	division
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	logical equals (double =)

Vectors

The real power of the calculator is manifest in its abilities for vector calculations. Suppose that y contains the following 10 values:

```
y<-c(7,5,7,2,4,6,1,6,2,3)
```

where **c** is the concatenation function. This is a long name for a simple idea; **c** combines objects from left to right into a vector (as here) or a list, tagging the next item onto the bottom of the list so far. This is useful if you want a vector to be twice as long, with the contents repeated. For example, to string together 2 copies of the vector called y, end to end, we just write:

```
y<-c(y,y)
```

y

```
[1] 7 5 7 2 4 6 1 6 2 3 7 5 7 2 4 6 1 6 2 3
```

The most commonly used vector functions are **sum** and **mean**. Although we don't need to do this, because there is a built in function **var** to calculate the variance of a vector of numbers, it is useful as a demonstration of sum and mean in action. The formula we want to apply to our vector is the familiar definition of sample variance

$$s^2 = \frac{\sum (y - \bar{y})^2}{n - 1}$$

which involves both sum \sum and mean \bar{y} . There is an extra vector function that is very useful here. In this case we know that there are 20 numbers in y so we could divide by 19 to get the variance. But this is no good in general. The sample size n is found as the **length** of any vector, so we can write the variance like this:

```
sum((y-mean(y))^2)/(length(y)-1)
```

```
[1] 4.642105
```

In this case, we can use the built in function to check whether we got it right:

```
var(y)
```

```
[1] 4.642105
```

Sorting, Ranking and Ordering

These three related concepts are important and one of them (order) is difficult to understand on first acquaintance. Let's take a simple example

```
data<-c(7,4,6,8,9,1,0,3,2,5,0)
```

Now we apply the 3 different functions to the vector called data

```
ranks<-rank(data)
sorted<-sort(data)
ordered<-order(data)
```

We make a data frame out of the 4 vectors like this

```
view<-data.frame(data,ranks,sorted,ordered)
view
```

	data	ranks	sorted	ordered
1	7	9.0	0	7
2	4	6.0	0	11
3	6	8.0	1	6
4	8	10.0	2	9
5	9	11.0	3	8
6	1	3.0	4	2
7	0	1.5	5	10
8	3	5.0	6	3
9	2	4.0	7	1
10	5	7.0	8	4
11	0	1.5	9	5

Rank

The data themselves are in no particular sequence. Ranks contains the value that is the rank, out of **length**(data), of the particular data point. So the first element data = 7 is the 9th highest value in data. You should check that there are 8 values smaller than 7 in the vector called data. Fractional ranks indicate ties. There are 2 zero's in the data and their ranks are 1 and 2. Because they are tied, each gets the average of the sum of their ranks $(1+2)/2 = 1.5$.

Sort

The sorted vector is very straightforward. It contains the values of data sorted into ascending order. If you want to sort into descending order, use the reverse order directive **rev** like this `y<-rev(sort(x))`. Note that **sort is potentially very dangerous**, because it uncouples values that might need to be in the same row of the data frame (e.g. because they are the explanatory variables associated with a particular value of the response variable).

Order

This returns an integer vector containing the permutation that will sort the input into ascending order. You will need to think about this one. Look at the data frame and ask yourself "What is the subscript in the original vector called data of the value in the 1st element of the sorted vector. The 1st zero is in location 7 and this is `order[1]`. Where is the 2nd value in the sorted vector found within the original data? It is in position 11, so this is `order[2]`. The only 1 is found in position 6 within data, so this is `order[3]`. And so on.

Dataframes

A dataframe is an object with rows and columns (a bit like a 2-dimensional matrix). The rows contain different observations from your study, or measurements from your experiment. The columns contain different variables. The values in the body of the dataframe can be numbers (as they would be in a matrix), but they could also be text (e.g. the names of factor levels for categorical variables, like “male” or “female” in a variable called “gender”), they could be calendar dates (like 23/5/04), or they could be logical variables (like “TRUE” or “FALSE”). Here is a dataframe with 7 variables, the left-most of which comprises the row names, and other variables are numeric (Area, Slope, Soil pH and Worm density), categorical (Field Name and Vegetation) or logical (Damp is either true = T or false = F).

