Chapter 3

Writing functions in R

3.1 Key ideas

3.1.1 Good programming practice

A program is a set of instructions for a computer to follow. Putting a set of instructions together in a program means that we do not have to rewrite them every time we want to execute them. Programming a computer is a demanding (but potentially rewarding) task. The process is made simpler and the end product more effective by following the simple guidelines set out below.

- **Problem specification**: The starting point for any program should be specification of the problem that we would like to solve. We need to have a clear idea of what we want the program to do before we start trying to write it. In particular, the available inputs (arguments) and the desired output (return value) should be specified.
- **Code planning**: In the early stage of writing a program it is best to stay away from the machine. This is hard to do, but time sketching out a rough version of the program with pen and paper is almost always time well spent. Developing your own pseudo-code, some thing between natural and programming language, often helps this process.
- **Identifying constants**: If you see a number cropping up repeated in the same context, it is good practice to give this value to an identifier at the start of the program. The advantage is one of maintenance; if we need to change the value it can be done with a single alteration to the program.
- **Program documentation**: A good program should be self-documenting. The code should be laid out in a clear and logical fashion; appropriate use of indentation adds enormously to readability of the code. Comments should be used to describe how the more complicated parts of the program work. Writing comments is for your own benefit as much as anyone else's. I write programs that work in a completely obvious way; no need for comments. A couple of weeks later, I waste hours unraveling this 'obvious code'.

Don't follow my example. Just a few simple comments about what the program is for and what each part does will help.

- **Solving runtime problems**: Having carefully crafted and typed in your program, more often than not, the computer will refuse to run it at first time of asking. You may get some vaguely helpful suggestion as to where you have gone wrong. We will not be writing big programs so we do not require very sophisticated debugging tools. If the error message does not lead you straight to the solution of the problem, it is possible to comment out sections of the program to try to isolate the code which is causing the problem.
- **Program verification**: Once the program runs, we need to make sure it does what it is intended to do. It is possible to automate the process of program verification but we will usually be satisfied with checking some cases for which the correct output is known.

3.1.2 Flow control

Computers are very good at performing repetitive tasks. If we want a set of operations to be repeated several times we use what is known as a *loop*. The computer will execute the instructions in the loop a specified number of times or until a specified condition is met. Once the loop is complete, the computer moves on to the section of code immediately following the loop as illustrated below.

program section	`A`
	`loop B`
	`C`
order of execution	`A B B ...B C`

There are three type of loop common to most (C-like) programming languages: the `for` loop, the `while` loop and the `repeat` loop. These types of loop are equivalent in the sense that a loop constructed using one type could also be constructed using either of the other two. Details of the R implementation can be found in sections 3.5.1 and 3.5.3. In general loops are implemented very inefficiently in R; we discuss ways of avoiding loops in section 3.5.4. However, a loop is sometimes the only way to achieve the result we want.

Often there are parts of the program that we only want to execute if certain conditions are met. *Conditional statements* are used to provide branches in computer programs. The simplest structure is common to most programming languages;

`if (`*condition*`)` *ifbranch*

program section	`A`
	`if (`*condition*`) B`
	`D`
order of execution	if the condition is true `A B D`
	if the condition is false `A D`

In this example `B` form the if branch of our program. Another common conditional statement takes the form

```
if (condition) ifbranch else elsebranch
```

program section	`A`
	`if (condition) B`
	`else C`
	`D`

order of execution	if the condition is true `A B D` if the condition is false `A C D`

In R it is also possible to construct vector conditional statements. Conditioning in R is discussed in section 3.5.2

3.1.3 Pseudo-random numbers

Mathematical models fall into two categories, deterministic and stochastic. For a deterministic model, if we know the input, we can determine exactly what the output will be. This is not true for a stochastic model; the output of a stochastic model is a random variable. Remember that we are talking about the properties of models; discussions of the deterministic or stochastic nature of real-life phenomena often involve very muddled thinking and should be left to the philosophers. Simple deterministic models can produce remarkably complex and beautiful behaviour. A example is the logisitic map,

$$x_{n+1} = rx_n(1 - x_n), \tag{3.1}$$

for $n = 1, 2, \ldots$ and starting value $x_0 \in (0, 1)$. For some values of the parameter r, this map will produce a sequence of numbers $x_1, x_2, \ldots$ that look random. Put more precisely, we would not reject a null hypothesis of independence if we were to test these values. We call sequences of this sort *pseudo-random*. Pseudo-random number generators play an important role in the study of model properties using simulation (see chapter 4).

