Week 3 - Concepts in Mathematical Modelling

Introduction

One of uses of R is re-creating equations that we use for mathematical modelling. Consider the equation for quadratic equations.

Here is a simple example of a program for calculating the real roots of a quadratic equation. Note the use of # for commenting the code.

```
# clear the workspace
rm(list=ls())

# input
a2 <- 1
a1 <- 4
a0 <- 2

# calculation
root1 <- (-a1 + sqrt(a1^2 - 4*a2*a0))/(2*a2)
root2 <- (-a1 - sqrt(a1^2 - 4*a2*a0))/(2*a2)

# output
print(c(root1, root2))
```

```
[1] -0.5857864 -3.4142136
```

Executing this code (running the program) produces the following output

```
[1] -0.5857864 -3.4142136
```

In order to write programs that implement mathematical algorithms, we need to be able to make choices and repeat operations. These tasks are achieved using the if command and the for and while commands.

Branching with if

It is often useful to choose the execution of some or other part of a program to depend on a condition. The if function has the form

```
if (logical_expression) {
expression_1
...
}
```

A natural extension of the if command includes an else part:

```
if (logical_expression) {
expression_1
...
} else {
expression_2
...
}
```

Braces { } are used to group together one or more expressions. If there is only one expression then the braces are optional. When an if expression is evaluated, if logical_expression is TRUE then the first group of expressions is executed and the second group of expressions is not executed. Conversely if logical_expression is FALSE then only the second group of expressions is executed. if statements can be nested to create elaborate pathways through a program.

Warning: because the else part of an if statement is optional, if you type

```
if (logical_expression) {
expression_1
...}
else {
expression_2
...}
```

then you get an error. This is because R believes the if statement is finished before it sees the else part, which appears on a separate line. That is, R treats the else as the start of a new command, but there is no command that starts with an else, so R generates an error. Other useful functions for conditional execution are *ifelse*, which we cover further in this tutorial, and *switch*, which allows for multiple branches.

Here is an improved version of our program for finding the roots of a quadratic. Try it with some different values of a2, a1, and a0.

```
# program spuRs/resources/scripts/quad2.r
# find the zeros of a2*x^2 + a1*x + a0 = 0
# clear the workspace
rm(list=ls())
# input
a2 <- 1
a1 <- 4
a0 <- 5
# calculate the discriminant
discrim <- a1^2 - 4*a2*a0
# calculate the roots depending on the value of the discriminant
if (discrim > 0) {
roots <- c( (-a1 + sqrt(a1^2 - 4*a2*a0))/(2*a2),
(-a1 - sqrt(a1^2 - 4*a2*a0))/(2*a2) )
} else {
if (discrim == 0) {
roots <- -a1/(2*a2)
} else {
roots <- c()
}
}
# output
show(roots)
```

```
NULL
```

Expressions that are grouped using braces { } are viewed by R as a single expression. Similarly, an if command is viewed as a single expression. Thus the code

```
if (logical_expression_1) {
expression_1
...
} else {
if (logical_expression_2) {
expression_2
...
} else {
expression_3
```

```
...
}
}
```

can be written equivalently (and more clearly) as

```
if (logical_expression_1) {
expression_1
...
} else if (logical_expression_2) {
expression_2
...
} else {
expression_3
...
}
```

Looping with for

The for command has the following form, where x is a simple variable and vector is a vector.

```
for (x in vector) {
expression_1
...
}
```

When executed, the for command executes the group of expressions within the braces { } once for each element of vector. The grouped expressions can use x, which takes on each of the values of the elements of vector as the loop is repeated. Note that vector can be a list, which we cover in a future tutorial.

The following example uses a loop to sum the elements of a vector. Note that we use the function cat (for concatenate) to display the values of certain variables. The advantage of cat over print or show is that it allows us to combine text and variables together. The combination of characters "\n" (backslash-n) is used to 'print' a new line.

Also note that to sum the elements of a vector, it is more accurate and much easier (but less instructive) to use the built-in function sum.

```
(x_list <- seq(1, 9, by = 2))
```

```
[1] 1 3 5 7 9
```

```
sum_x <- 0
for (x in x_list) {
  sum_x <- sum_x + x
  cat("The current loop element is", x, "\n")
  cat("The cumulative total is", sum_x, "\n")
}
```

```
The current loop element is 1
The cumulative total is 1
The current loop element is 3
The cumulative total is 4
The current loop element is 5
The cumulative total is 9
The current loop element is 7
The cumulative total is 16
The current loop element is 9
The cumulative total is 25
```

```
sum(x_list)
```

```
[1] 25
```

The following program calculates n!.

```
# program: spuRs/resources/scripts/nfact1.r
# Calculate n factorial
# clear the workspace
rm(list=ls())
# Input
n <- 6
# Calculation
n_factorial <- 1
for (i in 1:n) {
n_factorial <- n_factorial * i
}
# Output
show(n_factorial)
```

