```
# program: spuRs/resources/scripts/threexplus1.r
x <- 3
for (i in 1:3) {
  show(x)
  cat("i = ", i, "\n")
  if (x %% 2 == 0) {
    x <- x/2
  } else {
    x <- 3*x + 1
  }
}
show(x)
```

Running the program gives the following output

```
> source("../scripts/threexplus1.r")
[1] 3
i =  1
[1] 10
i =  2
[1] 5
i =  3
[1] 16
```

It is good programming style to solve the simplest possible version of the problem at hand, and then add complexity only as it becomes necessary. Although such an organic approach seems slow at first blush, it provides considerable protection against the complexities that inevitably accrue as the full exercise takes shape.

It is also very helpful to make dry runs of your code, using simple starting conditions for which you know what the answer should be. These dry runs should ideally use short and simple versions of the final program, so that analysis of the output can be kept as simple as possible. Graphs and summary statistics of intermediate outcomes can be very revealing, and the code to create them is easily commented out for production runs.

A more sophisticated approach would be to add an extra logical argument to the function (a flag), named `reporting` say, with default `FALSE`. We could then enclose all diagnostic output inside an `if (reporting)` statement, so by default it will not be printed, but can be easily turned on by setting the flag argument to `TRUE`. This approach creates a modest overhead cost of requiring the evaluation of the condition at each run of the function.

Careful use of indentation and spacing will improve the readability of your code considerably. Indentation can be used to reinforce the overall structure of the code, for example, to show where loops and conditional statements begin and end. Some text editors, for example, the emacs family, provide syntactically aware indentation, which facilitates writing such code.

3.8 Good programming habits

Good programming is clear rather than clever. Being clever is good, but given a choice, being clear is preferable. The reason for this is that in practice much more time is spent correcting and modifying programs than is ever spent writing them, and if you are to be successful in either correcting or modifying a program, you will need it to be clear.

You will find that even programs you write yourself can be very difficult to understand after only a few weeks have passed.

We find the following to be useful guidelines: start each program with some comments giving the name of the program, the author, the date it was written, and what the program does. A description of what a program does should explain what all the inputs and outputs are.

Variable names should be descriptive, that is, they should give a clue as to what the value of the variable represents. Avoid using reserved names or function names as variable names (in particular `t`, `c`, and `q` are all function names in R). You can find out whether or not your preferred name for an object is already in use by the `exists` function.

Use blank lines to separate sections of code into related parts, and use indenting to distinguish the inside part of an `if` statement or a `for` or `while` loop.

Document the programs that you use in detail, ideally with citations for specific algorithms. There is no worse feeling than returning to undocumented code that had been written several years earlier to try to find and then explain an anomaly.

3.9 Exercises

1. Consider the function $y = f(x)$ defined by

x	≤ 0	$\in (0,1]$	> 1
$f(x)$	$-x^3$	x^2	$\sqrt{x}$

Supposing that you are given x, write an R expression for y using `if` statements.

Add your expression for y to the following program, then run it to plot the function f.

```
# input
x.values <- seq(-2, 2, by = 0.1)

# for each x calculate y
n <- length(x.values)
```

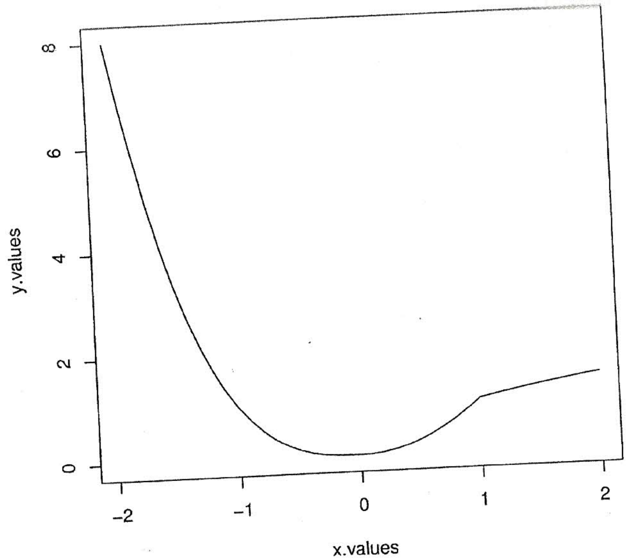

Figure 3.2 *The graph produced by Exercise 1.*

```
y.values <- rep(0, n)
for (i in 1:n) {
    x <- x.values[i]
    # your expression for y goes here
    y.values[i] <- y
}

# output
plot(x.values, y.values, type = "l")
```

Your plot should look like Figure 3.2. Do you think f has a derivative at 1? What about at 0?

