Appendix 1: Fundamentals of the R language

R as a Calculator

Immediately to the right of the prompt symbol ‘ > ’ on the command line is space in which you can perform a wide range of calculations. The arithmetic operators for addition, subtraction and division are +, – and / respectively, while * means multiply (×) and ^ means ‘to the power’. The first operations to be carried out are powers, then multiplication and division, and finally addition and subtraction. You can overide this hierarchy of calculation by the use of brackets, so to calculate the cube root of 17 × 0.35 you would type

```
(17 * 0.35) ^ (1/3)
```

```
[ 1] 1.812059
```

There is a huge range of mathematical functions; the ones you will use most often are log (logarithm to base e), exp (natural antilog) and sqrt (square root).

```
log(10)
```

```
[ 1] 2.302585
```

```
exp(1)
```

```
[ 1] 2.718282
```

Negative powers are reciprocals, so x^{-1} is the same as $1/x$:

```
3^-1
```

```
[ 1] 0.3333333
```

```
1/3
[ 1] 0.3333333
```

© 2005 John Wiley & Sons, Ltd ISBNs: 0-470-02298-1 (PBK); 0-470-02297-3 (PPC)

Assigning Values to Variables

Variables are assigned values in R using 'gets' <− rather than the more familiar 'equals' = sign. Gets is a composite operator made up of a 'less than' symbol < and a minus sign −. To assign one value to a variable (creating a scalar) called *x* just write

```
x<-12.6
```

Now, whenever we use the variable *x* the value 12.6 is used in its place (until we change the value of *x* with another assignment). More usually in R we work with variables that contain many values (vectors). These can be assigned values in several different of ways. The simplest is to write down all the values separated by commas, and turn this into a vector using the **concatenate** function, c, like this:

```
y<-c(3,7,9,11)
```

If the vector of numbers was long, this would be tedious to type and difficult to proof read. Alternatively, you can type the numbers in at the keyboard during an R session, using the scan function like this:

```
z<-scan()
```

```
1: 8
2: 4
3: 7
4: 5
5:
Read 4 items
```

After typing each number (starting with 8 in this case), press the Return key, then number 4, Return key, and so on. To finish, type two successive Return keys. R responds by telling you how many values you have entered.

If the numbers you want to put into a vector form a regular sequence of some sort, then you can automate the procedure. Suppose you want the integer (whole) numbers 1 to 6 in a vector called *a*. You just type

```
a<-1:6
```

and R understands the colon operator : to mean 'a series of integers between'. Alternatively, your series might be in non-integer steps (say, in steps of 0.1) in which case you use the seq function. For decreasing series you specify negative values of the step size. The vector *b* contains six numbers stepped down from one half to zero:

```
b<-seq(0.5,0,-0.1)
```

Generating Repeats

The rep function replicates the object which is its first argument by the number of times specified in the second argument. Thus

```
rep("A",10)
[ 1] "A"  "A"  "A"  "A"  "A" "A" "A" "A"  "A"  "A"
```

produces ten copies of the character 'A'. The object to be repeated might be a series:

```
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. If we want the **elements** of a series to be repeated (rather than the series as a whole), then the second argument of the rep function needs to be a vector of the same length as the first argument. That sounds complicated, but suppose we wanted three 1's, then three 2's and so on up to three 6's, we would put

```
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
```

The most complex case arises when we want to repeat each element of the first vector a different number of times. One 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  5   6  6  6 6  6  6
```

but more generally, we would like to be able to specify each repeat separately. Here, the elements of the first vector c(4,7,1,5) are repeated by a number of times contained in the second vector c(3,2,5,2) (the two vectors must be the same length)

```
rep(c(4,7,1,5),c(3,2,5,2))
[ 1]  4  4  4   7   7   1   1   1   1   1    5   5
```

Generating Factor Levels

The function gl is very useful for generating levels of factors automatically. The arguments of the function are:

- 'up to', and
- 'with repeats of'.

Suppose you want to generate factor levels up to 5 with repeats of 3 you write

```
gl(5,3)
```

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

By default this pattern is executed just once. If you want repeats of the whole pattern, you specify the total length of the object as the optional third argument. To get two repeats (i.e. total length = 30) you write

```
gl(5,3,30)
```

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

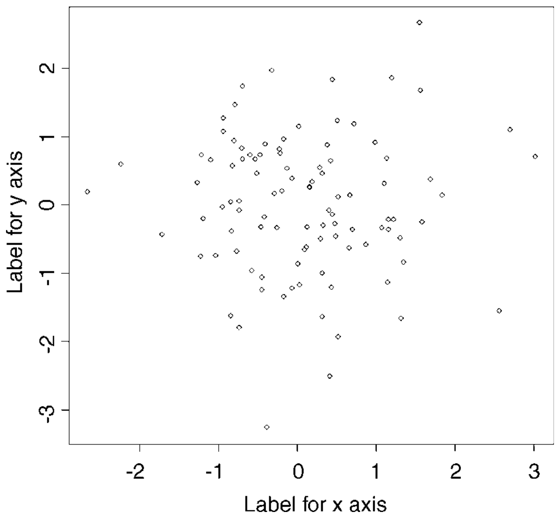
Usefully, the function gl automatically declares the vector to be a factor. You can see this from:

```
is.factor(gl(5,3,30))
```

```
[ 1] TRUE
```