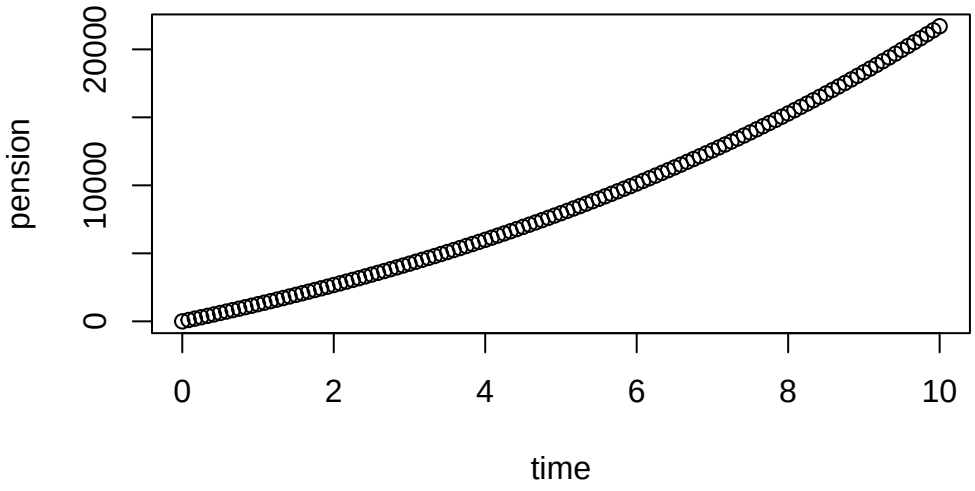
```
[1] 720
```

Note that we can also compute the factorial easily using prod(1:n) or even factorial(n).

Example: pension value pension.r

Here is an example for calculating the value of a pension fund under compounding interest. It uses the function floor(x), whose value is the largest integer smaller than x.

```
# program: spuRs/resources/scripts/pension.r
# Forecast pension growth under compound interest
# clear the workspace
rm(list=ls())
# Inputs
r <- 0.11 # Annual interest rate
term <- 10 # Forecast duration (in years)
period <- 1/12 # Time between payments (in years)
payments <- 100 # Amount deposited each period
# Calculations
n <- floor(term/period) # Number of payments
pension <- 0
for (i in 1:n) {
pension[i+1] <- pension[i]*(1 + r*period) + payments
}
time <- (0:n)*period
# Output
plot(time, pension)
```



The next example highlights an inefficiency in pension.r.

Example: redimensioning an array

Here is an observation that you may be able to use to make some of your programs run faster. The following two programs produce the same result, but the first is faster.

0 2 4 6 8 10 0 5000 10000 15000 20000 time pension

Figure 3.1 Value of a pension fund: output from Exercise 3.3.3.

```
Program 1
n <- 1000000
x <- rep(0, n)
for (i in 1:n) {
x[i] <- i
}
Program 2
n <- 1000000
x <- 1
for (i in 2:n) {
x[i] <- i
}
```

The reason for the difference is a technical one, namely changing the size of a vector takes just about as long as creating a new vector does. This because each time the size changes, R needs to reconsider its allocation of memory to the object. In the second program, each statement x[i] <- i changes the length of x from i - 1 to i, and this is what makes it slower than the first program.

Looping with while

Often we do not know beforehand how many times we need to go around a loop. That is, each time we go around the loop, we check some condition to see if we are done yet. In this situation we use a while loop, which has the form

```
while (logical_expression) {
expression_1
...
}
```

When a while command is executed, logical_expression is evaluated first. If it is TRUE then the group of expressions in braces { } is executed. Control is then passed back to the start of the command: if logical_expression is still TRUE then the grouped expressions are executed again, and so on. Clearly, for the loop to stop eventually, logical_expression must eventually be FALSE. To achieve this logical_expression usually depends on a variable that is altered within the grouped expressions. The while loop is more fundamental than the for loop, as we can always rewrite a for loop as a while loop.

Example: Fibonacci numbers fibonacci.r

Consider the Fibonacci numbers F1, F2, . . ., which are defined inductively using the rules F1 = 1, F2 = 1, and Fn = Fn−1 + Fn−2 for n 2. Suppose that you wished to know the first Fibonacci number larger than 100. We can find this using a while loop as follows:

```r
# program: spuRs/resources/scripts/fibonacci.r
# calculate the first Fibonacci number greater than 100
# clear the workspace
rm(list=ls())
# initialise variables
F <- c(1, 1) # list of Fibonacci numbers
n <- 2 # length of F
# iteratively calculate new Fibonacci numbers
while (F[n] <= 100) {
# cat("n =", n, " F[n] =", F[n], "\n")
n <- n + 1
```

```
F[n] <- F[n-1] + F[n-2]
}
# output
cat("The first Fibonacci number > 100 is F(", n, ") =", F[n], "\n")
```

```
The first Fibonacci number > 100 is F( 12 ) = 144
```

Example: compound interest compound.r

In this example we use a while loop to work out how long it will take to pay off a loan.

```
# program: spuRs/resources/scripts/compound.r
# Duration of a loan under compound interest
# clear the workspace
rm(list=ls())
# Inputs
r <- 0.11 # Annual interest rate
period <- 1/12 # Time between repayments (in years)
debt_initial <- 1000 # Amount borrowed
repayments <- 12 # Amount repaid each period
# Calculations
time <- 0
debt <- debt_initial
while (debt > 0) {
time <- time + period
debt <- debt*(1 + r*period) - repayments
}
# Output
cat('Loan will be repaid in', time, 'years\n')
```

```
Loan will be repaid in 13.25 years
```