We remark that it is possible to vectorise the program above, using the `ifelse` function.

2. Let $h(x,n) = 1 + x + x^2 + \cdots + x^n = \sum_{i=0}^{n} x^i$. Write an R program to calculate $h(x,n)$ using a `for` loop.
3. The function $h(x,n)$ from Exercise 2 is the finite sum of a geometric sequence. It has the following explicit formula, for $x \neq 1$,

$$h(x,n) = \frac{1 - x^{n+1}}{1 - x}.$$

Test your program from Exercise 2 against this formula using the following values

x	n	$h(x,n)$
0.3	55	1.428571
6.6	8	4243335.538178

You should use the computer to calculate the formula rather than doing it yourself.

4. First write a program that achieves the same result as in Exercise 2 but using a `while` loop. Then write a program that does this using vector operations (and no loops).

   If it doesn't already, make sure your program works for the case $x = 1$.
5. To rotate a vector $(x,y)^T$ widdershins (anticlockwise) by θ radians, you premultiply it by the matrix

$$\begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}.$$

   Write a program in R that does this for you.
6. Given a vector x, calculate its geometric mean using both a for loop and vector operations. (The geometric mean of $x_1, \ldots, x_n$ is $(\prod_{i=1}^{n} x_i)^{1/n}$.)

   You might also like to have a go at calculating the harmonic mean, $(\sum_{i=1}^{n} 1/x_i)^{-1}$, and then check that if the x_i are all positive, the harmonic mean is always less than or equal to the geometric mean, which is always less than or equal to the arithmetic mean.
7. How would you find the sum of every third element of a vector x?
8. How does program `quad2.r` (Exercise 3.2.1) behave if `a2` is 0 and/or `a1` is 0? Using `if` statements, modify `quad2.r` so that it gives sensible answers for all possible (numerical) inputs.
9. Chart the flow through the following two programs.

   (a). The first program is a modification of the example from Section 3.6, where x is now an array. You will need to keep track of the value of each element of x, namely $x[1]$, $x[2]$, etc.

```
# program spuRs/resources/scripts/threexplus1array.r

x <- 3
for (i in 1:3) {
  show(x)
  if (x[i] %% 2 == 0) {
    x[i+1] <- x[i]/2
  } else {
    x[i+1] <- 3*x[i] + 1
  }
}
show(x)
```

(b). The second program implements the Lotka–Volterra model for a 'predator-prey' system. We suppose that $x(t)$ is the number of prey animals at the start of a year t (rabbits) and $y(t)$ is the number of predators (foxes), then the Lotka–Volterra model is:

$$\begin{aligned} x(t+1) &= x(t) + b_r \cdot x(t) - d_r \cdot x(t) \cdot y(t); \\ y(t+1) &= y(t) + b_f \cdot d_r \cdot x(t) \cdot y(t) - d_f \cdot y(t); \end{aligned}$$

where the parameters are defined by:

b_r is the natural birth rate of rabbits in the absence of predation;
d_r is the death rate per encounter of rabbits due to predation;
d_f is the natural death rate of foxes in the absence of food (rabbits);
b_f is the efficiency of turning predated rabbits into foxes.

```
# program spuRs/resources/scripts/predprey.r
# Lotka-Volterra predator-prey equations
br <- 0.04    # growth rate of rabbits
dr <- 0.0005 # death rate of rabbits due to predation
df <- 0.2     # death rate of foxes
bf <- 0.1     # efficiency of turning predated rabbits into foxes
x <- 4000
y <- 100
while (x > 3900) {
  # cat("x =", x, " y =", y, "\n")
  x.new <- (1+br)*x - dr*x*y
  y.new <- (1-df)*y + bf*dr*x*y
  x <- x.new
  y <- y.new
}
```

Note that you do not actually need to know anything about the program to be able to chart its flow.

10. Write a program that uses a loop to find the minimum of a vector `x`, without using any predefined functions like `min(...)` or `sort(...)`.

    You will need to define a variable, `x.min` say, in which to keep the smallest value you have yet seen. Start by assigning `x.min <- x[1]` then use a `for` loop to compare `x.min` with `x[2]`, `x[3]`, etc. If/when you find `x[i] < x.min`, update the value of `x.min` accordingly.

11. Write a program to merge two sorted vectors into a single sorted vector.

    Do not use the `sort(x)` function, and try to make your program as efficient as possible. That is, try to minimise the number of operations required to merge the vectors.