Field Name	Area	Slope	Vegetation	Soil pH	Damp	Worm density
Nash's Field	3.6	11	Grassland	4.1	F	4
Silwood Bottom	5.1	2	Arable	5.2	F	7
Nursery Field	2.8	3	Grassland	4.3	F	2
Rush Meadow	2.4	5	Meadow	4.9	T	5
Gunness' Thicket	3.8	0	Scrub	4.2	F	6
Oak Mead	3.1	2	Grassland	3.9	F	2
Church Field	3.5	3	Grassland	4.2	F	3
Ashurst	2.1	0	Arable	4.8	F	4
The Orchard	1.9	0	Orchard	5.7	F	9
Rookery Slope	1.5	4	Grassland	5	T	7
Garden Wood	2.9	10	Scrub	5.2	F	8
North Gravel	3.3	1	Grassland	4.1	F	1
South Gravel	3.7	2	Grassland	4	F	2
Observatory Ridge	1.8	6	Grassland	3.8	F	0
Pond Field	4.1	0	Meadow	5	T	6
Water Meadow	3.9	0	Meadow	4.9	T	8
Cheapside	2.2	8	Scrub	4.7	T	4
Pound Hill	4.4	2	Arable	4.5	F	5
Gravel Pit	2.9	1	Grassland	3.5	F	1
Farm Wood	0.8	10	Scrub	5.1	T	3

Perhaps the most important thing about analysing your own data properly is getting your dataframe absolutely right. The expectation is that you will have used a spreadsheet like Excel to enter and edit the data, and that you will have used plots to check for errors. The thing that takes some practice is learning exactly how to put your numbers into the spreadsheet. There are countless ways of doing it wrong, but only one way of doing it right. And this way is not the way that most people find intuitively to be the most obvious.

The key thing is this: *all values of the same variable must go in the same column*. It does not sound like much, but this is what people tend not to do. If you had an experiment with a control, pre-heated and pre-chilled as three treatments, and 4 measurements per treatment, it might seem like a good idea to create the spreadsheet like this:

control	preheated	prechilled			
6.1	6.3	7.1			
5.9	6.2	8.2			
5.8	5.8	7.3			
5.4	6.3	6.9			

This is not a dataframe, because values of the response variable appear in 3 different columns. The correct way to enter these data is to have two columns: one for the response and one for the levels of the experimental factor. Here are the same data, entered correctly as a dataframe:

response	treatment			
6.1	control			
5.9	control			
5.8	control			
5.4	control			
6.3	preheated			
6.2	preheated			
5.8	preheated			
6.3	preheated			
7.1	prechilled			
8.2	prechilled			
7.3	prechilled			
6.9	prechilled			

A good way to practice this layout is to use the excellent Excel function called Pivot Table (found under the Data tab on the main menu bar) on your own data: it requires your spreadsheet to be in the form of a dataframe, with each of the explanatory variables in its own column.

Once you have made your dataframe in Excel and corrected all the inevitable data-entry and spelling errors, then you need to save the dataframe in a file format that can be read by R. Much the simplest way is to save all your dataframes from Excel as tab-delimited text files: File / Save As / ... then from the “Save as type” options choose “Text (Tab delimited)”. There is no need to add a suffix, because Excel will automatically add “.txt” to your file name. This file can then be read into R directly as a dataframe, using the `read.table` function.

It is important to note that `read.table` would fail if there were any spaces in any of the variable names in row 1 of the dataframe (the header row) like Field Name, Soil pH or Worm Density, or between any of the words within the same factor level (as in many of

the Field Names). We should replace all these spaces by dots “.” before saving the dataframe in Excel (use Edit /Replace with “ “ replaced by “.”). Now the dataframe can be read into R. There are 3 things to remember:

- the whole path and file name needs to be enclosed in double quotes: “c:\\abc.txt”
- **header =T** says that the first row contains the variable names
- always use double backslash \\ rather than \ in the file path definition

Think of a name for the data frame (say “worms” in this case). Now use the **gets arrow** <- which is made up of the two characters < (less than) and - (minus) like this

```
worms<-read.table("c:\\temp\\worms.txt",header=T,row.names=1)
```

Once the file has been imported to R we want to do two things:

- use **attach** to make the variables accessible by name within the R session
- use **names** to get a list of the variable names

Typically, the 2 commands are issued in sequence, whenever a new data frame is imported from file:

```
attach(worms)
names(worms)
```

```
[1] "Area"           "Slope"           "Vegetation"
[4] "Soil.pH"        "Damp"            "Worm.density"
```

To see the contents of the dataframe, just type its name:

```
worms
```

	Area	Slope	Vegetation	Soil.pH	Damp	Worm.density
Nash's.Field	3.6	11	Grassland	4.1	FALSE	4
Silwood.Bottom	5.1	2	Arable	5.2	FALSE	7
Nursery.Field	2.8	3	Grassland	4.3	FALSE	2
Rush.Meadow	2.4	5	Meadow	4.9	TRUE	5
Gunness'.Thicket	3.8	0	Scrub	4.2	FALSE	6
Oak.Mead	3.1	2	Grassland	3.9	FALSE	2
Church.Field	3.5	3	Grassland	4.2	FALSE	3
Ashurst	2.1	0	Arable	4.8	FALSE	4
The.Orchard	1.9	0	Orchard	5.7	FALSE	9
Rookery.Slope	1.5	4	Grassland	5.0	TRUE	7
Garden.Wood	2.9	10	Scrub	5.2	FALSE	8
North.Gravel	3.3	1	Grassland	4.1	FALSE	1
South.Gravel	3.7	2	Grassland	4.0	FALSE	2