3.2 R essentials

3.2.1 Saving, loading and running R commands

R maintains a record of command history so that we can scroll through previous commands using ↑ and ↓ keys. It may sometimes be useful to save commands for future use. We can save the command history at any stage using

```
> savehistory("filename.Rhistory")
```

The commands entered so far will be saved to file in a plain text format. All of the sessions and my solutions to the exercises are available as `.Rhistory` files on the webpage and public folders. In order to restore saved commands we use

```
> loadhistory("filename.Rhistory")
```

The commands will not be run but will be made available via ↑ key. In order to run a saved set of commands or commands in plain text format from some other source we use

```
> source("filename", echo=TRUE)
```

The echo command tells R to print the results to screen. All of the commands in the file will be executed using this approach. It is also possible to paste commands (that have been copied to the Windows clipboard) at the R command prompt.

3.2.2 Packages

A package in R (library section in S) is a file containing a collection of objects which have some common purpose. For example, the library `MASS` contains objects associated with the Venables and Ripley's *Modern Applied Statistics with S*. We can generate a list of available packages, make the objects in a package available and get a list of the contents of a package using the `library(...)` command.

```
> library()  # lists available packages
> library(MASS)  # loads MASS package
> library(help=MASS)  # lists contents of MASS package
```

If we are specifically interested in the data sets available in a package we can use the `data(...)` command to list them and add them to our workspace.

```
> data()  # lists data sets in loaded packages
> data(package=nlme)  # lists data sets in mixed effects package
> data(Alfalfa, package=nlme)  # adds Alfalfa data set to workspace
> summary(Alfalfa)
```

Information on recommended packages (in the very large R Reference Index) along with a guide to writing R extensions can be found at `http://cran.r-project.org/manuals.html`

3.2.3 The search path

When we pass an identifier to a function R looks for an associated object. The first place R looks is in the workspace (also known as the global environment `.GlobalEnv`). If the object is not found in the workspace, R will search any attached lists (including data frames), packages loaded by the user and automatically loaded packages. The real function of `attach(...)` is now clear; `attach(...)` places a list in the search path and `detach(...)` removes it. The function `search()` returns the search path. Passing the name of any environments or its position to the objects function will list the objects in that environment.

```
> search()
> objects(package:MASS)
> objects(4)
```

3.2.4 Formatting text output

For anything other than the simplest functions, the format of the output that is printed to screen must be carefully managed; a function is of little use if its results are not easy to interpret. The function `cat(...)` concatenates its arguments and prints them as a character string. The function `substring(...)` is use to specify part of a sequence of characters.

```
> str3 <- "ST419 Computational Statistics"
> cat(substring(str3,1,13), substring(str3,26,29), "\n", sep="")
```

Note that `"\n"` is the new line character. The function `format(...)` allows greater flexibility including setting the accuracy to which numerical results are displayed. A combination of `cat(...)` and `format(...)` is often useful.

```
> roots.of.two <- 2^(1/1:10)
> roots.of.two
> format(roots.of.two)
> format(roots.of.two, digits=3)
> cat("Roots of two:  ", format(roots.of.two, digits=3),"\n")
```

3.2.5 Missing and not computable values

We have already encountered the special value `NA` denoting not available. This is used to represent missing values. In general, computations involving `NA`s result in `NA`. In order to get the result we require, it may be necessary to specify how we want missing values to be treated. Many function have named arguments in which the treatment of missing values can be specified. For example, for many statistical functions, using the argument `na.rm = TRUE` instructs R to perform the calculation as if the missing values were not there. R has a representation of infinity (`Inf`). However, the results of some expression are not computable, for example `0/0` or `Inf - Inf`). R uses `NaN` to denote not a number.