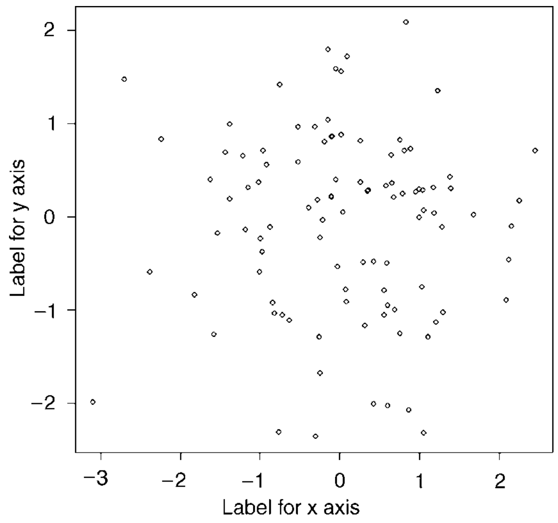
Changing the Look of Graphics

The most likely changes required of R graphics are in the orientation and size of the labels for the *x* and *y* axes. Here are the defaults:



Many journals require that the *y* axis figures are vertically aligned like those on the *x* axis, rather than at right angles to the *y* axis. This is achieved by las = 1. If you want to make the text of the axis labels bigger (increase the font size), specify cex ('character expansion') for the labels, using cex.lab = 1.5

```
plot(rnorm(100),rnorm(100),ylab="Label for y axis",
                                xlab="Label for x axis",las=1,cex.lab=1.5)
```



To see the full range of changes that you could make to graphics if you wanted to, it is worth spending a while browsing the help window on graphics parameters by typing

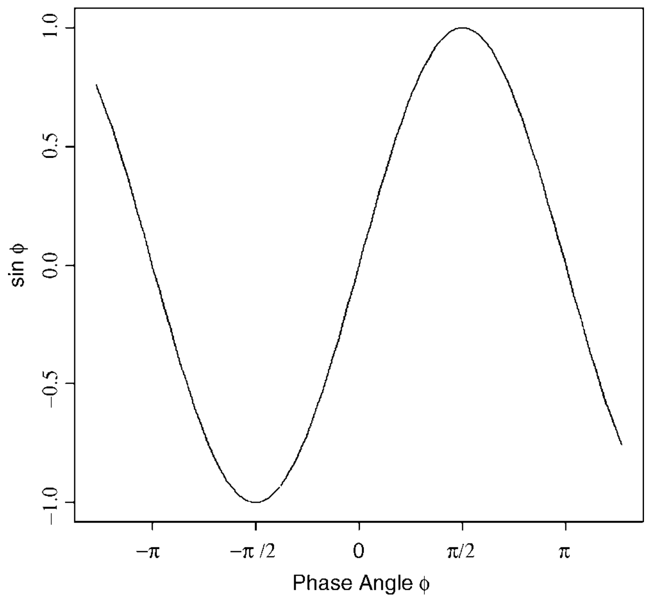
```
?par
```

You will be impressed by how much is being done for you, automatically, behind the scenes when you say plot(x,y).

To write on plots using more intricate mathematical symbols or Greek letters we use expression or substitute. Here are some examples of their use. First, we produce a plot of $\sin\phi$ against the phase angle ϕ on the range $-\pi$ to $+\pi$ radians:

```
x <- seq(–4, 4, len = 101)
plot(x,sin(x),type="l",xaxt="n",
   xlab=expression(paste("Phase Angle ",phi)),
   ylab=expression("sin "*phi))
axis(1, at = c(-pi, -pi/2, 0, pi/2, pi),
lab = expression(-pi, -pi/2, 0, pi/2, pi))
```

Note the use of xaxt = "n" to suppress the default labelling of the x axis, and the use of expression in the labels for the x and y axes to obtain mathematical symbols like phi or pi. The more intricate labels on the x axis are obtained by the axis function, specifying 1 (the x axis is the first axis), then using the at function to say where the labels and tick marks are to appear, and lab with expression to say what the labels are to be.



Suppose you wanted to add '$\chi^2 = 24.5$' to this graph at location $(-\pi/2, 0.5)$. You use text with substitute, like this:

```
text(-pi/2,0.5,substitute(chi^2=="24.5"))
```

Note the use of 'double equals' to print a single equals sign.

You can write quite complicated formulae on plots using paste to join together the elements of an equation: here is the density function of the Normal written on the plot at location $(\pi/2, -0.5)$:

```
text(pi/2, -0.5, expression(paste(frac(1, sigma*sqrt(2*pi)), " ",
        e^{frac(-(x-mu)^2, 2*sigma^2)})))
```

Note the use of frac to obtain individual fractions; the first argument is the text for the numerator, the second the text for the denominator. Most of the arithmetic operators have obvious formats (+ , –, /, *, ^, etc.); the only non-intuitive symbol that is commonly used is 'plus or minus', $\pm$; this is written as %+–% like this:

```
text(pi/2,0,expression(hat(y) %+-% se))
```

Reading Data From a File

For real applications, the typical way to get numbers into a variable in R is to read them from a file that you created and error-checked earlier, perhaps in Excel. The vector may contain many thousands of numbers, and it would make no sense at all to type the numbers directly into R via the keyboard using scan. The R function that reads data from a file is called read.table and you use it like this:

```
dataframe<-read.table("c:\\directory\\filename.txt",header=T)
```

Note that the drive, the directory and the file name are enclosed in double quotes. The phrase header=T says that row 1 of your file contains the variable name(s). You must use 'double backslash' \\ to separate the drive, directory and file names.