Observatory.Ridge	1.8	6	Grassland	3.8	FALSE	0
Pond.Field	4.1	0	Meadow	5.0	TRUE	6
Water.Meadow	3.9	0	Meadow	4.9	TRUE	8
Cheapside	2.2	8	Scrub	4.7	TRUE	4
Pound.Hill	4.4	2	Arable	4.5	FALSE	5
Gravel.Pit	2.9	1	Grassland	3.5	FALSE	1
Farm.Wood	0.8	10	Scrub	5.1	TRUE	3

If, as here, the rows have unique names as part of the dataframe, we suppress R's natural inclination to produce its own row numbers, by telling R the number of the column containing the row names (1 in this case). Notice, also, that R has expanded our abbreviated T and F into TRUE and FALSE. The object called worms now has all the attributes of a dataframe. For example, you can summarize it, using `summary`:

```
summary(worms)
```

Area		Slope		Vegetation		Soil.pH	
Min.	:0.800	Min.	: 0.00	Arable	:3	Min.	:3.500
1st Qu.	:2.175	1st Qu.	: 0.75	Grassland	:9	1st Qu.	:4.100
Median	:3.000	Median	: 2.00	Meadow	:3	Median	:4.600
Mean	:2.990	Mean	: 3.50	Orchard	:1	Mean	:4.555
3rd Qu.	:3.725	3rd Qu.	: 5.25	Scrub	:4	3rd Qu.	:5.000
Max.	:5.100	Max.	:11.00			Max.	:5.700
Damp		Worm.density					
Length	:20	Min.	:0.00				
Mode	:logical	1st Qu.	:2.00				
		Median	:4.00				
		Mean	:4.35				
		3rd Qu.	:6.25				
		Max.	:9.00				

Note that the summary does not count the numbers of cases of TRUE and FALSE in the logical variable called Damp, but it does count the number of cases of each Vegetation type. Note that the Field Names are not summarized, because they have been declared to be the row names (and hence all the names have to be unique).

Selecting parts of a dataframe: Subscripts

To select just the first 3 columns of the data frame, use subscript `[,1:3]`. The “blank then comma” means “all the rows”, just as the “comma then blank” means all the columns. So to select all the rows of the first 3 columns of worms, we write:

worms[,1:3]

	Area	Slope	Vegetation
Nash's.Field	3.6	11	Grassland
Silwood.Bottom	5.1	2	Arable
Nursery.Field	2.8	3	Grassland
Rush.Meadow	2.4	5	Meadow
Gunness'.Thicket	3.8	0	Scrub
Oak.Mead	3.1	2	Grassland
Church.Field	3.5	3	Grassland
Ashurst	2.1	0	Arable
The.Orchard	1.9	0	Orchard
Rookery.Slope	1.5	4	Grassland
Garden.Wood	2.9	10	Scrub
North.Gravel	3.3	1	Grassland
South.Gravel	3.7	2	Grassland
Observatory.Ridge	1.8	6	Grassland
Pond.Field	4.1	0	Meadow
Water.Meadow	3.9	0	Meadow
Cheapside	2.2	8	Scrub
Pound.Hill	4.4	2	Arable
Gravel.Pit	2.9	1	Grassland
Farm.Wood	0.8	10	Scrub

To select just the middle 10 rows of the data frame, use subscript [5:15,]

worms[5:15,]

	Area	Slope	Vegetation	Soil.pH	Damp	Worm.density
Gunness'.Thicket	3.8	0	Scrub	4.2	FALSE	6
Oak.Mead	3.1	2	Grassland	3.9	FALSE	2
Church.Field	3.5	3	Grassland	4.2	FALSE	3
Ashurst	2.1	0	Arable	4.8	FALSE	4
The.Orchard	1.9	0	Orchard	5.7	FALSE	9
Rookery.Slope	1.5	4	Grassland	5.0	TRUE	7
Garden.Wood	2.9	10	Scrub	5.2	FALSE	8
North.Gravel	3.3	1	Grassland	4.1	FALSE	1
South.Gravel	3.7	2	Grassland	4.0	FALSE	2
Observatory.Ridge	1.8	6	Grassland	3.8	FALSE	0
Pond.Field	4.1	0	Meadow	5.0	TRUE	6

It is often useful to select certain rows, based on the logical tests on the values of one or more variables. Here is the code to select only those rows with Area > 3 and Slope < 3:

worms[Area>3 & Slope <3,]

	Area	Slope	Vegetation	Soil.pH	Damp	Worm.density
Silwood.Bottom	5.1	2	Arable	5.2	FALSE	7
Gunness'.Thicket	3.8	0	Scrub	4.2	FALSE	6
Oak.Mead	3.1	2	Grassland	3.9	FALSE	2
North.Gravel	3.3	1	Grassland	4.1	FALSE	1
South.Gravel	3.7	2	Grassland	4.0	FALSE	2
Pond.Field	4.1	0	Meadow	5.0	TRUE	6
Water.Meadow	3.9	0	Meadow	4.9	TRUE	8
Pound.Hill	4.4	2	Arable	4.5	FALSE	5