† The value `NA` is of mode `"logical"`. Despite the assurances of sections 2.1.1, R can actually handle three valued logic. Try the following, construct the truth tables and try to work out what is going on.

```
> lvec1 <- rep(c(TRUE, FALSE, NA), times=3)
> lvec2 <- rep(c(TRUE, FALSE, NA), each=3)
> lvec1 & lvec2
> lvec1 | lvec2
```

3.2.6 Coercion and class checking (`as` and `is`)

Some functions will only accept objects of a particular mode or class. It is useful to be able to convert (*coerce*) an object from one class to another. The generic function for this purpose is `as(...)`. The first argument is the object that we would like to convert and the second is

the class we would like to convert to. There are a number of *ad hoc* functions of the form `as.`*x* that perform a similar task. Particularly useful are `as.numeric(...)` and `as.vector(...)`. The function `is(...)` and similar `is.`*x*`(...)` functions check that an object is of the specified class or type, for example `is.logical(...)` and `is.numeric(...)`. [There are two functions of the form `is.`*x*`(...)` command that do not fit this pattern: `is.element(...)` checks for membership of a set; `is.na(...)` checks for missing values element by element.]

```
> is.vector(roots.of.two)
> is.character(roots.of.two)
> is(roots.of.two, "character")
> as.character(roots.of.two)
> is.logical(lvec1)
> as(lvec1, "numeric")
> is(as(lvec1, "numeric"), "numeric")
```

3.2.7 Arrays and matrices

An array in R consists of a data vector that provides the contents of the matrix and a dimension vector. A matrix is a two dimensional array. The dimension vector has two elements the number of rows and the number of columns. A matrix can be generated by the `array(...)` function with the `dim` argument set to be a vector of length 2. Alternatively, we can use the `matrix(...)` command or provide a dim attribute for a vector. Notice the direction in which the matrix is filled (columns first)

```
> m1 <- array(1:15,dim = c(3,5))
> m1
> m2 <- matrix(1:15,3,5)
> m2
> m3 <- 1:15
> dim(m3) <- c(3,5)
> m3
> c(class(m1), class(m2), class(m3))
> array(1:15, dim=c(3,6))  # recycling rule used
> array(1:15, dim=c(2,4))  # extra values discarded
```

Using arithmetic operators on similar arrays will result in element by element calculations being performed. This may be what we want for matrix addition. However, it does not correspond to the usual mathematical definition of matrix multiplication.

```
> m1+m1
> m1*m1
```

Conformal matrices (that is, matrices for which the number of columns of the first is equal to the number of rows of the second) can be multiplied using the operator `%*%`. Matrix inversion and (equivalent) solution of linear systems of equations can be performed using the function `solve(...)`. For example, to solve the system $\boldsymbol{ax} = \boldsymbol{b}$, we would use `solve(A, b)`. If a single

matrix argument is passed to `solve(...)`, it will return the inverse of the matrix.

```
> m4 <- array(1:3, c(4,2))
> m5 <- array(3:8, c(2,3))
> m4 %*% m5
> m6 <- array(c(1,3,2,1),c(2,2))
> m6
> v1 <- array(c(1,0), c(2,1))
> solve(m6,v1)
> solve(m6)  # inverts m6
> solve(m6) %*% v1  # does the same as solve(m6,v1)
```

The outer product of two vectors is the matrix generated by looking at every possible combination of their elements. The first two arguments of the `outer(...)` function are the vectors. The third argument is the operation that we would like to perform (the default is multiplication). This is a simplification since outer product works on arrays of general dimension (not just vectors). The following example is taken from Venables *et. al.*'s *An Introduction to R* page. The aim is to generate the mass function for the determinant of a 2×2 matrix whose elements are chosen from a discrete uniform distribution on the set $\{0, \ldots, 5\}$. The determinant is of the form $AD - BC$ where A, B, C and D are uniformly distributed. The mass function can be calculated by enumerating all possible outcomes. It exploits the `table(...)` function that forms a frequency table of its argument (notice how `plot(...)` works for objects of class `"table"`.

```
> m7 <- outer(0:5,0:5)  # every possible value of AD and BC
> freq <- table(outer(m7,m7,"-"))  # frequency for all values of AD-BC
> freq
> plot(freq/sum(freq), xlab="Determinant value", ylab = "Probability")
```