Common things that can go wrong at this stage include:

- the file does not exist (e.g. you have mis-spelled the file name, or given it the wrong extension, `.prn` instead of `.txt` for instance);
- the file is not in the directory you specified, but somewhere else;
- the variable names in row 1 of your file have blank spaces between words in some of them (e.g. 'dry weight' or 'body mass') – in such a case, R would look for four columns of numbers ('dry', 'weight', 'body' and 'mass'), not two as intended and the solution is to use a dot instead of a blank in variable names (e.g. 'dry.weight' or 'body.mass');
- you have used blanks in the file to represent missing values (you should use NA).

To use variables contained within the dataframe, you need to use attach like this:

```
attach(dataframe)
```

and to see the names of the variables, and the order in which they appear use

```
names(dataframe)
```

When you have finished using a set of variables, tidy up by detaching the dataframe and removing (rm) any variables names that you have assigned using 'gets':

```
detach(dataframe)
rm(a,b,x,y,z)
```

Vector Functions in R

```
y<-c(5,7,7,8,2,5,6,6,7,5,8,3,4)
z<-13:1
```

Typical operations on vectors include summary statistics, using functions like mean, var, range, max, min, summary, IQR and fivenum.

```
mean(y)
```

```
[1] 5.615385
```

```
var(y)
```

```
[1] 3.423077
```

```
range(y)
```

```
[ 1] 2   8
```

Some functions (like range) produce more than one number as output (min and max in this case). A very powerful feature is that you can do arithmetic with entire vectors. The * operator performs vector multiplication. Since *y* and *z* are the same length, y*z gives a vector of the same length as *y* containing the point-wise products (5×13, 7×12, 7×11, etc.):

```
y*z
```

```
[ 1] 65  84   77   80   18   40   42   36   35   20   24   6   4
```

In R, if two vectors are not the same length, then the shorter vector is repeated as necessary, up to the length of the longer vector. You can see this with the expression

```
y*6
```

```
[ 1]  30  42   42   48  12   30   36   36   42   30   48  18   24
```

in which the scalar, 6, is repeated 13 times to match the length of *y*. Two vectors are joined together, the top of the second vector to the bottom of the first, using the concatenate function, c:

```
c(y,z)
```

```
[ 1]   5  7  7  8  2  5  6  6  7  5  8  3  4  13  12  11  10  9  8  7  6  5  4  3  2
[ 26]  1
```

Subscripts: Obtaining Parts of Vectors

Elements of vectors are addressed by subscripts which appear in square brackets []. The third element of *y* is extracted like this:

```
y[3]
```

```
[ 1]  7
```

the third to the seventh elements of *y* in sequence like this

```
y[3:7]
```

```
[ 1]  7   8   2   5    6
```

and the third, fifth, sixth and ninth elements of *y* like this

```
y[c(3,5,6,9)]
```

```
[ 1]  7  2  5  7
```

To drop an element from an array, you use negative subscripts. Here is the vector *y* without its first element

```
y[-1]
[1] 7  7  8  2  5  6  6  7  5  8 3 4
```

and here is a general way of dropping the last element of the array, without knowing in advance how long the array might be

```
y[-length(y)]
[1] 5  7  7  8  2  5  6  6  7  5 8 3
```

Subscripts as Logical Variables

Often you want to use some kind of logical condition to find a subset of the values in a vector. Suppose, for instance, that we wanted to know all the values of *y* that were bigger than 6? This could not be simpler in R: just state the logical condition as the subscript:

```
y[y > 6]
[1] 7  7  8  7  8
```

We might want to know the values of *z* for which $y > 6$. Again, this could not be simpler:

```
z[y > 6]
[1] 12  11  10  5  3
```

Suppose we wanted to extract all of the elements of *y* that were **not multiples of three**. If a number is a multiple of three then y%%3 ('y modulo 3') will be zero. The symbol for 'not equal' in R is != (exclamation, equals), so the way to extract the non-multiples of three is

```
y[y%%3! = 0]
[1] 5  7  7  8  2  5  7  5  8  4
```

and you see that all the threes and sixes have been removed from *y*.

Subscripts with Arrays

Begin by making a three-dimensional array containing the numbers 1 to 30, structured so that there are five rows and three columns in each of two tables. The first dimension refers to the number of rows, the second to the number of columns, and the third to the number of two-dimensional tables. Note that the numbers enter each table column-wise (rather than row-wise), and that the elements of the array are filled up through the dimensions of the array from left to right (rows then columns then tables):

```
A <- array(1:30, c(5,3,2) )
A
```

```
, , 1

     [,1] [,2] [,3]
[1,]    1    6   11
[2,]    2    7   12
[3,]    3    8   13
[4,]    4    9   14
[5,]    5   10   15

, , 2

     [,1] [,2] [,3]
[1,]   16   21   26
[2,]   17   22   27
[3,]   18   23   28
[4,]   19   24   29
[5,]   20   25   30
```

You might want to select only the second and third columns of A. Columns are the second (middle) subscript, so the first and third subscripts are left blank

```
A[,2:3,]
```

```
, , 1

     [,1] [,2]
[1,]    6   11
[2,]    7   12
[3,]    8   13
[4,]    9   14
[5,]   10   15

, , 2

     [,1] [,2]
[1,]   21   26
[2,]   22   27
[3,]   23   28
[4,]   24   29
[5,]   25   30
```