Sorting

You can sort the rows or the columns of the data frame in any way you choose, but you need to state which columns you want to be sorted (typically you want them all sorted; i.e. columns 1:6 in the present case). Suppose we want the rows of the whole data frame sorted by Area (this is the first column [,1]):

```
worms[order(worms[,1]),1:6]
```

	Area	Slope	Vegetation	Soil.pH	Damp	Worm.density
Farm.Wood	0.8	10	Scrub	5.1	TRUE	3
Rookery.Slope	1.5	4	Grassland	5.0	TRUE	7
Observatory.Ridge	1.8	6	Grassland	3.8	FALSE	0
The.Orchard	1.9	0	Orchard	5.7	FALSE	9
Ashurst	2.1	0	Arable	4.8	FALSE	4
Cheapside	2.2	8	Scrub	4.7	TRUE	4
Rush.Meadow	2.4	5	Meadow	4.9	TRUE	5
Nursery.Field	2.8	3	Grassland	4.3	FALSE	2
Garden.Wood	2.9	10	Scrub	5.2	FALSE	8
Gravel.Pit	2.9	1	Grassland	3.5	FALSE	1
Oak.Mead	3.1	2	Grassland	3.9	FALSE	2
North.Gravel	3.3	1	Grassland	4.1	FALSE	1
Church.Field	3.5	3	Grassland	4.2	FALSE	3
Nash's.Field	3.6	11	Grassland	4.1	FALSE	4
South.Gravel	3.7	2	Grassland	4.0	FALSE	2
Gunness'.Thicket	3.8	0	Scrub	4.2	FALSE	6
Water.Meadow	3.9	0	Meadow	4.9	TRUE	8
Pond.Field	4.1	0	Meadow	5.0	TRUE	6
Pound.Hill	4.4	2	Arable	4.5	FALSE	5
Silwood.Bottom	5.1	2	Arable	5.2	FALSE	7

or in descending order by Soil pH, with only Field Name and Worm.density as output

```
worms[rev(order(worms[,4])),c(4,6)]
```

	Soil.pH	Worm.density
The.Orchard	5.7	9
Garden.Wood	5.2	8
Silwood.Bottom	5.2	7
Farm.Wood	5.1	3
Pond.Field	5.0	6
Rookery.Slope	5.0	7
Water.Meadow	4.9	8
Rush.Meadow	4.9	5
Ashurst	4.8	4
Cheapside	4.7	4
Pound.Hill	4.5	5
Nursery.Field	4.3	2
Church.Field	4.2	3
Gunness'.Thicket	4.2	6
North.Gravel	4.1	1

Nash's.Field	4.1	4
South.Gravel	4.0	2
Oak.Mead	3.9	2
Observatory.Ridge	3.8	0
Gravel.Pit	3.5	1

Here is a simple example to demonstrate the beauty of the `order` function

```
houses <- read.table("c:\\temp\\houses.txt",header=T)
names(houses)
attach(houses)
houses
```

	Location	Price
1	Ascot	325
2	Sunninghill	201
3	Bracknell	157
4	Camberley	162
5	Bagshot	164
6	Staines	101
7	Windsor	211
8	Maidenhead	188
9	Reading	95
10	Winkfield	117
11	Warfield	188
12	Newbury	121

To see how this works, let's inspect the order of the house prices:

```
order(Price)
```

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

This says that the lowest price is in subscript 9 of (price = 95) where Location = Reading. Next cheapest is in location 6 (price = 101) where Location = Staines, and so on. The beauty of it is that we can use `order(Price)` as a subscript for Location to obtain the price-ranked list of locations:

```
Location[order(Price)]
```

[1]	Reading	Staines	Winkfield	Newbury
[5]	Bracknell	Camberley	Bagshot	Maidenhead
[9]	Warfield	Sunninghill	Windsor	Ascot

When you see it used like this, you can see exactly why the function is called 'order'. If you want to reverse the order, just use the `rev` function like this:

```
Location[rev(order(Price))]
```

[1] Ascot	Windsor	Sunninghill	Warfield
[5] Maidenhead	Bagshot	Camberley	Bracknell
[9] Newbury	Winkfield	Staines	Reading

Subscripts with vectors

The use of subscripts in S Plus is superb. It is so simple, yet so powerful. There are basically two things you would want to do with subscripts:

- select values from a vector according to some logical criterion (i.e. questions involving the **contents** of the vector)
- discover which elements of a vector (i.e. which subscripts) contain certain values (i.e. questions involving the **addresses** of items within the vector)