Various other matrix operations are available in R: `t(...)` will take the transpose, `nrow(...)` will give the number of rows and `ncol(...)` the number of columns. The function `rbind(...)` (`cbind(...)`) will bind together the rows (columns) of matrices with the same number of columns (rows). We can get the eigenvectors and eigenvalues of a symmetric matrix using `eigen(...)` and perform singular value decomposition using `svd(...)`. Investigating the use of these commands is left as an exercise.

3.3 Writing your own functions

We have seen in the previous two sections that there are a large number of useful function built into R; these include `mean(...)`, `plot(...)` and `lm(...)`. Explicitly telling the computer to add up all of the values in a vector and then divide by the length every time we wanted to calculate the mean would be extremely tiresome. Fortunately, R provides the `mean(...)` function so we do not have to do long winded calculations. One of the most powerful features of R is that the user can write their own functions. This allows complicated procedures to be built with relative ease.

The general syntax for defining a function is

```
name <- function(arg1, arg2, ...)  expr1
```

The function is called by using

```
name(...)
```

We will discuss the possible forms of the argument list in section 3.6. When the function is called the statements that make up *expr1* are executed. The final line of *expr1* gives the return value.

Consider the logistic map (3.1). The map could be rewritten as follows

$$x_{n+1} = g(x_n),$$

where

$$g(x) = rx(1-x).$$

Clearly, the quadratic function g plays an important role in the map. We can write R code for this function; note that we follow the usual computing convention and give this function the descriptive name `logistic` rather than calling it g.

```
> logistic <- function(r,x) r*x*(1-x)
```

Now that we have defined the function, we can use it to evaluate the logistic function for different values of r and x (including vector values).

```
> logistic(3,0)
[1] 0
> logistic(3,0.4)
[1] 0.72
> logistic(2,0.4)
[1] 0.48
> logistic(3.5, seq(0,1,length=6))
[1] 0.00 0.56 0.84 0.84 0.56 0.00
```

The expression whose statements are executed by a call to `logistic(...)` is just the single line `r*x*(1-x)`. This is also the return value of the function. The expression in a function may run to several lines. In this case the expression is enclosed in curly braces { } and the final line of the expression determines the return value. In the following function we use the `logistic(...)` function that we have just defined.

```
> lmap1 <- function(r)
+ {  temp <- logistic(r, 0.6)
+    logistic(r, temp)
+ }
> lmap1(1)
[1] 0.1824
> lmap1(3)
[1] 0.6048
```

More than one command on a line
Separate commands can be placed on a single line separated by `;` for example, we could have written `temp <- logistic(r, 0.6); logistic(r, temp)`. This is best avoided unless the commands are very short.

This function `lmap1` performs a single iteration in the logistic map. Note that the argument name `r` is just a label. We can name this argument what we like.

```
> lmap1a <- function(turin)
+ {  galileo <- logistic(turin,0.6)
+    logistic(turin,galileo)
+ }
> lmap1a(1)
[1] 0.1824
> lmap1a(3)
[1] 0.6048
```

This is a silly example to prove a point; in practice it is best to use sensible names that you will be able to understand when you come back to look at the code.

We may construct a vector return value for a function.

```
> lmapvec <- function(x)
+ {  ans <- x
+    ans <- c(ans, logistic(3,ans))
+    ans
+ }
> lmapvec(0.2)
[1] 0.20 0.48
> lmapvec(0.48)
[1] 0.4800 0.7488
```

Making changes to a function
We often want to correct or make additions to an existing function. One of the easiest ways to do this is using `fix(...)`. For example, we could edit our `lmapvec(...)` function using `fix(lmapvec)`. This will open an edit window. Make the changes then close the window clicking `Yes` to save any alterations you have made.

