

Introduction to R

Using R:

You can use R interactively, by typing commands into the terminal. However most of the time you should write your programs using a text editor (like WordPad on a Windows machine). Save your program as a text file with extension “.R.” Then run your program by typing

```
source('myprog.R')
```

into the R terminal (where `myprog.R` is the file name of your program). You will probably need to set the directory path to point to your home directory.

Scalar variables:

A scalar variable holds one numerical value. To create a scalar variable called `x` that holds the value 5.2, type

```
x <- 5.2
```

The variable `x` is now available for subsequent use, for example by continuing with the line

```
y <- x*2
```

which creates a variable called `y` holding the value 10.4.

Note that the value of `y` is determined based on the value of `x` at the time that `y` was created. So if we now change the value of `x` using

```
x <- 2
```

the value of `y` is still 10.4, not 4.

To see the current value of a variable, you can type the variable name into the terminal (i.e. just type `x` to see the value of `x`), or include a `print` statement in your program, e.g.

```
print(x)
```

Arithmetic and Boolean operators:

You can perform arithmetic and other standard mathematical operations using the familiar symbols. For basic arithmetic, +, -, *, and / give addition, subtraction, multiplication and division respectively. Exponentiation is given by ^.

You can include variables or constant values in arithmetic calculations, for example

```
x <- 5 * 3
y <- x / (1 + x^2)
```

Another arithmetic operator we will need is the remainder operator %%. The expression

$$a \% b$$

gives the remainder obtained when dividing b into a using integer arithmetic.

Boolean expressions evaluate to either true or false. For example,

$$3 + 2 < 5$$

is false, and

$$10 - 4 > 5$$

is true. The *and* operator && checks two expressions and is true only if both expressions are true. For example,

$$(3 < 5) \&\& (2 > 0)$$

is true, and

$$(2 < 3) \&\& (5 > 5)$$

is false. The *or* operator || is true if at least one of the expressions surrounding it is true. For example,

$$(3 < 5) || (2 > 3)$$

is true, and

$$(2 < 1) \ || \ (5 > 5)$$

is false. The *not* operator `!` evaluates to true for false statements and false for true statements. For example

$$!(2 < 1)$$

is true and

$$!(3 < 6)$$

is false.

Boolean expressions can be combined, using parentheses to confirm precedence. For example,

$$((5 > 4) \ \&\& \ !(3 < 2)) \ || \ (6 > 7)$$

is true.

The final Boolean operator we will need is the *test equality* operator `==`. This operator returns either true or false based on whether the values on either side are numerically equal. For example,

$$6 == 4 + 2$$

is true. Be careful to use `==` and not `=` to test equality.

Standard mathematical functions:

Important higher mathematical functions include `log` (natural logarithm), `exp` (the exponential function, or inverse of the natural logarithm), and the standard trigonometric functions `sin`, `cos`, and `tan` (sine, cosine, and tangent). For example

```
x <- log(2)
```

```
y <- exp(x)
```

yields a value of 2 for the variable `y`.

R provides four rounding functions: `floor` rounds to $-\infty$, `ceiling` rounds to $+\infty$, `round` rounds to the nearest integer, and `trunc` rounds toward zero.

Vector and array variables:

A vector is essentially a list of values, and an array is essentially a table of values (remember that a scalar is always a single value). Don't be confused by the fact that there is another data structure in R called a "list," which is slightly different from a vector.

A variable in R can hold a vector or array value instead of a scalar value. There are several ways to create vector variables in R. To construct a "literal" vector containing specified values, use the `c(...)` construct, as follows.

```
x <- c(10,-1,43.5)
```

The previous code produces a vector variable called `x` containing the values 10, -1 and 43.5, in that order. Another way to create a vector variable is by using the `seq` function to create a regular sequence of values. For example,

```
z <- seq(3, 8)
```

creates a vector variable called `z` that contains the values 3, 4, 5, 6, 7, 8. The `seq` function is quite flexible. You can also use

```
z <- seq(8, 3)
```

to create the vector of values 8, 7, 6, 5, 4, 3, or

```
z <- seq(3, 8, 2)
```

to create the vector of values 3, 5, 7 (the final 2 is called the "by" parameter, it indicates in this case that every second value starting at 3 is taken).