Operation	Meaning
max(x)	maximum value in x
min(x)	minimum value in x
sum(x)	total of all the values in x
mean(x)	arithmetic average of the values in x
median(x)	median value in x
range(x)	vector of min(x) and max(x)
var(x)	sample variance of x, with degrees of freedom = length(x)-1
cor(x,y)	correlation between vectors x and y
sort(x)	a sorted version of x
rank(x)	vector of the ranks of the values in x
order(x)	an integer vector containing the permutation to sort x into ascending order
quantile(x)	vector containing the minimum, lower quartile, median, upper quartile, and maximum of x
cumsum(x)	vector containing the sum of all of the elements to that point.
cumprod(x)	vector containing the product of all of the elements to that point.
cummax(x)	vector of non-decreasing numbers which are the cumulative maxima of the values in x to that point.
cummin(x)	vector of non-increasing numbers which are the cumulative minima of the values in x to that point.
pmax(x,y,z)	vector, of length equal to the longest of x,

	y, or z containing the maximum of x, y or z for the ith position in each
<code>pmin(x,y,z)</code>	vector, of length equal to the longest of x, y, or z containing the minimum of x, y or z for the ith position in each
<code>colMeans(x)</code>	column means of data frame x
<code>ColSums</code>	column totals of data frame x
<code>ColVars</code>	column variances of data frame x
<code>RowMean</code>	row means of data frame x
<code>RowSums</code>	row totals of data frame x
<code>RowVars</code>	row variances of data frame x
<code>peaks(x)</code>	logical vector of length(x) with T for peak values bigger than both their neighbours

In working with vectors, it is important to understand the distinction between

- the logical status of a number (True or False)
- the value of a number (0.64 or 2541)

Take the example of a vector containing the 11 numbers 0 to 10

```
x<-0:10
```

There are two quite different things we might want to do with this. We might want to *add up* the values of the elements:

```
sum(x)
```

```
[1] 55
```

Alternatively we might want to *count* the elements that passed some logical criterion. Suppose we wanted to know how many of the values were less than 5:

```
sum(x<5)
```

```
[1] 5
```

You see the distinction. We use the vector function **sum** in both cases. But **sum(x)** adds up the values of the *x*'s and **sum(x<5)** counts up the number of cases that pass the logical condition “*x* is less than 5”.

That is all well and good, but how do you add up the values of just some of the elements of *x*? We specify a logical condition, but we don't want to count the number of cases that pass the condition, we want to add up all the values of the cases that pass. This is the final

piece of the jigsaw, and involves the use of *logical subscripts*. Note that when we counted the number of cases, the counting was applied to the entire vector, using `sum(x<5)`. To find the sum of the x values that are less than 5, we write:

```
sum(x[x<5])
```

```
[1] 10
```

The logical condition `x<5` is either true or false.

```
x<5
```

```
[1] T T T T T F F F F F F
```

You can imagine false as being numeric zero and true as being numeric one. Then the vector of subscripts `[x<5]` is 5 1's followed by 6 0's.

```
1*(x<5)
```

```
[1] 1 1 1 1 1 0 0 0 0 0 0
```

Now imagine multiplying the values of x by the values of the logical vector

```
x*(x<5)
```

```
[1] 0 1 2 3 4 0 0 0 0 0 0
```

When the function **sum** is applied, it gives us the answer we want: the sum of the values of the numbers $0 + 1 + 2 + 3 + 4 = 10$.

```
sum(x*(x<5))
```

```
[1] 10
```

The same answer as `sum(x[x<5])`, but rather less elegant.

Suppose we want to work out the sum of the 3 largest values in a vector. There are two steps: first sort the vector into descending order. Then add up the values of the first 3 elements of the sorted array. Let's do this in stages. First, the values of y:

```
y<-c(8,3,5,7,6,6,8,9,2,3,9,4,10,4,11)
```

Now if you apply **sort** to this, the numbers will be in ascending sequence, and this makes life a bit harder for the present problem:

```
sort(y)
```

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

We can use the **reverse** function like this (use the Up Arrow key to save typing):

```
rev(sort(y))
```

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

So the answer to our problem is $11 + 10 + 9 = 30$. But how to compute this? We can use specific subscripts to discover the contents of any element of a vector. We can see that 10 is the second element of the sorted array. To compute this we just specify the subscript [2]

```
rev(sort(y))[2]
```

```
[1] 10
```

A range of subscripts is simply a series generated using the colon operator. We want the subscripts 1 to 3, so this is:

```
rev(sort(y))[1:3]
```

```
[1] 11 10 9
```

So the answer to the exercise is just

```
sum(rev(sort(y))[1:3])
```

```
[1] 30
```

Note that we have not changed the vector `y` in any way, nor have we created any new space-consuming vectors during intermediate computational steps.