Vector-based programming

It is often necessary to perform an operation upon each of the elements of a vector. For example, given a vector of measurements in inches, we may wish to convert them all to centimetres. R is set up so that such programming tasks can be accomplished using vector operations rather than looping. Using vector operations is more efficient computationally, as well as more concise literally. For example, we could find the sum of the first n squares using a loop as follows:

```
n <- 100
S <- 0
for (i in 1:n) {
  S <- S + i^2
}
S
```

```
[1] 338350
```

Alternatively, using vector operations we have:

```
sum((1:n)^2)
```

```
[1] 338350
```

Here, R has interpreted 1:n to mean "the integers from 1 up to n, inclusive", then squared each of those integers using the vectorised "^2", and added them up in sum. Of course, for the above example we can also use the formula n(n+1)(2n+1)/6, assuming that we remember it.

The powerful ifelse function performs elementwise conditional evaluation upon a vector. ifelse(test, A, B) takes three vector arguments: a logical expression test, and two expressions A and B. The function returns a vector that is a combination of the evaluated expressions A and B: the elements of A that correspond to the elements of test that are TRUE, and the elements of B that correspond to the elements of test that are FALSE. As before, if the vectors have differing lengths then R will repeat the shorter vector(s) to match the longer, if possible. An example follows.

```
x <- c(-2, -1, 1, 2)
ifelse(x > 0, "Positive", "Negative")
```

```
[1] "Negative" "Negative" "Positive" "Positive"
```

Two other useful functions are pmin and pmax, which provide vectorised versions of the minimum and maximum. For example,

```
pmin(c(1,2,3),c(3,2,1),c(2,2,2))
```

```
[1] 1 2 1
```

Program flow

The term flow is used to describe how control of a program moves from one line to another, and can be altered by if statements, for loops and while loops (and functions, as we will see later). Given a program, we can chart its flow by numbering each line, making sure we have a single command per line, then systematically working out the order in which each line is visited. To do this we need to keep a list of all the variables in use and their values, as they can affect the flow.

Consider the following example; line numbers are given on the left

```
# program: spuRs/resources/scripts/threexplus1.r
1 x <- 3
2 for (i in 1:3) {
3 show(x)
4 if (x %% 2 == 0) {
5 x <- x/2
6 } else {
7 x <- 3*x + 1
8 }
9 }
10 show(x)
```

Charting the flow through this program, we get the output presented in Table 3.1. This is exactly what the computer does when it executes a program: it keeps track of its current position in the program and maintains a list of variables and their values. Whatever line you are currently at, if you know all the variables then you always know which line to go to next.

Pseudo-code

Pseudo-code is used to describe shorthand and/or informally written programs. Pseudo-code does not conform to the strict syntax (grammatical rules) of any particular programming language, but it does use variables, arrays, if statements and loops. That is, it contains enough information to work out how control will flow through the program.

As you learn other high-level programming languages, you will see that the fundamental programming structures—such as variables, arrays, if statements, and loops—are common to all of them. Pseudo-code pays attention to these fundamentals but ignores the details. It is a useful way of describing algorithms without worrying about all the bookkeeping required of a full program.

Table 3.1 Charting the flow for program threexplus1.r

line	x	i	comments
1	3	-	i not defined yet
2	3	1	i is set to 1
3	3	1	3 written to screen
4	3	1	(x %% 2 == 0) is FALSE so go to line 7
7	1	0	1 x is set to 10
8	10	1	end of else part
9	10	1	end of for loop, not finished so back to line 2
2	10	2	i is set to 2
3	10	2	10 written to screen
4	10	2	(x %% 2 == 0) is TRUE so go to line 5
5	5	2	x is set to 5
6	5	2	end of if part, go to line 9
9	5	2	end of for loop, not finished so back to line 2
2	5	3	i is set to 3
3	5	3	5 written to screen
4	5	3	(x %% 2 == 0) is FALSE so go to line 7
7	16	3	x is set to 16
8	16	3	end of else part
9	16	3	end of for loop, finished so continue to line 10
10	16	3	16 written to screen

Basic debugging

We recognise errors in a few ways. First, R may stop processing and report an error, along wi

::: {.cell}

```{.r .cell-code}
options(warn = 2)
```

:::

You will spend a lot of time correcting errors in your programs. To find an error or bug, you need to be able to see how your variables change as you move through the branches and loops of your code. An effective and simple way of doing this is to include statements like

```
cat("var =", var, "\n")
```

throughout the program, to display the values of variables such as var as the program executes. Once you have the program working you can delete these or just comment them so they are not executed.

For example, if we wanted to see how the variable i changed in the program above, we could add a line as follows:

```
# 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
}
}
```

```
[1] 3
i =  1
[1] 10
i =  2
[1] 5
i =  3
```

```
show(x)
```

```
[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.

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