Programming in S

For-Loop Statements

The general form of a for-loop is

for(*variable* **in** *vector***)** *expression*

where expression is either a simple or compound expression.

The loop is evaluated by successively setting the value of the *variable* to each element of the *vector* and then evaluating the *expression*. The value of the loop is the value of the last expression computed during evaluation of the loop.

Control Flow

- S is extensible because it is a full-blown computer language.
- Like most computer languages, S has a number of *control-flow* features designed to make it possible to carry out interesting calculations.
- These features fall into two general categories — *iteration* and *alternation*. We will look at examples of each of these features.

For-Loop Statements

S for-loops are not restricted to just incrementing a simple integer index. The following code fragments are equivalent (but the second is more efficient).

```
> sum <- 0
> for(i in 1:length(x))
+   sum <- sum + x[i]

> sum <- 0
> for(elt in x)
+   sum <- sum + elt
```

This illustrates that you should not just translate idioms from other languages into S.

Compound Statements

Compound expressions give a way of treating a sequence of expressions as a single expression. The general form of a compound expression is:

{ $expression_1$ **; ... ;** $expression_n$ }

A compound expression is evaluated by evaluating each of its component expressions in turn and taking the value of the last one as the value of the compound. (Note that the statements here are separated by semi-colons, but newlines will also serve as separators.)

If-Then-Else and If-Then Statements

In many programs it may be desirable to perform one computation if a particular condition is true, and to otherwise perform some alternative computation. This is made possible through the use of an *if-then-else* statement.

if (*condition* **)** $expression_1$ **else** $expression_2$

where *condition* is an expression which results in a logical (true/false) value, and $expression_1$ and $expression_2$ are simple or compound expressions.

If *condition* is true, then $expression_1$ is evaluated, otherwise $expression_2$ is evaluated. The value of an if-then-else statement is the value of whichever of $expression_1$ or $expression_2$ which was computed.

For-Loop Statements

For loops give a way of repeatedly repeatedly carrying out a computation, perhaps with some slight variation. For example, the effect of the code fragment:

```
> sum <- 0
> for(i in 1:length(x))
+   sum <- sum + x[i]
```

is to set the variable **i** successively equal to each of the values **1**, **2**, . . ., **length(x)**, and for each of the successive values to evaluate the expression **sum <- sum + x[i]**.

Examples

The expression

```
> if (x > 0) y <- sqrt(x) else y <- -sqrt(-x)
```

provides an example of an if-then-else statement which will look familiar to Pascal, Java, C, or C++ programmers.

The statement can however be written more succinctly in S as

```
> y <- if (x > 0) sqrt(x) else -sqrt(-x)
```

which will look familiar to Lisp or Algol programmers.

If Statements

There is a simplified form of if-then-else statement which is available when there is no *expression*$_2$ to evaluate. This statement has the general form

if (*condition* **)** *expression*

and is completely equivalent to the statement

if (*condition* **)** *expression* **else NULL**

Here **NULL** is a special null object which acts as a placeholder.

Example

Consider the function defined by:

```
> hypot <- function(a, b) sqrt(a^2 + b^2)
```

and suppose we make a call to this function by typing:

```
> hypot(3, 4)
[1] 5
```

This function call is evaluated as follows:

- Temporary variables, **a** and **b**, are created with the values 3 and 4.
- These variable definitions are used to evaluate the expression **sqrt(a^2 + b^2)** to obtain the value **5**.

Functions

New functionality is added to S by defining new *functions*. As a very simple example, let's define a function which squares its argument. We can create this function as follows.

```
> square <- function(x) x * x
```

The expression **function(x) x * x** creates a function which is assigned as the value of the variable **square**. The function has a single argument x and the value is multiplied by itself to provide the value of the function. We can use this function in exactly the same way as any other S function.

```
> square(10)
[1] 100
```

- When the evaluation is complete the temporary definitions of **a** and **b** are removed.