Addresses

There are two important functions for finding addresses within arrays. The function **which** is very easy to understand. The vector `y` looks like this:

```
y<-c(8, 3, 5, 7, 6, 6, 8, 9, 2, 3, 9, 4, 10, 4, 11)
```

Suppose we wanted to know which elements of `y` contained values bigger than 5 we type

```
which(y>5)
```

```
[1] 1 4 5 6 7 8 11 13 15
```

Notice that the answer to this enquiry is *a set of subscripts*. We *don't* use subscripts in the **which** directive itself. The function is applied to the whole array. To see the *values of y* that are larger than 5 we just type

```
y[y>5]
```

```
[1] 8 7 6 6 8 9 9 10 11
```

If we want to know whether or not each element of *y* is bigger than 5 we just do this:

```
y>5
```

```
[1] TRUE FALSE FALSE TRUE TRUE TRUE TRUE TRUE FALSE FALSE TRUE FALSE TRUE FALSE  
[15] TRUE
```

and the answer is a logical vector of TRUE's and FALSE's.

Sample

This function *shuffles the contents of a vector* into a random sequence while maintaining all the numerical values intact. It is extremely useful in bootstrapping, simulation and Monte Carlo techniques of computationally intensive hypothesis testing. Here is the original *y* again:

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

and here are 2 samples of *y*

```
sample(y)
```

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

```
sample(y)
```

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

This is “sampling without replacement”: all the values appear once and only once in the shuffled list. The option **replace=T** allows for sampling *with replacement*, which is the basis of bootstrapping (see Practical 12). The vector produced by the **sample** function is the same length as the vector sampled, but some values are left out at random and other values, again at random, appear 2 or more times. The values selected will be different each time that **sample** is invoked.

```
sample(y,replace=T)
```

```
[1] 9 6 11 2 9 4 6 8 8 4 4 4 3 9 3
```

In this sample, 10 has been left out, and are now three 9's.

```
sample(y,replace=T)
```

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

while in this one, there are two 10's and only one 9.

Logical arithmetic

A really useful computing skill involves the use of logical statements in arithmetic calculations. This takes advantage of the fact that logical "true" evaluates to 1.0 and logical "false" evaluates to 0.0. When we make a command only involving logic we get back a logical statement, T or F

Note that 'equals' is handled rather oddly in logical statements. We need to use "==" (double equals) to mean 'exactly equal to'. Suppose we want to add 1 to the value of y if y = 3 but otherwise leave y unaltered:

```
y
```

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

```
y+(y==3)
```

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

It is often important to make one or more shorter versions of an array, and logical subscripting is a superb way of doing this. Let's make an array *ys* containing the small values of *y*, and *yb* containing the big values.

```
ys<-y[y<5]
ys
```

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

```
yb<-y[y>=5]
yb
```

```
[1] 8 5 7 6 6 8 9 9 10 11
```

We can count the number of elements in an array that have various (potentially quite complicated) properties. For example we can count the number of extreme values in *y*, defining extreme values as those more than 2 greater than the mean and those less than 2 below the mean (there are 9 of them in this case)

```
sum(y> mean(y)+2 | y < mean(y)-2 )
```

```
[1] 9
```

Note the use of the 'or operator', |, to separate the two parts of the expression. If we want the total of the *y* values that are extreme in one direction or the other we type

```
sum(y[y> mean(y)+2 | y < mean(y)-2] )
```

```
[1] 55
```

Do you see the distinction between these two operations? The first one evaluates as TRUE or FALSE whereas the second evaluates as numerical values of *y*.

If

Suppose that you wanted to replace all of the negative values in an array by zeros. In the old days, you might have written something like this;

```
for (i in 1:length(y)) { if(y[i] < 0 ) y[i] <- 0 }
```

Now, however, you would use logical subscripts like this

```
y[ y< 0 ] <- 0
```

Ifelse

Sometimes you want to do one thing if a condition is true and a different thing if the condition is false (rather than do nothing, as in the last example). The **ifelse** function allows you to do this for entire vectors without using **for** loops. We might want to replace any negative numbers by -1 and any positive values and zero by $+1$:

```
y <- ifelse( y < 0 , -1, 1 )
```

Trimming vectors using [minus]

Individual subscripts are referred to in square brackets. So if x is like this:

```
x <- c(5,8,6,7,1,5,3)
```

we can find the 4th element of the vector just by typing

```
x[4]
```

```
[1] 7
```

An extremely useful facility is to use negative subscripts. These drop terms from a vector. Suppose we wanted a new vector, y , to contain everything but the first element of x

```
y <- x[-1]
```

```
y
```

```
[1] 8 6 7 1 5 3
```

This facility allows some extremely useful things to be done. Suppose we want to calculate a trimmed mean of x which ignores both the smallest and largest values (i.e. we want to leave out the 1 and the 8). There are two steps to this. First we **sort** the vector x . Then we remove the first element using $x[-1]$ and the last using $x[-length(x)]$. We can do both drops at the same time by concatenating both instructions like this: $-c(1,length(x))$. Then we use the built-in function `mean`.

```
trim.mean <- function (x) mean(sort(x)[-c(1,length(x))])
```

Now try it out. The answer should be $\text{mean}(5,6,7,5,3) = 26/5 = 5.2$

```
trim.mean(x)
```

```
[1] 5.2
```

Summary of vectors

The rule is this. If you *can* use vectorized functions then *do*. Loops should be a last resort. You need to use them when you do something different to each element of an object, but when you do the same thing to each element, use subscripts or built in vector functions. It is a good idea to go through Table 2 and make sure that you understand when you might want to use each of the vector functions listed there.