3.4 Using side effects

In many instances we are more interested in the side effects of a function than its return value. This type of function is written in exactly the same way as other functions. The only difference

is that we do not need to worry about the final line. [In fact it is polite to return a value that indicates whether the function has been successful or not but this will not concern us for now.]

```
> logisticplot <- function(r,lower,upper,npoints)
+ {  x <- seq(lower,upper,length=npoints)
+    plot(x,logistic(r,x), type = "l")
+ }
> logisticplot(3,0,1,10) # a bit ropey (not enough points)
> logisticplot(3,0,1,100) # better
```

3.5 Flow control

3.5.1 Loops – `for`

A `for` loop often provides the most obvious implementation.

`for (`*loopvariable* `in` *sequence*`)` *expr1*

Here `sequence` is actually any vector expression but usually takes the form of a regular sequence such as `1:5`. The statements of *expr1* are executed for each value of the loop variable in the sequence. An couple of examples follow.

```
> for (i in 1:5) print(i)
```

```
> load("moreRessentials.Rdata")
> attach(mk2nd)
> for (i in 1:length(exam1))
+ {  ans <- exam1[i] + exam2[i] + exam3[i]
+    cat(row.names(mk2nd)[i], " total:  ", ans, "\n")
+ }
```

3.5.2 Conditional statements – `if`

The `if` statement in R follows the standard syntax given in section 3.1.2.

`if (`*condition*`)` *ifbranch*
`if (`*condition*`)` *ifbranch* `else` *elsebranch*

Here the *condition* is an expression that yields a logical value (`TRUE` or `FALSE`) when evaluated. This is typically a simple expression like `x > y` or `dog == cat`. A couple of examples follow to illustrate the difference between `if` and `if - else` statements.

```
> for (i in 1:5)
+ {  cat(i, ":  ")
```

```
+    if (i<3) cat("small")
+    cat(" number \n")
+ }

> for (i in 1:5)
+ {  cat(i, ":  ")
+    if (i<3) cat("small")
+    else cat("big")
+    cat(" number \n")
+ }
```

As a slightly more substantial example, suppose that in order to pass student must achieve a total mark of 60 or higher. We can easily write R code to tell us which students have passed.

```
> for (i in 1:length(exam1))
+ {  ans <- exam1[i] + exam2[i] + exam3[i]
+    cat(row.names(mk2nd)[i], ":  ")
+    if (ans >= 60) cat("PASS \n")
+    else cat("FAIL \n")
+ }
```

In section 3.1.2 we mention the fact that loops are not efficiently implemented in R. One way of avoiding the use of loops is to use commands that operate on whole objects. For example, `ifelse(...)` is a conditional statement that works on whole vectors (rather than requiring a loop to go through the elements).

`ifelse(`*`condition`*`,`*`vec1`*`,`*`vec2`*`)`

If `condition`, `vec1` and `vec2` are vectors of the same length, the return value is a vector whose i^{th} elements if `vec1[i]` if `condition[i]` is true and `vec2[i]` otherwise. If `condition`, `vec1` and `vec2` are of different lengths, the recycling rule is used. We exploit recycling in this version of the PASS/FAIL example above.

```
> pf <- ifelse(exam1+exam2+exam3>=60,"PASS","FAIL")
> pf
```

This is not quite what we want. In the previous example we had candidate numbers associated with the PASS/FAIL results. We will return to this example in section 3.5.4.

3.5.3 More loops – `while` and `repeat` †

We can achieve the same result as a `for` loop using either a `while` or `repeat` loop.