A third way to create vectors in R is using the `array` function. One use of the `array` function is to repeat a value a given number of times. For example,

```
z <- array(3, 5)
```

constructs a vector of 5 consecutive 3's. You can also use the array function to string several copies of an array together end-to-end. For example

```
z <- array(c(3,5), 10)
```

creates the vector 3,5,3,5,3,5,3,5,3,5.

Vectors of the same size (dimension) can be operated on arithmetically, with the operations acting element-wise. For example,

```
x <- seq(3, 12, 2)
```

```
y <- seq(10, 6)
```

```
z <- x + y
```

calculates the vector sum

$$(3, 5, 7, 9, 11) + (10, 9, 8, 7, 6) = (13, 14, 15, 16, 17)$$

Multi-dimensional arrays can be constructed with the array function. We will mostly use 2-dimensional arrays (matrices). To create a 3×5 matrix with all entries set to 3, use

```
M <- array(3, c(3,5))
```

You can also take a vector and reshape it into an array, for example

```
V <- seq(1, 11, 2)
```

```
M <- array(V, c(3,2))
```

which yields the array

$$\begin{pmatrix} 1 & 7 \\ 3 & 9 \\ 5 & 11 \end{pmatrix}.$$

Note that the matrix is filled in column-wise, not row-wise.

Basic operations on vectors and arrays

The values in a vector can be summed using the `sum` function. For example,

```
A <- seq(1, 100, 2)
x <- sum(A)
```

calculates the sum of the odd integers between 1 and 100.

Frequently it is useful to count how many elements within a vector satisfy some condition. For example, if we wanted to know how many of the integers between 1 and 100 are divisible by 7, we could use

```
A <- seq(100)
B <- (A %% 7 == 0)
x <- sum(B)
```

The `A%% 7` statement constructs a vector of true/false values, with a true in each position of `B` where the corresponding value in `A` is evenly divisible by 7. Then `x` contains the number of trues within this vector. Note that when summing a Boolean vector, true values are counted as 1 and false values are counted as 0.

Many built-in R functions operate on vectors or arrays. For example, `sum` calculates the sum of all elements in a vector or an array, and `mean` calculates the average of all elements in a vector or an array. When working with a matrix, we often want, for example, the sum of each row, rather than the sum of the whole matrix. The `apply` function allows us to do this. For example, the following program constructs a 10×10 matrix, and places the row-wise means in `RM` and the column-wise means in `CM`.

```
M <- array(seq(100), c(10,10))
RM <- apply(M, 1, mean)
CM <- apply(M, 2, mean)
```

Note that the second argument of `apply` determines whether the function is applied to the rows (if it is 1), or the columns (if it is 2).

Element access and slicing

To access the 5th element of the vector V , use $V[5]$. To access the value in row 3, column 2 of a matrix M , use $M[3,2]$.

To access the entire third row of a matrix M use $M[3,]$. To access the entire second column of a matrix M use $M[,2]$.

To access the 2×3 submatrix spanning rows 3 and 4, and columns 5, 6 and 7 of a matrix M , use $M[3:4,5:7]$.

Loops:

Loops are used to calculate values that result by accumulating results over many steps. For example, suppose we want to sum the integers from 1 to 10. We could use the following:

```
x <- 0
for (i in 1:10)
{
  x <- x + i
}
```

In the above program, x is an *accumulator variable*, meaning that its value is repeatedly updated while the program runs. The `for` statement creates a loop in which the *looping variable* i takes on the values 1, 2, ..., 10 in sequence. For each of these values, the code inside the braces `{}` is executed (this code is called the *body* of the loop). Each execution of the loop body is called an *iteration*.