Plotting mathematical functions

It is easy to plot mathematical functions in order to see how their shape depends on the values of their parameters. You need to know the equation relating y to x and you need to generate a set of values for the x variable. Suppose we want to draw the following asymptotic exponential function:

$$y = a(1 - e^{-bx})$$

for the particular parameter values $a = 3$ and $b = 0.1$. First we need to generate a vector of x values (between 30 and 50 values is all you need for producing smooth-looking curves). You need to know the range of x values. In this case we want the x axis to go from 0 to 50. To generate the x values all we do is this:

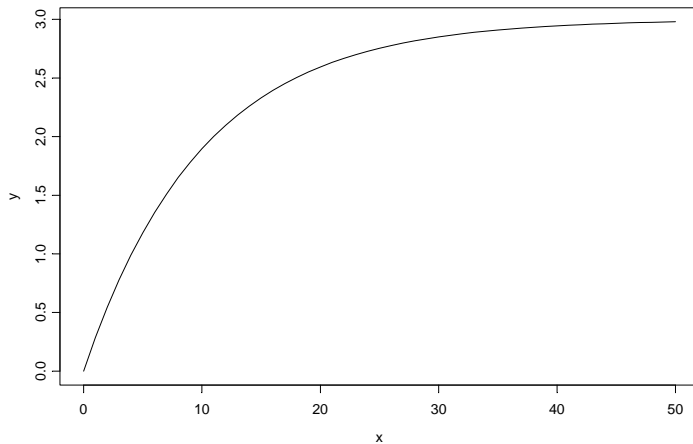
```
x<-0:50
```

This says x gets the values 0 to 50 in steps of 1. Now the values of y are obtained by evaluating the equation for every value of x

```
y<-3*(1-exp(-0.1*x))
```

If we were to say `plot(x,y)` we would get a scatterplot, rather than a smooth function. To get the desired result we just change the plot **type** to "l" which stands for "line" (note that the symbol is a lower case L and not a numeric 1):

```
plot(x,y,type="l")
```



Another thing we might want to do is to plot a family of curves showing their behaviour with different parameter values. Let us plot a family of 2-parameter logistic curves

$$y = \frac{e^{a+bx}}{1 + e^{a+bx}}$$

for different values of b . It is worth working out the extreme values of y if we are going to produce multiple plots, because we do not want subsequent plots to have y values that are out of range when compared to the first plot. Try putting $x = -\infty$. The numerator is $\exp(-\infty)$ which is zero. The denominator is $1+0 = 1$. So y is 0 when x is minus infinity. What about x is plus infinity. Now both the numerator and the denominator are plus infinity, so y should be 1. We shall scale the y axis to go from 0 to 1 in our first plot directive. We shall use the same range of x values for each plot (-5 to 5) so we do not need to make any adjustments to **xlim**. Because we want to have an increment of 0.1 in the x values (rather than 1 as in the last example), we use **seq** to generate our sequence of x values:

```
x<-seq(-5,5,.1)
```

The first plot we want to see has $a = 0.1$ and $b = 0.4$. So we calculate the y values

```
y<-exp(.1+.4*x)/(1+exp(.1+.4*x))
```

Now we make the first plot, bearing in mind 2 things: we need to make the plot **type** a line, and we need to scale the y axis from 0 to 1 using **ylim**

```
plot(x,y,type="l",ylim=c(0,1))
```

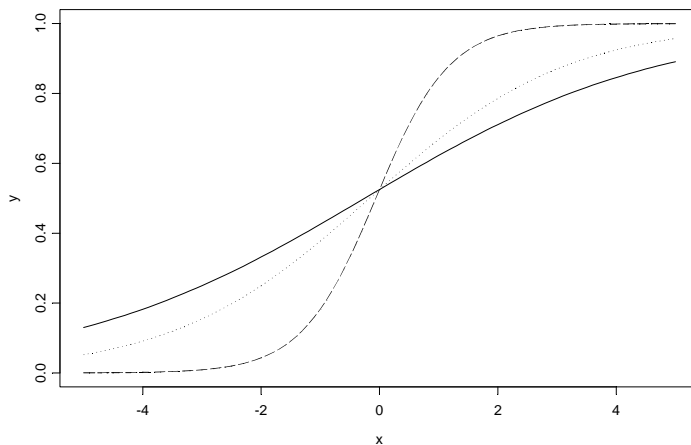
On these same axes we now want to overlay another plot (y_2) with $a = 0.1$ and $b = 0.6$. Use Up Arrow to edit the new values into the original equation:

```
y2<-exp(.1+.6*x)/(1+exp(.1+.6*x))
```