In another application you might want to take rows two to four of this reduced array (but keep both tables, so the last subscript is blank):

```
A[2:4,2:3,]
```

```
, , 1

      [ ,1]  [ ,2]
[ 1,]    7    12
[ 2,]    8    13
[ 3,]    9    14

, , 2

       [ ,1]   [ ,2]
[ 1,]    22     27
[ 2,]    23     28
[ 3,]    24     29
```

Finally, you might want only the second of the two tables of this reduced array, so all three subscripts are specified

```
A[2:4,2:3,2]
```

```
       [ ,1]   [ ,2]
[ 1,]    22     27
[ 2,]    23     28
[ 3,]    24     29
```

Subscripts with Lists

Vectors are subscripted like this [3], but lists are subscripted like this [[3]]. Understanding the distinction takes a good deal of practice. Here is a list called cars, with three elements to the list: make, capacity and colour

```
cars < -list(c("Toyota","Nissan","Honda"),
  c(1500,1800,1750),c("blue","red","black","silver"))
cars
```

```
[[ 1]]
[ 1] "Toyota"  "Nissan"  "Honda"

[[ 2]]
[ 1] 1500  1800  1750

[[ 3]]
[ 1] "blue"  "red"  "black"  "silver"
```

You need to understand the difference between cars[[3]] and cars[3]

```
cars[[3]]
```

```
[ 1] "blue"  "red"   "black"  "silver"
```

```
cars[3]
```

```
[[ 1]]
[ 1] "blue"  "red"  "black"  "silver"
```

The distinction is apparently rather subtle, but it is very important when you try to extract one element of the sub-list using subscripts (e.g. suppose we want to extract the third colour, red): double brackets works

```
cars[[3]][2]
```

```
[ 1]  "red"
```

but single brackets does not

```
cars[3][2]
```

```
[[ 1]]
NULL
```

Writing Functions in R

One of the outstanding features of R is the ease with which you can write your own functions. Here is a function to produce a summary of various measures of central tendency: we want to print the median, the arithmetic mean, the geometric mean and the harmonic mean of the numbers in a vector called *x*. Let's call the function central and define it like this:

```
central <- function (x) {
gm<-exp(mean(log(x)))
hm<-1/mean(1/x)
cat("Median", median(x),"\n")
cat("Arithmetic mean",mean(x),"\n")
cat("Geometric mean",gm,"\n")
cat("Harmonic mean",hm,"\n")  }
```

Functions are created using function and the code is contained within 'curly brackets', {}. Lines of code are separated with a 'hard return'. There are no built-in functions for geometric mean or harmonic mean, so we have to write our own code on lines two and three. The rest of the code produces nicely formatted output. The function to do this is cat (it is like print but with control over format). The key point here is that to get a new line for the next bit of output, you must include "\n" at the end of your cat function. Now we can run the function to compare the different measures of central tendency for the data in *y*:

```
central(y)
```

```
Median                     6
Arithmetic mean            5.615385
Geometric mean             5.261941
Harmonic mean              4.823322
```

Sorting and Ordering

It is important to understand the distinction between sorting and ordering. Typically you have several variables in a dataframe, including the response variable and the various explanatory variables. In such a case, it is very dangerous to sort any one of the variables on its own, because it becomes uncoupled from its associated explanatory variables. This can cause terrible problems if statistical modelling is subsequently carried out, because values of the response variable will be associated with the wrong values of the explanatory variables. The answer is never to use sort on variables that are part of a dataframe.

```
sort(y)
```

```
[ 1]  2   3   4   5   5   5   6   6   7   7   7   8   8
```

produces the intuitively obvious output, as does

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

```
[ 1]  8   8   7   7   7   6   6   5   5   5   4   3   2
```

The problems can arise if you say y < -sort(y) because there is no 'unsort' function. It is much better practice to leave the variables in your dataframe in their original unsorted sequence and to use order to produce new sequences, because this function leaves the original order undisturbed. Let's see it in action

```
order(y)
```

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

Now you will need to concentrate. What does the number 5 in the first element of order(y) mean? The thing you need to realize is that order (y) does **not** produce **values of** *y*. It produces **subscripts for** *y*. In particular, it produces the subscripts necessary to order the values of *y* into an increasing sequence. So the 5 in position 1 is a subscript – it says that the smallest value in *y* is the value in the fifth element of *y*. Let's see if that is correct:

```
y
```

```
[ 1]  5   7   7   8   2   5   6   6   7   5   8   3   4
```

Yes, it is. The smallest value in *y* is 2, and this is the fifth number in *y*. By the same logic, the eleventh value in *y* should be the largest, because 11 is the last number in order (y). This, too, is correct: there is a tie for highest number because there are two 8's, one in position 4 and one in position 11. Here are the values of *z* ordered by the matching elements of *y*

```
z[order(y)]
```

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

The fifth element of *z* is 9, then the twelfth element is 2, the thirteenth element is 1, the first is 13 and so on. The great advantage of order over sort is that it can be applied to whole dataframes, as illustrated on p. 20.