To clarify, we can add a `print` statement inside the loop body.

```
x <- 0
for (i in 1:10)
{
  x <- x + i
  print(c(i,x))
}
```

Run the above code. The output (to the screen) should be as follows.

```
[1] 1 1
[1] 2 3
[1] 3 6
[1] 4 10
[1] 5 15
[1] 6 21
[1] 7 28
[1] 8 36
[1] 9 45
[1] 10 55
```

Ignore the [1] for now. The first number of each pair is the value of i at a given iteration, and the next number is the value of x . The 3 (line 2, column 2) is $1 + 2 = 3$, the 6 (line 3, column 2) is $1 + 2 + 3 = 6$, and so on.

Note that when the expression $x+i$ is processed, first the current value of x is summed with i , then the value of x is replaced with the updated value.

Loops can run over any vector of values. For example,

```
x <- 1
for (v in c(3, 4, 7, 2))
{
  x <- x*v
}
```

calculates the product $3 \cdot 4 \cdot 7 \cdot 2 = 168$.

Loops can be nested. For example,

```

x <- 0
for (i in seq(4))
{
  for (j in seq(i))
  {
    x <- x+i*j
  }
}

```

calculates the following sum of products:

$$1 \cdot 1 + 2 \cdot 1 + 2 \cdot 2 + 3 \cdot 1 + 3 \cdot 2 + 3 \cdot 3 + 4 \cdot 1 + 4 \cdot 2 + 4 \cdot 3 + 4 \cdot 4 = 65.$$

The first factor in each summand is the value of the i variable and the second factor is the value of the j variable. Note that $j \leq i$ since the range of j is specified as `seq(i)`.

“while” loops

A `while` loop is used when it is not known ahead of time how many loop iterations are needed. For example, the following program calculates the partial sums of the harmonic series

$$\sum_{j=1}^n 1/j$$

until the partial sum exceeds 5. It is a fact that the harmonic series diverges:

$$\sum_{j=1}^{\infty} 1/j = \infty,$$

so eventually the partial sum must exceed any given constant.

```

n <- 1
x <- 0
while (x < 5)
{
  x <- x + 1/n
  n <- n+1
}

```

As with a `for` loop, the body of the `while` loop consists of the code contained in braces after the `while` keyword. Before the body of the `while` loop, an expression that can be tested to be either true or false (a Boolean expression), e.g. `x < 5`, is given in parentheses. This expression is checked before each iteration. If the expression is true, the loop proceeds with the next iteration. Otherwise it does not.

“if” statements within loops

An `if` block can be used to make a decision about what statements of a program get executed. For example,

```

y <- 7
if (y < 10) { x <- 2 }
else      { x <- 1 }

```

The value of `x` after executing the previous code is 2. This doesn't appear very useful, but `if` statements can be very useful inside a loop. For example,

```

A <- 0
B <- 0
for (k in (1:100))
{
  if (k %% 2 == 1) { A <- A + k }
  else            { B <- B + k }
}

```

places the sum of all odd integers between 1 and 100 in `A`, and places the sum of all even integers between 1 and 100 in `B`. Note that an integer is odd if and

only if its remainder upon division by 2 is 1.

The following demonstrates an even more complicated `if` block.

```
A <- 0
B <- 0
C <- 0
for (k in (1:100))
{
  if ((k %% 2 == 1) && (k < 50))      { A <- A + k }
  else if ((k %% 2 == 1) && (k >= 50)) { B <- B + k }
  else                                { C <- C + k }
}
```

The preceding program places the sum of odd integers between 1 and 49 in A, the sum of odd integers between 50 and 100 in B, and the sum of even integers between 1 and 100 in C. You may have as many `else if` statements as you like, and note that the final `else` is optional.

The “break” statement

A `break` statement is used to exit a loop when a certain condition is met. For example, the following program sums the integers between 1 and 49. Once `k` reaches 50, the iterations stop, even though `k` was initially specified to run up to 100.

```
x <- 0
for (k in seq(100))
{
  if (k == 50) { break }
  x <- x + k
}
```