`while (condition)` *`expr`*
`repeat` *`expr`*

A `while` loop continues execution of the expression while the condition holds true. A `repeat`

loop repeated executes the expression until explicitly terminated. We illustrate with the trivial example from section 3.5.1 that just prints out numbers from 1 to 5.

```
> j <- 1
> while (j<6)
+ {  print(j)
+    j <- j+1
+ }

> j <- 1
> repeat
+ {  print(j)
+    j <- j+1
+    if (j>5) break
+ }
```

Note that in these examples we have to explicitly initialise and increment the value of our loop variable `j`. The `break` command can be used in any type of loop to terminate and move execution to the final } of the loop expression. In a `repeat` loop, `break` is the only way to terminate the loop.

A `while` loop can be useful when we have a condition for termination of the loop rather than an exact point. For example, suppose that we want to get the first 10 pass marks from the `mk2nd` data. We do not know exactly how far through the data we must look to find 10 pass marks. We could write a for loop with a break statement but a while loop is neater.

```
> npass <- 0
> j <- 1
> tot <- exam1+exam2+exam3
> while (npass < 10)
+ {  if (tot[j] >= 60)
+    {  npass <- npass+1
+       cat(row.names(mk2nd)[j], ":  ", tot[j], "\n")
+    }
+    j <- j+1
+ }
```

3.5.4 Vectorization and avoiding loops

Loops are an important part of any programming language. However, as we have mentioned before, loops are very inefficiently implemented in R (S is no better). Having learnt about loops, it is now important to learn how to avoid them where ever possible. We return to previous examples to demonstrate how the same operations can be performed using whole vectors rather than running through the indices using a loop. First the example in section 3.5.1. It is easy to avoid using loops here by adding the vectors and using the `cat(...)` and `paste(...)` functions to get the correct output.

```
> ans <- exam1 + exam2 + exam3
> cat(paste(row.names(mk2nd), "total:", ans), fill=15)
```

Not the `fill` argument of the `paste(...)` determines the width of the output. This allows us to get the output as a column (of the appropriate width). In section 3.5.2 we had nearly solved the vectorization problem but the output was not quite what we wanted. This problem is solved below.

```
> pf <- ifelse(ans>=60, "PASS", "FAIL")
> cat(paste(row.names(mk2nd), ":", pf), fill=12)
```

We now consider an example that involves the use of two functions that we have not encountered before. The function `scan(...)` is used to read vectors in from file while `apply(...)` allows us to apply functions to whole sections of arrays (a good way of avoiding loops). The `apply` function has three arguments, the first is an array (or matrix) and the third is the function that we would like to apply. The second argument gives the index of the array we would like to apply the function to. In matrices an index value of `1` corresponds to rows while an index value of `2` corresponds to columns. In the following example we scan in a set of values of the NASDAQ index for 23 days in October 2003. For each day we have 39 reading (intradaily data, ten minute readings from 8:40 to 15:00). We construct a matrix of log returns, remove the first row (why?) and work out the standard deviation of the log returns for each of the remaining 38 time points (8:50 to 15:00). Try to interpret the resulting plot.

```
> nas <- scan("NDoct2003.dat")
> naslr <- c(NA,log(nas[1:length(nas)-1])-log(nas[2:length(nas)]))
> nasmat <- array(naslr, c(39,23))
> nasmat <- nasmat[2:39,1:23]
> timesd <- apply(nasmat,1,sd) # calculate st.dev.  of rows
> length(timesd)
> plot(timesd, type="l")
```

3.6 Functions with default values*

We can exploit the ideas of named arguments and default values from 2.2.2 in functions that we write ourselves. Consider a function to draw the binomial mass function for some values of number of trial n (`size`) and probability of success p (`prob`).

```
> binomplot <- function(size, prob, colour, outputvals)
+ {  x <- 0:size
+    y <- dbinom(x, size, prob)
+    plot(x, y, type="h", col=colour)
+    if (outputvals) y
+ }

> binomplot(20, 0.2, 2, TRUE)
```

The argument `colour` is a number to specify the colour of the plotted lines while `outputvals` is a logical value; if `outputvals` is `TRUE` R will print out the values used to generate the plot. In general, we may want green lines and not printing out all of the values used to generate the plot. If this is the case, we can specify default values for these arguments.

```
> binomplot <- function(size, prob=0.5, colour=3, outputvals=FALSE)
+ {  x <- 0:size
+    y <- dbinom(x, size, prob)
+    plot(x, y, type="h", col=colour)
+    if (outputvals) y
+ }
```