Vectorisation

Because the operation ***** acts element-wise on vectors, the new square function will also.

```
> square(1:10)
 [1]   1   4   9  16  25  36  49  64  81
[10] 100
```

Using this fact we can write a simple sum-of-squares function.

```
> sumsq <- function(x) sum(square(x))
> sumsq(1:10)
[1] 385
```

Optional Arguments

S has a notion of default argument values which makes it possible for the writer of a function to specify reasonable default values for arguments, while still providing the flexibility users the option of overriding these defaults. As an example, consider the following sum-of-squares function.

```
> sumsq <-
+     function(x, about = 0)
+     sum((x - about)^2)
```

The function definition provides a default definition for the **about** argument.

General Functions

In general, an S function has the form:

function (*arglist* **)** *body*

where *arglist* is a (comma separated) list of variable names known as the *formal arguments* of the function, and *body* is a simple or compound expression known as the body of the function.

The general rule for evaluating a call to a function is to temporarily create a set of variables by associating the arguments passed to the function with the variable names in *arglist*, and then to use these variable definitions to evaluate the function body.

Example

When invoked with just a single argument the function returns the sum of the squared values in that argument.

```
> sumsq(1:10)
[1] 385
```

When provided with a value for the **about** argument, the function computes the sum of squared deviations about the specified value.

```
> sumsq(1:10, mean(1:10))
[1] 82.5
```

Argument Matching

Because it is not necessary to specify all the arguments to S functions, it is important to be clear about which argument corresponds to which formal parameter of the function. This can be done by providing a name for the argument explicitly.

```
> sumsq(1:10, about=mean(1:10))
[1] 82.5
```

When names are provided for arguments, they are used in preference to position which matching up formal arguments and arguments. For example,

```
> sumsq(about=mean(1:10), 1:10)
[1] 82.5
```

returns the same answer as the function call above.

The Roots of a Quadratic Equation – Version 1

The following function computes both roots of a quadratic equation.

```
> roots <-
+   function(a, b, c)
+   c((-b + sqrt(b^2 - 4 * a * c))/(2 * a),
+     (-b - sqrt(b^2 - 4 * a * c))/(2 * a))

> roots(1,2,1)
[1] -1 -1
```

This function computes each root separately and then uses **c** to combine the two results into a vector.

The Argument Matching Process

The general rule for matching formal and actual arguments is as follows.

1. Use any names provided with the actual arguments to determine the formal arguments associated with the named arguments. Partial matches are acceptable, unless there is an ambiguity.
2. Match the unused actual arguments, in the order given, to any unmatched formal arguments, in the order they appear in the function declaration.

The Roots of a Quadratic Equation – Version 2

The following function computes both roots of a quadratic equation.

```
> roots <-
+   function(a, b, c)
+   (-b + c(1, -1) *
+    sqrt(b^2 - 4 * a * c))/(2 * a)

> roots(1, 2, 1)
[1] -1 -1
```

This version of the function uses vector arithmetic to compute both roots simultaneously.

Argument Matching Example

If **sumsq** is defined by

```
> sumsq <-
+     function(x, about = 0)
+     sum((x - about)^2)
```

then all the following calls to **sumsq** are equivalent.

```
> sumsq(1:10, mean(1:10))
> sumsq(1:10, about=mean(1:10))
> sumsq(1:10, a=mean(1:10))
> sumsq(x=1:10, mean(1:10))
> sumsq(mean(1:10), x=1:10)
```

Complex Roots

If the quantity Δ defined by

$$\Delta = b^2 - 4ac$$

is negative, then the roots of the quadratic equation are complex.

Versions 1 and 2 of the roots function do not address the problem of complex roots.

```
> roots(1, 0, 1)
[1] NaN NaN
Warning message:
NaNs produced in: sqrt(b^2 - 4 * a * c)
```

The Roots of a Quadratic Equation

The formula

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

gives the values of x for which the function

$$f(x) = ax^2 + bx + c$$

takes on the value 0.

Let's consider how to turn this into an S function.