√ 12. The dice game craps is played as follows. The player throws two dice, and if the sum is seven or eleven, then he wins. If the sum is two, three, or twelve, then he loses. If the sum is anything else, then he continues throwing until he either throws that number again (in which case he wins) or he throws a seven (in which case he loses).

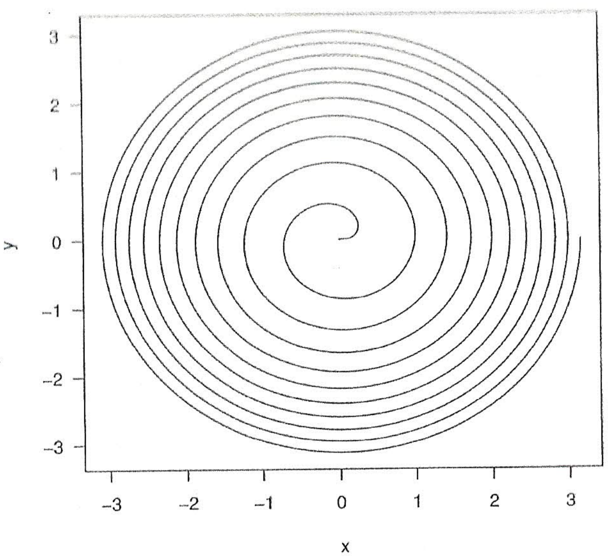

Figure 3.3 *The output from Exercise 13.*

Write a program to simulate a game of craps. You can use the following snippet of code to simulate the roll of two (fair) dice:

```
x <- sum(ceiling(6*runif(2)))
```

13. Suppose that $(x(t), y(t))$ has polar coordinates $(\sqrt{t}, 2\pi t)$. Plot $(x(t), y(t))$ for $t \in [0, 10]$. Your plot should look like Figure 3.3.

14. Improve the code for program `threexplus1.r` as shown in Section 3.7. Assume that the intermediary calls to `show` and `cat` are unnecessary.

15. A room contains 100 toggle switches, originally all turned off. 100 people enter the room in turn. The first toggles every switch, the second toggles every second switch, the third every third switch, and so on, to the last person who toggles the last switch only.

    At the end of this process, which switches are turned on?

One way of having more than one plot visible is to open additional graphics devices. In a Windows environment this is done by using the command `windows()` before each additional plot. In Unix use the command `X11()` instead, and for MacOS use `quartz()`. See `?dev.new` and `?dev.control` for more information.

Alternatively you can create a grid of plots in a single graphics window using the commands `par(mfrow = c(nr, nc))` or `par(mfcol = c(nr, nc))`. The command `par` is used to set many different parameters that control how graphics are produced. Setting `mfrow = c(nr, nc)` creates a grid of plots with `nr` rows and `nc` columns, which is filled row by row. `mfcol` is similar but fills the plots column by column.

The following example illustrates `mfrow` and the function `curve`, which is used to plot the function $x\sin(x)$ over different ranges.

```
> par(mfrow = c(2, 2), mar=c(5, 4, 2, 1))
> curve(x*sin(x), from = 0, to = 100, n = 1001)
> curve(x*sin(x), from = 0, to = 10, n = 1001)
> curve(x*sin(x), from = 0, to = 1, n = 1001)
> curve(x*sin(x), from = 0, to = 0.1, n = 1001)
> par(mfrow = c(1, 1))
```

The output is given in Figure 4.2.

We return to the subject of plotting in Chapter 7.

4.6 Exercises

1. Here are the first few lines of the files `age.txt` and `teeth.txt`, taken from the database of a statistically minded dentist:

```
ID    Age
1     18
2     19
3     17
.     .
.     .
.     .

ID    Num_Teeth
1     28
2     27
3     32
.     .
.     .
.     .
```

Write a program in R to read each file, and then write an amalgamated list to the file `age_teeth.txt`, of the following form:

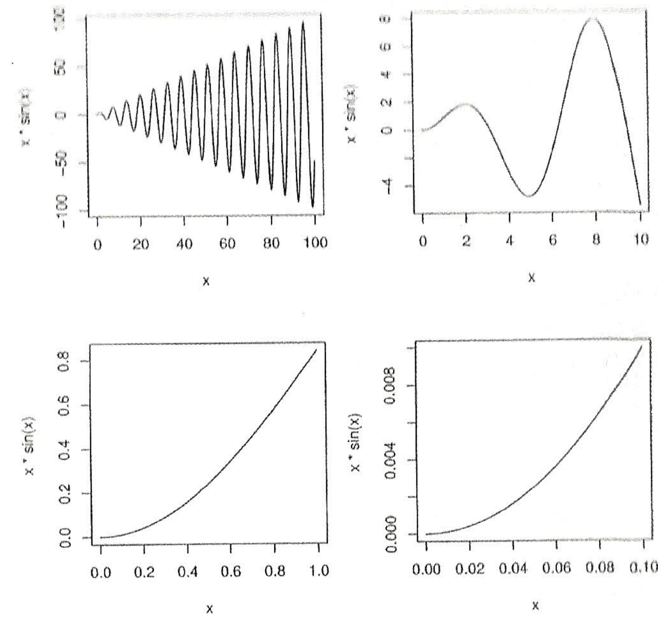

Figure 4.2 *An array of plots. Refer to Section 4.5 for the code to produce this diagram.*

```
ID    Age    Num_Teeth
1     18     28
2     19     27
3     17     32
.     .      .
.     .      .
.     .      .
```

2. The function `order(x)` returns a permutation of `1:length(x)` giving the order of the elements of `x`. For example

```
> x <- c(1.1, 0.7, 0.8, 1.4)
> (y <- order(x))
[1] 2 3 1 4
> x[y]
[1] 0.7 0.8 1.1 1.4
```

Using `order` or otherwise, modify your program from Exercise 1 so that the output file is ordered by its second column.

3. Devise a program that outputs a table of squares and cubes of the numbers 1 to n. For n <- 7 the output should be as follows:

```
> source("../scripts/square_cube.r")
 number  square   cube

     1       1       1
     2       4       8
     3       9      27
     4      16      64
     5      25     125
     6      36     216
     7      49     343
```

4. Write an R program that prints out the standard multiplication table:

```
> source("../scripts/mult_table.r")
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
 [1,]    1    2    3    4    5    6    7    8    9
 [2,]    2    4    6    8   10   12   14   16   18
 [3,]    3    6    9   12   15   18   21   24   27
 [4,]    4    8   12   16   20   24   28   32   36
 [5,]    5   10   15   20   25   30   35   40   45
 [6,]    6   12   18   24   30   36   42   48   54
 [7,]    7   14   21   28   35   42   49   56   63
 [8,]    8   16   24   32   40   48   56   64   72
 [9,]    9   18   27   36   45   54   63   72   81
```

Hint: generate a matrix `mtable` that contains the table, then use `show(mtable)`.

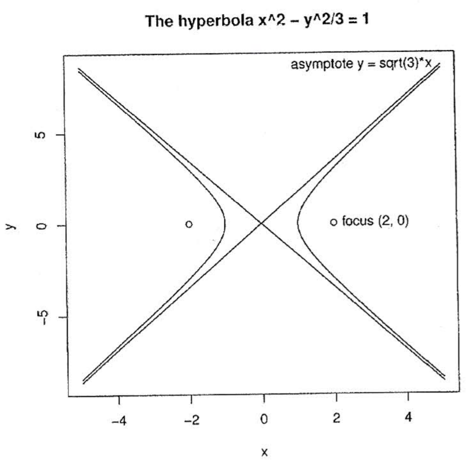

Figure 4.3 *The hyperbola $x^2 - y^2/3 = 1$; see Exercise 5.*

5. Use R to plot the hyperbola $x^2 - y^2/3 = 1$, as in Figure 4.3.

CHAPTER 5

Programming with functions

In this chapter we cover the creation of functions, the rules that they must follow, and how they relate to and communicate with the environments from which they are called. We also present some tips on the construction of efficient functions, with especial reference to how functions are treated in R.

Functions are one of the main building blocks for large programs: they are an essential tool for structuring complex algorithms.

In some other programming languages *procedures* and *subroutines* play the same role as functions in R. The computer code used to navigate and control the successful Apollo missions to the moon in the 60's was written in a low-level assembly language to work efficiently within very limited hardware resources. But even then a special higher order language was developed to translate assembly code modules into a set of 'subroutines' or 'functions'.

Nowadays in high level programming languages like R, the concept of a 'function' is a powerful tool for structuring programs. User-defined functions are now one of the main building blocks for developing sophisticated software. Arguably one of R's strengths as a tool for scientific programming is the ease with which it can be extended for specific purposes, using functions written by the R community and made available as R packages (see Section 8.1 for more on the latter).

5.1 Functions

A function has the general form

```
name <- function(argument_1, argument_2, ...) {
    expression_1
    expression_2
    <some other expressions>
    return(output)
}
```

Here `argument_1`, `argument_2`, etc., are the names of variables and `expression_1`, `expression_2`, and `output` are all regular R expressions. `name` is the name of the function. Note that some functions have no arguments, and