A Brief Introduction to R

R-HTA in LMICs

Contents

Note: this basic introduction to R is derived from the freely available An Introduction to R document on the CRAN website.

R: an analogy to understanding what's going on 'under-the-R-hood'

In a general sense, the R language centres on assigning 'objects' specific 'values'. Although this is not entirely accurate, it is okay to think of the processing of R in that way. For example:

```
# here we assign the numeric value 2 to the object 'x'
x <- 2
```

Note the assignment operator `<-`, i.e., the thing that assigns the value to the object. You can also use `=` sign like in Excel - it doesn't make a difference and it is completely up to personal preference!

Vectors

So, R operates on named *data structures*. The simplest data structure in R is the numeric vector. A numeric vector is a single entity (think 'object' like `x`) consisting of a collection of ordered numbers. The mathematical analogy is a 'set'. Hence, we can think of the above code as assigning an *ordered collection of numbers* with length 1 to `x`.

```
# we can see the length of x using the following 'base' r function:
length(x)
```

```
## [1] 1
```

To expand on the above, we assign *several* ordered values to a single object using the *concatenate* function:

```
x <- c(5, 3, 1, 5.2, 2.45, 10.4, 9, 2.2)
# and by calling the object we can see what values are associated with that
# object:
print(x)
```

```
## [1]  5.00  3.00  1.00  5.20  2.45 10.40  9.00  2.20
```

```
# or simply
x
```

```
## [1]  5.00  3.00  1.00  5.20  2.45 10.40  9.00  2.20
```

However: when using basic vectors using the `c()` function, it is very important to use the same type of variable. If you use numeric values as well as string in a single *atomic* vector (i.e., vectors created using `c()`), this will coerce values to a single type. For example, when using a string and a numeric type in a vector, the vector will become a string vector and `R` will no longer recognise the vector as a numeric vector (i.e. you will not be able to perform numeric operations on the object anymore:

```
temp <- c(1, 2, "4", 5, "3 31")
temp * 2
```

```
## Error in temp * 2: non-numeric argument to binary operator
```

Logical Vectors

`R` also allows manipulation of logical quantities. The elements of a logical vector can have the values `TRUE`, `FALSE`, and `NA` (for "not available"). Note that Logical vectors are generated by conditions. For instance, using the previous example, we can see if the length of `x` is the same of `x`:

```r
# here we are just creating an object v and assigning values between 1:10
# randomly, using the base sample() function:
v <- sample(x = c(1:10), size = 5, replace = TRUE)
# are the lengths of x and v equal?
length(x) == length(v)
```

```
## [1] FALSE
```

```r
# are the lengths of x and v not equal? (in R we use != operator to identifying
# whether an object or value is not equal to another)
length(x) != length(v)
```

```
## [1] TRUE
```

```r
# is the length of v greater than/less than/greater or equal to/less than or
# equal to v? (in R we use >, <, >=, <=, like in maths)
length(x) > length(v)
```

```
## [1] TRUE
```

```r
length(x) < length(v)
```

```
## [1] FALSE
```

```r
length(x) >= length(v)
```

```
## [1] TRUE
```

```r
length(x) <= length(v)
```

```
## [1] FALSE
```

Lists

Lists are a type of vector. However, unlike basic vectors (referred to as atomic vectors in R), Lists - created using the `list()` function in `R` - can store an type of values. So, unlike atomic vectors using the `c()` function, Lists will not coerce your values to a single type:

```r
list_example <- list(1:3, "a", c(TRUE, FALSE, TRUE), c(2.3, 5.9))
list_example
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] "a"
##
## [[3]]
## [1]  TRUE FALSE  TRUE
##
## [[4]]
## [1] 2.3 5.9
```

You can intuitively think of a list as an object storing multiple objects. Hence, each object within the list is stored as a separate 'object'.

Manipulating Objects in General

Once we have objects stored in memory, we can further manipulate all the values we assigned to a object. For instance, let's calculate the inverse of our values assigned to `x` above, i.e., $\frac{1}{x}$:

```
1/x
```

```
## [1] 0.20000000 0.33333333 1.00000000 0.19230769 0.40816327 0.09615385 0.11111111
## [8] 0.45454545
```

We can also mix this with logical operators to assign values ≤ 1 for example:

```
1/x[1/x <= 1]
```

```
## [1] 0.20000000 0.33333333 1.00000000 0.19230769 0.40816327 0.09615385 0.11111111
## [8] 0.45454545
```

Note that the example above uses 'indexing'. Indexing is a way of 'accessing' the elements of an object. For instance, if we wanted to access specific values in the object `x`, we do the following:

```
# the third value of x:
x[3]
```

```
## [1] 1
```

```
# the third, fourth, and fifth value of x:
x[c(3, 4, 5)]
```

```
## [1] 1.00 5.20 2.45
```

```
# ... note the concatenate function to specify multiple values!
```

However, see how in both examples we did not assign the output values to a *new* object and so the outputs are *not* stored in memory. We cannot recall the result without rerunning the code! Every time we want specific values from a specific object, which we performed an operation on, we have to carefully remember where the code is in our project. This isn't very helpful!

This may not seem like a problem here but it is much easier to assign values to objects when we are dealing with large models which include matrices and arrays, like Markov models! Assigning large outputs with thousands of values to single objects can thus be very helpful.