Counting Elements Within Arrays

You often want to know how many times particular values appear in a vector. The function for this is table. Here we generate 10 000 random numbers from a negative binomial distribution with mean = 1.2 and aggregation parameter $k = 0.63$ (prob = $k/(\mathrm{mu} + k)$. The question is: how many zeros are there amongst the 10 000 numbers?

```
vals < -rnbinom(10000,size = 0.63,prob = 0.63/1.83)
table(vals)
```

```
vals
   0      1      2    3     4     5    6    7    8    9  10   11   12   13   14  15+
5048   2169   1116  651   379   230  156   97   59   31  29   13    9    5    3   5
```

The answer is 5048 zeros (about half of the numbers). You will get a different figure each time you execute the rnbinom function because the randomizations will be different.

Tables of Summary Statistics

One of the most commonly used functions in R is tapply. This is the function by which tables of means, variances, sample sizes and suchlike are produced. We need a big dataframe to see this in full swing:

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

```
[ 1] "Growth.rate"  "Water"  "Detergent"  "Daphnia"
```

There is a response variable (Growth.rate) and three categorical explanatory variables (Water, Detergent and Daphnia clone). The first argument of tapply is the variable to be summarized, the second argument is the variable by which the summary is to be classified, and the third argument is the function to be applied (mean, variance or whatever). Here is the use of tapply for means classified by the levels of a single categorical variable:

```
tapply(Growth.rate,Detergent,mean)
```

```
  BrandA       BrandB       BrandC       BrandD
3.884832     4.010044     3.954512     3.558231
```

```
tapply(Growth.rate,Water,mean)
```

```
    Tyne         Wear
3.685862     4.017948
```

When you want a two-way (or higher) classification, then the two (or more) classifying variables appear in a list as the second argument: the levels of the first variable create the rows (Water) and the levels of the second variable create the columns (Detergent) of the summary table:

```
tapply(Growth.rate,list(Water,Detergent),mean)
```

```
         BrandA     BrandB     BrandC     BrandD
Tyne   3.661807   3.911116   3.814321   3.356203
Wear   4.107857   4.108972   4.094704   3.760259
```

To check that the replication is equal for each combination of factors, you can use the length function as the third argument:

```
tapply(Growth.rate,list(Water,Detergent),length)
```

```
        BrandA      BrandB      BrandC      BrandD
Tyne         9           9           9           9
Wear         9           9           9           9
```

Yes, all combinations are based on nine numbers.

Converting Continuous Variables into Categorical Variables Using cut

It might be that you want to reduce a continuous variable into a categorical variable with a small number of levels (like small, medium and large). The cut function makes this very straightforward

```
sml<-cut(vals,3)
table(sml)
```

```
sml
(-0.017,5.66]  (5.66,11.3]   (11.3,17]
         9593          385          22
```

Here we take the vector of 10 000 negative binomial random numbers (p. 294) and use cut with table to see how many of them were small, medium or large (the elements of `sml` are produced by cutting `vals` into three equal parts based on the range of values in `vals`). There were only 22 individuals in the large category which ran from “[11.3” (meaning 11.3 and greater–‘from and including’), to “17”] (meaning less than 17.0–‘up to but not including’). Alternatively, instead of specifying the number of bits for cutting (three in this case), you can specify where you want the break points to be.

The split Function

The function called split produces a list of vectors on the basis of the levels of a factor, and is particularly useful in generating plots:

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

```
[1] "xc" "yc" "fac"
```

The idea is to create scatterplots with different symbols for fac = "A" and fac = "B". Start by producing the blank

```
plot(xc,yc,type="n",xlab="x",ylab="y")
```

Now create new vectors for the *x* and *y* axes split on the basis of fac = "A" or "B":

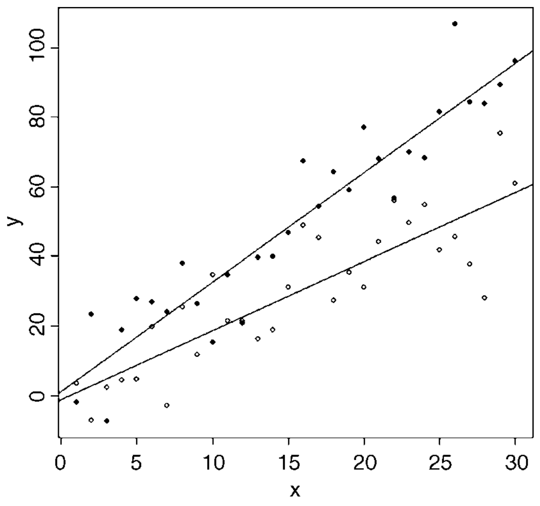
```
sxc<-split(xc,fac)
syc<-split(yc,fac)
```

add the points with different plotting symbols

```
points(sxc[[1]],syc[[1]])
points(sxc[[2]],syc[[2]],pch=16)
```

then add the regression lines for each factor level separately:

```
for (i in 1:2) abline(lm(syc[[i]]~sxc[[i]]))
```

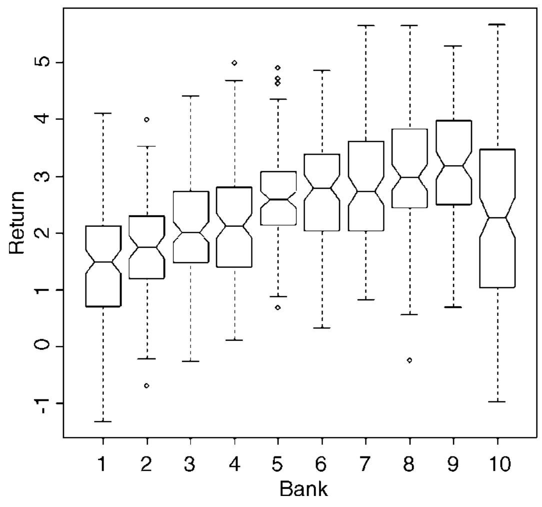


```
forms<-read.table("c:\\temp\\results.txt",header=T)
attach(forms)
names(forms)
```

```
[ 1] "return"  "bank"
```