Notice that the body of the function is unchanged. You can make these alterations easily using `fix(binomplot)`. Experiment with passing different arguments to the function.

```
> binomplot(20, 0.2, 2, TRUE)
> binomplot(20)
> binomplot(100)
> binomplot(100, prob=0.9)
> binomplot(100, 0.9, 4)
> binomplot(55, outputvals=TRUE, colour=1)
```

3.7 Functions as arguments*

It is often useful to be able to pass functions are arguments. For example, we may want an R function that can plot any mathematical function (not just binomial mass). In R, we treat an argument which is a function in an identical way to any other argument. The general plotting function below will plot the values of a function `ftoplot` for a specified set of `x` values.

```
> genplot <- function(ftoplot, x=seq(-10,10,length=200),
+ ptype="l", colour=2)
+ {  y <- ftoplot(x)
+    plot(x, y, type=ptype, col=colour)
+ }

> genplot(sin)
> genplot(sin, ptype="h")
> cubfun <- function(x) x^3-6*x-6
> genplot(cubfun, x=seq(-3,2,length=500))
```

This works well for plotting a (mathematical) function with a single argument, however, in many instances we need to pass more information to the (mathematical) function to be plotted. For example, to generate binomial mass using a general function like `genplot(...)`, we would like to be able to pass the values of n and p. This can be done by using a `...` argument. Any arguments that cannot be identified as arguments of the main function will be passed to the location of the `...`

```
> genplot <- function(ftoplot, x=seq(-10,10,length=200),
+ ptype="l", colour=2, ...)
+ {  y <- ftoplot(x, ...)
+    plot(x, y, type=ptype, col=colour)
+ }

> genplot(cubfun, x=seq(-4,3,length=500))
> genplot(dbinom, x=(0:20), prob=0.8, size=20, ptype="h")
```

3.8 Binary operators*

Binary operators are familiar from mathematics. The usual arithmetic operators such as $\times$ and $+$ could be viewed as functions with two arguments. Convention dictates that we write $1 + 2$ rather than $+(1, 2)$. R syntax (sensibly) follows this convention; using binary operators with the arguments either side of the function name is much easier for most of us to understand. Other binary operators include matrix mulitplication `%*%` and outer product `%o%`. R allows us to write our own binary operators of the form `%name%`. Examples follow (note that the `""` around the function name are not used when the function is called).

```
> "%p%" <- function(y,x) plot(x, y, type="l", col=2)
> x <- seq(0.1, 20, length=400)
> log(x) %p% x
> (0.3*cos(x) + 0.7*sin(2*x)) %p% x

> "%r%" <- function(y,x)
+ {  lmstore <- lm(y x)
+    lmstore$coefficients
+ }
> exam2 %r% exam1
```

3.9 Glossary for chapter 3

- **Storing commands** `savehistory(...)`
  `loadhistory(...)`
  `source(...`
- **Packages** `library(...)` list and load packages
  `data(...)` add data from a package to workspace
- **Search path** `search()` lists environments in the search path in the order in which they will be searched. Passing the position in search path or name of an environments to `objects(...)` lists objects in that environment.
- **User defined functions** `function(...){...}` the contents of the {} is a sequence of commands that form the function.

