A Brief Introduction to R: 2024 Tutorial

R-HTA in LMICs

Table of contents

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

R: 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 what R does to remember objects in that way. For example:

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

Note the assignment operator `<-`, i.e., it is the thing that assigns the value to the object. You can also use `=` sign like in Excel or Python - it doesn't make a difference and it is completely up to personal preference! But, what you are essentially doing is telling `R` to remember that the value `2` is associated with the object `x`. So, whenever you call `x`, this means you are asking `R` to please recall the object and interpret the data or instructions that were assigned to that object:

```r
# so, we call `x`, and now every time we do that, the computer
# tells us that this `object` has a numeric value of 2.
x
```

```
[1] 2
```

You can also explicity ask R to recall the object using the `print()` function:

```r
print(x)
```

```
[1] 2
```

Vectors

So, R operates on named *data structures*. Structures can be thought of as a group of 'things' - these 'things' can be similar or different. A simple data structure is a vector, which groups things of the same kind. For example, a collection of numbers, e.g., 1, 2, 3, 4, and so on. Such a data structure with a single number value is called a numeric vector in `R`. A numeric vector is a single entity (think 'object' like `x` above) 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`.

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

```
[1] 1
```

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

```r
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
```

```r
# 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 data. If you use numeric values as well as string (character) values in a single *atomic* vector (i.e., vectors created using `c()`), this will coerce, i.e., force, values to change 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 any values as numeric:

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

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

The above warning is telling us that we no longer have numeric values in our vector!

Logical Vectors

R also allows manipulation of logical values. The elements of a logical vector can have the values TRUE, FALSE, and NA (for "not available"). 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 any type of values. So, unlike atomic vectors using the `c()` function, Lists will not coerce your values to a single data 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. A list is like a neat file cabinet, where you can store many different things in each drawer and still access each drawer without letting the contents becoming mixed up.

Manipulating Objects

Once we have objects stored in the computer's memory, we can manipulate all the values we assigned to a object. For instance, let's manipulate the values assigned to `x` above:

```
1/x
```

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

Here, we are simply saying take each value and make it $\frac{1}{x}$.

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

```
x # there's only 1 value <= 1...
```

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

```
x[x <= 1] # and our condition returns what we expect...
```

```
[1] 1
```

Note how the above example 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
```

```
# ... and 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 is much easier to work with. You can think of it like storing a table of data in an Excel spreadsheet. But, instead of always having it visible, you tell `R` when it access it when it is needed - one of the many reasons `R` is so much more efficient than Excel.

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 until they match the length of the longest vector. So, let's set up the following simple equation,

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

and note that $\cos(\pi)$ is a single value. Let's translate it into `R`:

```r
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
```

While this makes life a lot easier, and avoids the need for `for loops`, there is a disadvantage.

Observe 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, it worked as intended as each value in `x` was multiplied by `cos(pi)`. However, it is *very* important to make sure that both objects have the same length. Again, in the example above it was acceptable to vectorise this calculation since we assigned values to a *new* object and so the new object `y` was assigned the same 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 very careful. Vectorisation is much more efficient but you must think about what you are trying to achieve and the objects you are working with applying these short cuts. This is why understanding how to work with `for loops` in `R` is important.

For Loops

Very simply speaking. `for loops` are used to iterate over an index 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!

To get an intuitive feel for `for loop`'s, you can try it for yourself iteratively multiplying `x` by a value of `z`. This will help get a sense of how the operation is taking place inside the for loop. Note how `x` is the *outer* loop and `z` is the *inner* loop, i.e., you multiply each value of `x` with every value of `z` before moving to the next `x` value in the sequence. For example:

```
print(z)
```

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

```
# the first for loop indexes x, so we start with the first value of x...
x[1]
```

```
[1] 5
```

```r
# multiply x[1]...
# by the first value of z
x[1] * z[1]
```

```
[1] 5
```

```r
# by the second value of z
x[1] * z[2]
```

```
[1] 10
```

```r
# and then the third value of z
x[1] * z[3]
```

```
[1] 15
```

```r
# which is the same as the first column of g from the for loop
g[1, ]
```

```
[1]  5 10 15
```

```r
# then the second value of x...
x[2]
```

```
[1] 3
```

```r
# multiply x[2]...
# by the first value of z
x[2] * z[1]
```

```
[1] 3
```

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

```
[1] 6
```

```r
# and then the third value of z
x[2] * z[3]
```

```
[1] 9
```

```r
# which is the same as the second column of g from the for loop
g[2, ]
```

```
[1] 3 6 9
```

```r
# and so on, until you have looped through each value of
# x multiplied by each value of z!
```

Do these match the corresponding values of the matrix?

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

```r
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 very 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:

```
# 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,]  0.5363100 -2.1320124  1.2259567  0.39296166  1.74507342
 [2,] -0.2388116 -0.4918589 -0.5419172 -0.92933944  0.69852747
 [3,]  0.5255316  0.2670429 -0.9515414  1.32597453  0.55807231
 [4,] -0.5194807 -0.2824360  1.2698900 -2.31881519 -0.19228482
 [5,]  0.3184393  0.1843534  3.9164062  0.16737631  0.23801032
 [6,]  0.2976156 -1.8684573  2.2472481  1.09275705  0.31272040
 [7,] -0.8204398 -1.9179426 -0.7950841 -0.05790273 -0.05597528
 [8,]  1.9318867  0.2463202 -0.3301649  0.42875079 -0.85799654
 [9,] -0.5303097  0.2803425  0.2221967  0.85721345 -1.10586187
[10,]  1.3848158 -0.1329057  0.3308981  0.68329802 -0.87143828
```

```
# 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!