The idea is to use split to produce a set of box and whisker plots, one for each bank

```
boxplot(split(return,bank),notch=T,ylab="Return",xlab="Bank")
```



The notch = T option allows significance testing of the difference in median return between banks: those where the notches do not overlap are significantly different at 5% (like 4 and 5) while those where the notches do overlap (like 7 and 8) are not significantly different.

Trellis Plots

These multi-panel plots are particularly useful in the context of grouped data of the kind met in mixed effects modelling. Here are data on repeated measures on the growth of 48 pigs:

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

```
[ 1] "Pig" "t1" "t2" "t3" "t4"  "t5" "t6"  "t7" "t8"  "t9"
```

```
pig.wt<-c(t1,t2,t3,t4,t5,t6,t7,t8,t9)
```

Next we create a vector for pig identity pig.id which is the vector Pig (a vector of numbers 1 to 48) repeated nine times

```
pig.id<-c(rep(Pig,9))
```

Now we generate a vector for the week number: 48 1s then 48 2s etc:

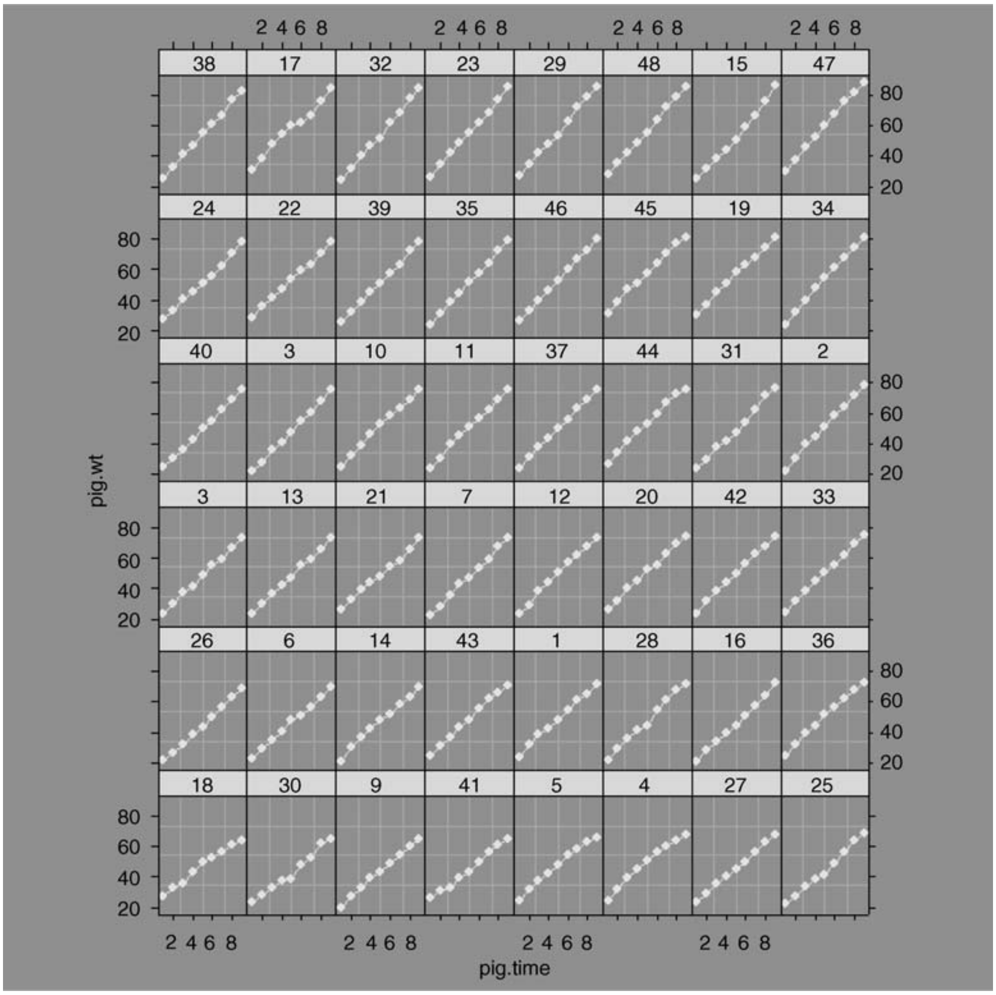
```
pig.time<-c(rep(c(1:9),each=48))
pig<-data.frame(cbind(pig.time,pig.id,pig.wt))
```

Finally convert the dataframe into a grouped data object using groupedData

```
library(nlme)
pig.growth <-groupedData(pig.wt~pig.time|pig.id,data=pig)
```