The Roots of a Quadratic Equation – Version 3

The following function version will deal with complex roots.

```
> roots <-
+   function(a, b, c)
+   {
+     delta <- b^2 - 4*a*c
+     if (delta < 0) delta <- delta+0i
+     (-b + c(1, -1) * sqrt(delta))/(2 * a)
+   }

> roots(1,0,-1)
[1]  1 -1

> roots(1,0,1)
```

```
[1] 0+1i 0-1i
```

Computing Normal Probabilities

Here is a direct implementation of summing the power series.

```
> Phi1 <-
+   function(z)
+   {
+     sum <- z
+     nfac <- 1
+     for (n in 1:30) {
+       nfac <- n * nfac
+       sum <- sum + (-1)^n * z ^ (2 * n + 1) /
+               (nfac * 2^n * (2 * n + 1))
+     }
+     0.5 + sum / sqrt(2 * pi)
+   }
```

The Roots of a Quadratic Equation – Version 4

To provide a final clean up the **`roots`** function we can use default values of the arguments.

```
> roots <-
+   function(a = 0, b = 0, c = 0)
+   {
+     delta <- b^2 - 4*a*c
+     if (delta < 0) delta <- as.complex(delta)
+     (-b + c(1, -1) * sqrt(delta))/(2 * a)
+   }

> roots(a = 1, c = 1)
[1] 0+1i 0-1i
```

This may or may not prove useful.

Computing Normal Probabilities

The function can be tested by comparing it with the built-in S function **`pnorm`**.

```
> Phi1(1.96)
[1] 0.9750021

> pnorm(1.96)
[1] 0.9750021
```

Computational Strategies

There are a two common ways of going about solving computational problems: *iteration* and *recursion*.

Examples of both follow.

Efficiency Considerations

The initial implementation of **`Phi1`** is very inefficient. Here is a more efficient version.

```
> Phi2 <-
+ function(z)
+ {
+   n2p1 <- 1;  mult <- -z^2/2
+   term <- z;  sum <- term
+   for (n in 1:30) {
+     n2p1 <- n2p1 + 2;
+     term <- term * mult / n
+     sum <- sum + term / n2p1
+   }
+   0.5 + sum / sqrt(2 * pi)
```

Normal Probabilities

Recall that for $z > 0$,

$$\Phi(z) = \frac{1}{2} + \frac{1}{\sqrt{2\pi}} \sum_{n=0}^{\infty} \frac{(-1)^n z^{2n+1}}{n!\,2^n(2n+1)}$$

We can use this as the basis for creating a computational procedure for computing normal probabilities.

The procedure works by *iteratively* summing the terms of the series.

Obviously, we can't sum all the way to infinity, but since the terms decrease in magnitude we can get a good approximation by summing a fixed number of them.

```
+ }
```

Measuring Efficiency

We can measure the difference in performance of these two functions using the **`unix.time`** function. We time 1000 calls because each call takes only a small amount of time.

```
> unix.time(for(i in 1:1000) Phi1(1))
[1] 0.44 0.00 0.47 0.00 0.00

> unix.time(for(i in 1:1000) Phi2(1))
[1] 0.14 0.00 0.14 0.00 0.00
```

The first number is the time taken in seconds.

The timing loop adds essentially nothing to the times, so the changes have more than tripled the performance.

Euclid's GCD Algorithm

As an illustration of recursion, we will look at Euclid's algorithm for determining the greatest common divisor of two integers.

The algorithm is based on the observations:

1. If k divides m and k divides n, then k must also divide m mod n (the remainder when m is divided by n).
2. If m mod $n = 0$, then n is the greatest common divisor of m and n.

[To see 1: Suppose that $m = d \times n + r$. Then $r = m - d \times n$ and so r must be divisible by k if m and n are.]

Euclid's GCD Algorithm

Here is an S function for the GCD.

```
> gcd <-
+   function(m, n)
+   if (n == 0) m else gcd(n, m %% n)

> gcd(100,12)
[1] 4
```

Note that **`gcd`** is defined in terms of itself. Such a function is said to be *recursive*.

> **recursion**: n., see recursion.
> – *The Devil's D.P. Dictionary*