Vectorisation

Now that we have a basic understanding of objects and vectors in `R` we are going to learn how to utilise *vector arithmetic*. The neat thing is that shorter vectors in the expression are recycled as often as need be until they match the length of the longest vector. Notationally this is like $y_i \times x_i$. So, let's set up the following simple equation,

$$y_i = \frac{1}{x_i} + cos(\pi)$$

and translate it into `R`:

```r
# in r this is coded as:
y <- (1 / x) * cos(pi)
# and now that we have an object stored in memory, we can recall it without
# running the code (which is very handy when we have lots of code!). Note, the
# matrix function is being here to just make it look neat. We will touch on
# matrix operations shortly.
matrix(data = y, nrow = 4)
```

```
##            [,1]        [,2]
## [1,] -0.2000000 -0.40816327
## [2,] -0.3333333 -0.09615385
## [3,] -1.0000000 -0.11111111
## [4,] -0.1923077 -0.45454545
```

Note how we don't have to be explicit that the object `x` has several values. In other words, we don't have to *explicitly* iterate over `x_{i}`. In this case, however, it is *very* important to make sure that both objects have the same length. In the example above, it was acceptable to vectorise since we assigned values to a *new* object and so the new object `y` was assigned the dimensional characteristics of `x`! But what happens if we take the product of `x` and a shorter object `z`?

```r
# create object
z <- c(1, 2, 3)
# product of x * z?
x * z
```

```
## Warning in x * z: longer object length is not a multiple of shorter object
## length
```

```
## [1]  5.0  6.0  3.0  5.2  4.9 31.2  9.0  4.4
```

R only multiplies the first three values!. So, be careful. Vectorisation is much more efficient but you must think about what you are trying to achieve and the objects you are working with.

For Loops

Very simply speaking. For loops are used to iterate over a sequence. First consider the form of a `for loop` below:

```
for (name in expr_1) expr_2
```

The `for loop` above repeatedly evaluates the object `expr_2` as `name` ranges ('loops') through the values in the vector result of `expr_1`. We can use a `for loop` to resolve the `x` × `z` issue discussed above.

```
# we first create a matrix object with `x` rows and `z` columns
g <- matrix(data = NA, nrow = length(x), ncol = length(z))
# then we iterate over the rows and columns, multiplying all the values of `x`
# with the j'th column of `z`. This means that the whole vector of `x` is
# multiplied by a single value of `z` until all values of `z` of been iterated
# over `x`.
for (i in 1:length(x)) {
 for (j in 1:length(z)) {
  g[i, j] <- x[i] * z[j]
 }
}
```

It is important to know that `for loops` are used extensively for Markov modelling, since Markov models use matrices and arrays!

You can try it for yourself analytically by multiplying `x` by a value of `z`, to help get a sense of how the operation is taking place. For example:

```
print(z)
```

```
## [1] 1 2 3
```

```
# then multiply by the second value of z
x * z[2]
```

```
## [1] 10.0  6.0  2.0 10.4  4.9 20.8 18.0  4.4
```

```
# or the third value of z
x * z[3]
```

```
## [1] 15.00  9.00  3.00 15.60  7.35 31.20 27.00  6.60
```

Do these match the corresponding values of the matrix?

There is an even simpler solution! Since vectorised operations work element-wise on matrices, we can manipulate `x` into a matrix object and then multiply by `z`

```
x_matrix <- matrix(data = x, nrow = length(x), ncol = length(z))
# and then, et voila!
x_matrix * z
```

```
##      [,1]  [,2]  [,3]
## [1,]  5.0 15.00 10.00
## [2,]  6.0  3.00  9.00
## [3,]  3.0  2.00  1.00
## [4,]  5.2 15.60 10.40
## [5,]  4.9  2.45  7.35
## [6,] 31.2 20.80 10.40
## [7,]  9.0 27.00 18.00
## [8,]  4.4  2.20  6.60
```

Again, it is still important to make sure that your data are being manipulated as intended.

Matrix Multiplication

Lastly, we will cover matrix multiplication in R. For those who don't know a matrix is a rectangular array or table of numbers, symbols, or expressions, arranged in rows and columns, which is used to represent a mathematical object or a property of such an object. Have you worked with a Markov model in Excel? Well, the structure of the model is just a matrix in spreadsheet form!

Let's create a matrix of samples and see what happens when we multiply it be a set of values for each column:

```r
# create matrix
matrix_temp <- matrix(data = rnorm(n = 10*5, mean = 0, sd = 1),
                      nrow = 10, ncol = 5)
# print results
matrix_temp
```

```
##              [,1]       [,2]       [,3]         [,4]        [,5]
##  [1,] -1.32942408 -0.9761526  1.8800824  1.917828233  0.68486567
##  [2,] -0.73136643 -0.9151047 -1.9131397 -0.178588868  0.03405937
##  [3,] -0.47436045  0.0997057 -0.5873488  1.142120381 -0.53357298
##  [4,] -0.38187346  0.9537538 -0.1135330  1.348841702 -0.09511842
##  [5,] -2.14341240  0.5378462  1.0178578  0.311685246  1.27941716
##  [6,]  0.27268909  0.5059059 -1.1970095 -1.041695022  0.90155159
##  [7,]  1.47357608  0.6621967 -1.0144452 -0.802418426 -0.17603613
##  [8,] -0.40923781 -0.5770716  1.0104905  0.009892769  0.73539455
##  [9,] -0.07268481  0.6080826  0.8876542  1.045995799  0.94862475
## [10,] -0.42306304 -1.0623952 -0.7138238 -0.070780404  0.69868263
```

```r
# create a vector of values:
vector_eg <- runif(n = 5, min = 0, max = 1)
# create object to store output
output_temp <- matrix(data = NA, nrow = 10, ncol = 5)
# iterate for-loop over nrows by ncols
for (i in 1:nrow((matrix_temp))) {
 output_temp[i, ] <- matrix_temp[i, ] %*% vector_eg
}
```

Have fun coding and hopefully this helps you throughout the tutorial.