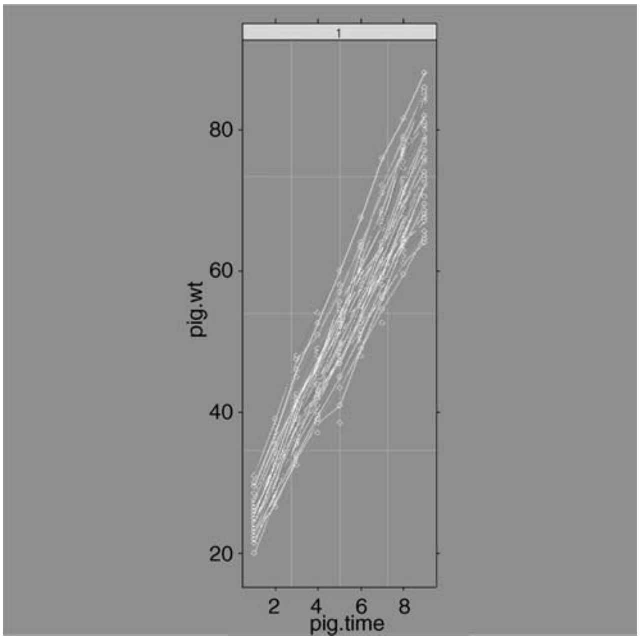
To see a trellis of time series plots of weight for each pig separately, just type:

```
plot(pig.growth,pch=16)
```



whereas to see all the time series in a single axis, set the outer option to ~1 like this:

```
plot(pig.growth,outer=~1,key=F)
```



The lack of crossing of the growth trajectories shows that 'tracking' is high (the individuals that were the biggest at the beginning were the biggest at the end of the trial). Statistical modelling of these data is explained on the web site: http://www.imperial.ac.uk/bio/research/crawley/statistics.

The xyplot Function

Trellis plots are very useful for showing conditioning (the dependence of the response to one variable on the level of another continuous explanatory variable); they are accessible from the library called 'lattice':

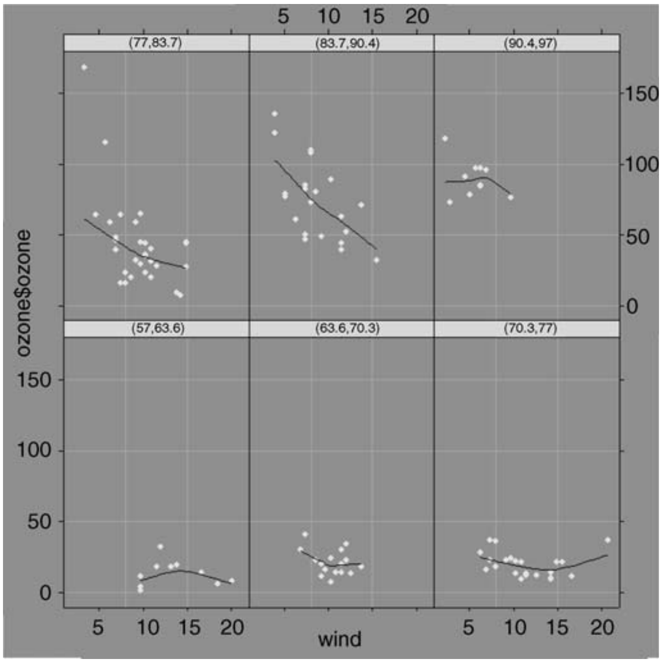
```
library(lattice)

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

```
[ 1] "rad"  "temp"  "wind"  "ozone"
```

We want to look at the graph of ozone against wind-speed at different temperatures, using cut to produce six panels, based on the range of values of temp. The vertical bar, |, is read as 'given': we want to plot ozone against wind-speed, given the temperature.

```
xyplot(ozone$ozone ~ wind|cut(temp,6),
panel = function(x, y) {
```

```
panel.grid(h = -1, v = 2)
panel.xyplot(x, y,pch = 16)
panel.loess(x,y, span = 1) } )
```



You can see that the dependence of ozone concentration on wind-speed is pronounced only for temperatures in the range 77–90°F. Note that when the dataframe has the same name as one of the variables contained within it, you need to extract the variable name using $ like this: ozone$ozone. Use the help function