Overlaying new functions is done with the lines directive; we use a different line type **lty**

```
lines(x,y2,lty=2)
```

The final curve (y3) has $a = 0.1$ and $b = 1.6$



```
y3<-exp(.1+1.6*x)/(1+exp(.1+1.6*x))  
lines(x,y3,lty=4)
```

Matrices

Matrix arithmetic is handled in an intuitively obvious way. A matrix is defined with the **matrix** directive. Suppose we want to turn a new vector y of length 15

```
y<-c(8, 3, 5, 7, 6, 6, 8, 9, 2, 3, 9, 4, 10, 4, 11)
```

```
y
```

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

into a matrix called m consisting of 5 rows and 3 columns

```
m<-matrix(y,nrow=5)
```

```
m
```

```

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

```

Another way to make matrices is to **bind** vectors together into matrices: row by row using **rbind**, or column by column using **cbind**. We'll meet this in a later practical.

Matrix arithmetic

It is important to understand that the `*` operator does **not** carry out matrix multiplication. This requires the `%%` operator. Consider this example where a Leslie matrix, `L`, is to be multiplied by a column matrix of population sizes, `n`

```
L<-c(0,0.7,0,0,6,0,0.5,0,3,0,0,0.3,1,0,0,0)
```

```
L<-matrix(L,nrow=4)
```

Note that the elements of the matrix are entered in column-wise, not row-wise sequence. We make sure that the Leslie matrix is properly conformed:

`L`

```

      [,1] [,2] [,3] [,4]
[1,]  0.0  6.0  3.0  1
[2,]  0.7  0.0  0.0  0
[3,]  0.0  0.5  0.0  0
[4,]  0.0  0.0  0.3  0

```

The top row contains the age-specific fecundities, and the sub-diagonal contains the survivorships. Now the population sizes at each age go in a column vector, `n`.

```
n<-c(45,20,17,3)
```

```
n<-matrix(n,ncol=1)
```

`n`

```

      [,1]
[1,]   45
[2,]   20
[3,]   17
[4,]    3

```

Population sizes next year in each of the 4 age classes are obtained by matrix multiplication, `%%`

```
L %**% n
```

```
      [,1]  
[1,] 174.0  
[2,]  31.5  
[3,]  10.0  
[4,]   5.1
```

We can check this out longhand. The number of juveniles next year (the 1st element of n) is the sum of all the babies born last year:

```
45*0+20*6+17*3+3*1
```

```
[1] 174
```

So that's OK. If you try to do ordinary multiplication with L and n you will fail because their dimensions do not match:

```
L*n
```

```
Error in L * n : non-conformable arrays
```

Matrix multiplication %**% on vectors of the same length returns the *vector dot product* (the sum of the pairwise products) as a 1x1 matrix.

Solving systems of linear equations

Suppose we have 2 equations containing 2 unknown variables:

$$3x + 4y = 12$$

$$x + 2y = 8$$

We can use the function **solve** to find the values of the variables if we provide it with 2 matrices:

- a square matrix **A** containing the coefficients
- a column vector **kv** containing the known values

We set the two matrices up like this (column-wise)

```
A<-matrix(c(3,1,4,2),nrow=2)
```

A

```
      [,1] [,2]
[1,]    3    4
[2,]    1    2
```

```
kv<-matrix(c(12,8),nrow=2)
```

kv

```
      [,1]
[1,]   12
[2,]    8
```

Now we can solve the simultaneous equations

```
solve(A,kv)
```

```
      [,1]
[1,]   -4
[2,]    6
```

so $x = -4$ and $y = 6$ (as you can easily verify by hand). The function comes into its own when there are many simultaneous equations to be solved.

Table 3. Matrix operations

Operation	Meaning
<code>solve(x)</code>	inverse of matrix x
<code>solve(x,y)</code>	solution of simultaneous linear equations with coefficients x and known values y
<code>backsolve(x,y)</code>	solves a system of linear equations when the square matrix x is upper triangular ,y is the matrix containing the right-hand sides to equations (the known values)
<code>eigen(x)</code>	eigenvalues and eigenvectors of a square matrix x
<code>diag(x)</code>	diagonal of the matrix x
<code>sum(diag(x))</code>	trace of square matrix x
<code>prod(eigen(x)\$values)</code>	determinant of real-valued matrix x
<code>svd(x)</code>	a list containing the singular value decomposition of x
<code>qr(x)</code>	a representation of an orthogonal (or unitary) matrix and a triangular matrix whose product is the input x
<code>chol(x)</code>	an upper-triangular matrix which is the Choleski decomposition of a (hermitian) symmetric, positive definite (or positive semi-definite) matrix
<code>kron(x,y)</code>	A matrix with dimension the product of the dimensions of the input matrices x & y; each element of x is replaced by that element times the entire matrix y
<code>t(x)</code>	transpose of the matrix x