- **Formatting output** `cat(...)` prints as character string
  `format(...)` flexible formatting function
- **Coercion** `as.x(...)`
  `as(...)`
- **Class checking** `is.x(...)`
  `is(...)`
- **Loops** `for(...)`
  `while(...)`
  `repeat`
- **Arrays** `matrix(...)` to generate a matrix (2D array)
  `array(...)` to generate a general array
- **Conditional statements** `if (...)`
  `else`
  `ifelse(...)` vector conditioning

3.10 Exercises

1. Determinants for matrices with random components: In this example we generalize the example given at the end of section 3.2.7.
   (a) Write a function to generate a plot of the mass function for the determinant of a 2×2 matrix with elements that are uniformly distributed on the set $\{0, \ldots, 5\}$.
   (b) Generalize the function written above so that it will work on an arbitrary set of integers.
   (c) Provide an argument to the function that allows us (if we choose) to print out the frequency table on which the plot is based.
2. Back-shift function: this is a useful function in time series analysis. It is defined as B where $BY_t = Y_{t-1}$ and $B^d Y_t = Y_{t-d}$.
   (a) Write a back-shift function that takes a vector as its argument and returns the back-shifted version of the vector. The first element of the returned vector should be `NA`. Do not use loops! Check that your function works.
   (b) Write a version of the function that back-shifts for an arbitrary number of times d. Set a default value for d.
3. Cobweb plot: One way to view the iterations of the logistic map is using a cobweb plot.
   (a) Write an R function to plot the logistic function and add the straight line $y = x$ to the plot (use the `lines()`) command.
   (b) Edit your function so that, from a starting point of $x = 0.1$, it draws a line up (vertically) to the logistic curve and then over (horizontally) to the line $y = x$.

(c) Include a loop to take a line vertically from the current point to the logistic curve then horizontally over to the line $y = x$ a fixed number of time (thus constructing the cobweb plot).

(d) Is there anyway to do this without using a loop?

4. Conditional likelihood for autoregressive process: A Gaussian autoregressive process of order 1 is a simple time series model in which the current observation is modelled explicitly as a function of the previous observation:

$$Y_t = \phi Y_{t-1} + \varepsilon_t,$$

where $\varepsilon \sim N(0, \sigma_\varepsilon^2)$. The log-likelihood for observations $\{y_1, \ldots, y_n\}$ is approximated by

$$\ell(\phi, \sigma_\varepsilon^2) = -\frac{n}{2} \log \sigma_\varepsilon^2 - \frac{1}{2\sigma_\varepsilon^2} \sum_{i=1}^{n} (y_t - \phi y_{t-1})^2.$$

(a) The file (saved workspace) `arsim.Rdata` contains a data frame (`simdata`) of values simulated from an AR(1) with ϕ taking values between -0.9 and 0.9. Use the `load(...)` command to add the objects from the file to your workspace. Write a function that will generate a time series plot of any named columns from a data frame. Test out your function on the `simdata` data frame.

(b) Write a function that evaluates the likelihood for values of ϕ and σ_ε^2 for a specified data vector.

(c) Write a function that uses the function from part ii. to draw the likelihood surface for a given region of the parameter space.

5. † Bifurcation plot: The convergence properties of the logistic map for different values of r are summarized in a bifurcation plot. The bifurcation plot has values of r on the x axis and the values that the logistic map converges to on the y axis. For small values of r the map converges to a single value. As r increases we see period doubling and then chaotic regions. By repeated running the logistic map for a fixed number of iterations and plotting the last 50 values, write a function that will draw a bifurcation plot.

6. †† Mandelbrot set: The Mandelbrot set is a subset of the complex plane. Consider the map

$$z_{n+1} = z_n^2 + c \tag{3.2}$$

where $z_0 = c$. The Mandelbrot set consists of those points c in the complex plane for which the limit as n tends to infinity of $|z_n|$ is finite. Write an R function to construct a reasonable visual impression of the Mandelbrot set.

3.11 Reading

3.11.1 Directly related reading

- Venables, W. N. *et. al.* (2001) *An Introduction to R.* [Chapter 5 for arrays and matrices, chapter 9 for program control and chapter 10 for writing functions.]

- Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*, 4th edition, Springer. [Section 3.5 for formatting a printing.]

3.11.2 Supplementary reading†

- Venables, W. N. and Ripley, B. D. (2000) *S Programming*, Springer. [Much of what is said here works in R.]
- Bentley, J (2000) Programming Pearls, 2nd edition. Addison-Wesley. [An entertaining book that contains a lot of wisdom about programming.]