```
?xyplot
```

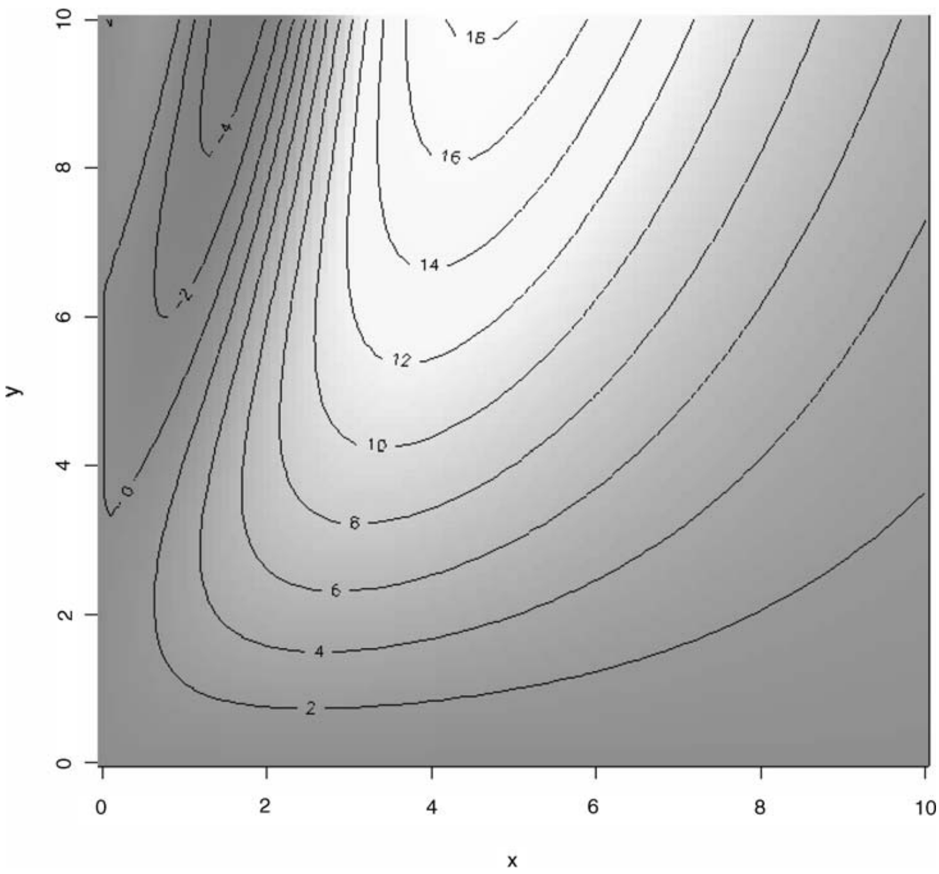
to see all the options available in panel plots and trellis graphics.

Three-dimensional (3-D) Plots

For producing 3-D plots (including image and contour), the outer function is very useful. It evaluates a function (called func in this case) at every combination of *x* and *y* values within a square or rectangular array; this produces data in exactly the form required by image and contour. Note the use of add = T to add the contours on top of the coloured image.

```
x < -seq(0,10,0.1)
y < -seq(0,10,0.1)
```

```
func<-function(x,y) 3*x*exp(0.1*x)*sin(y*exp(-.5*x))
image(x,y,outer(x,y,func))
contour(x,y,outer(x,y,func),add=T)
```



Other palettes for image plots include terrain.colors (greens, shading through yellows and browns into white for high ground) and topo.colors (blue, shading through greens to yellows for high ground) and rainbow (red, orange, yellow, green, blue, indigo, violet). For black and white printers you can specify a grey scale

```
image(x,y,outer(x,y,func),col=palette(gray(seq(0,.9,len=25))))
```

Matrix Arithmetic

It is important to understand that matrix multiplication requires the %*% operator (not *). Consider this example where a Leslie matrix, **L**, is to be multiplied by a column matrix of age-structured 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 (e.g. 2-year-olds produce six female offspring per year), and the sub-diagonal contains the survivorships (70% of 1-year-olds become 2-year-olds). 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 four 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 longhand. The number of juveniles next year (the first element of **n**) is the sum of all the babies born last year:

```
45*0+20*6+17*3+3*1
[1]  174
```

We write a function to carry out the matrix multiplication, giving next year's population vector as a function of this year's:

```
fun<-function(x) L%*%x
```

Now we can simulate the population dynamics over a period long enough (say, 40 generations) for the age structure to approach stability. So long as the population growth rate $\lambda > 1$ the population will increase exponentially, once the age structure has stabilized

```
pop<-numeric(40)
for (i in 1:40) {
  n<-fun(n)
  pop[i]<-sum(n)}
plot(log(pop),type="l")
```

The population growth rate (the per-year multiplication rate, λ) is approximated by the ratio of population sizes in the 40th and 39th years:

```
pop[40]/pop[39]
```

```
[1] 2.164035
```

and the approximate stable age structure is obtained from the 40th value of **n**

```
n/sum(n)
```

```
            [,1]
[1,]   0.709769309
[2,]   0.230139847
[3,]   0.052750539
[4,]   0.007340305
```

The exact values of the population growth rate and the stable age distribution are obtained by matrix algebra: they are the dominant eigenvalue and eigenvector respectively. Use the function eigen applied to the Leslie matrix, **L**, like this

```
eigen(L)
```

```
$values
[1] 2.1694041+0.00000000i -1.9186627+0.00000000i -0.1253707+0.09751046i -0.1253707-
0.09751046i

$vectors
             [,1]           [,2]                   [,3]                   [,4]
[1,]  -0.949264118+0i  -0.93561508+0i  -0.01336028-0.03054433i  -0.01336028+0.03054433i
[2,]  -0.306298338+0i   0.34134741+0i  -0.03616819+0.14241169i  -0.03616819-0.14241169i
[3,]  -0.070595039+0i  -0.08895451+0i   0.36511901-0.28398118i   0.36511901+0.28398118i
[4,]  -0.009762363+0i   0.01390883+0i  -0.87369452+0.00000000i  -0.87369452+0.00000000i
```

The dominant eigenvalue is 2.1694 (compared with our empirical approximation of 2.1640 after 40 years). The stable age distribution is given by the first eigenvector, which we need to turn into proportions

```
eigen(L)$vectors[,1]/sum(eigen(L)$vectors[,1])
```

```
[1]  0.710569659+0i   0.229278977+0i   0.052843768+0i   0.007307597+0i
```

This compares with our approximation (above) in which the proportion in the first age class was 0.70977 after 40 years (rather than 0.71057).

Solving Systems of Linear Equations

Suppose we have two equations containing two 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 two matrices:

- a square matrix **A** containing the coefficients, and
- 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.