Processing and Analysis of Biological Data

Appendix

Øystein H. Opedal, Department of Biology, Lund University

2024-10-31

Working in R: files, folders and projects

We can make our life easier by learning right from the start how to keep a tidy workflow in R. I recommend learning about `R` "projects", which become essential once you are working on many parallel tasks/projects. For example, I have an `R` project for these course notes. A project is associated to a folder that holds all the project files, including data files, R-code files (analyses, functions), saved results (e.g. model outputs), and saved figures.

One advantage of working within a project is that we don't need to worry about our working directories, it will be automatically set to the project folder. If you also develop a standard naming convention for folders, if will be easy to remember for example that the input data are kept in the folder 'data', and the figures are saved in the folder 'figures'. This is easier than keeping the files in folders like `C:/Users/Øystein/My documents/stuff/Studies/Lund/Coursework/Master/Stats/R` (I have seen even worse!).

To start a project, simply click `file/New Project`. You will see that there is an option to directly associate the project to a system for version control, such as GitHub (see below).

Another common "mistake" is to do way too many things within one R file. When working with "real" analyses, it is good style to keep for example one `R` file for formatting the data (which you of course will not do in Excel!) and then writing clean files for the analyses. These could include one file for performing e.g. model fitting, and another to produce summaries and graphics based on the results. For simpler analyses this can of course all be done within one file, but it is good to start practicing. Once you have your own custom-written functions, separate them from the main workflow and load them as needed using the `source` function.

Formatting and importing data Importing datafiles into `R` can be a real hassle in the start. In RStudio there is an "Import Dataset" button, but this often involves excessive clicking and somehow defeats the purpose of a code-based environment. It is better to figure out a file format that works, and then we have the code and can import the data in a second each time we open the `R` file.

Good formats for importing into `R` are .csv and .txt. I generally enter the data in Excel, and then save the file in either of those formats. A couple of points about organizing data in Excel:

1. Avoid empty cells, put `NA` instead. There are ways to make `R` insert `NA`s in empty cells, but it is always safe to add them already during data entry.
2. Avoid spaces, use underscore (_) instead. This will avoid problems where `R`understands 'body size' as two variables instead of one (body_size). Some also like to keep the units in the variable names, i.e. 'body_size_kg'.

`R` has functions to import many kinds of data. I typically use `read.table` for text files, and `read.csv` for .csv files. On Windows, I often need to use `read.csv2` instead, for some reason. Pay attention to the arguments of these functions, especially things like `sep`, `dec`, and `header` that tells how the data specify string separation, decimal places, and whether there are variable names (column headers) in the data file.

If you want to import Excel files directly, the `readxl` package seems to work well. It allows you to specify a specific sheet within an excel file, which is sometimes useful, and also seems to work well for inserting `NA`s for empty cells which could be good if you have huge data with many empty cells. But again, find a method that works for you and you will be fine.

Reproducibility and version control All of this is directly related to the issue of reproducibility. More and more journals now require code archiving, and by working in a tidy way we are ready to submit our code at the time of submitting the paper.

To take this one step further, we can use a system for version control like GitHub. This also provides us with backup in case our computer dies, and an easy way to publish the code (and data, although this is apparently not the recommended way of publishing data).

For this course, for example, I have made a public GitHub repository available at https://github.com/oysteiop/BIOS14_QuantitativeAnalysis

Simple programming

As soon as analyses become a little more complex, some programming skills are very helpful for working in `R`. First, `R` in function-based, and we can write our own functions that we can execute over a number of cases. Functions can for example produce a specific plot, fit a specific kind of model, or format a dataframe. It is good practice to use functions whenever we need to repeat a specific procedure multiple times. So the earlier you start practicing, the better.

The perhaps most common programming operation during data analyses is *for*-loops. A *for*-loop performs an operation *for* each element of an index vector.

```r
x = seq(1, 20, 2)
x
```

```
##  [1]  1  3  5  7  9 11 13 15 17 19
```

```r
out = NULL
for(i in 1:length(x)){
  out[i] = x[i]*2
}
out
```

```
##  [1]  2  6 10 14 18 22 26 30 34 38
```

Sometimes we want to perform an operation only under certain conditions. We can then use an *if* argument.

```r
out = NULL
for(i in 1:length(x)){
  if(i>10){
    out[i] = x[i]*2
  }
  else{
    out[i] = 0
  }
}
out
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0
```

A shortcut for *if* and *else* is the function `ifelse`.

```
ifelse(x>25, 1, 0)
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0
```

A final useful programming argument is *while*, which makes a process continue until a specific condition is reached. For example, this program draws values of 1-10 until the value 21 is reached.

```
out = 0
  while(out<21){
    out = out + floor(runif(1, 1, 10))
    print(out)
  }
```

```
## [1] 2
## [1] 6
## [1] 8
## [1] 17
## [1] 26
```

Writing functions

`R`-functions takes arguments, performs operations and, often, returns some results. Base `R` does not come with a built-in function to compute standard errors, so let's make out own.

```
computeSE = function(data){
  n = sum(!is.na(data), na.rm=T)
  SE = sqrt(var(data, na.rm=T)/n)
  return(SE)
}
```

Now let's feed some data to our function.

```
set.seed(1)
x = rnorm(200, 10, 2)
computeSE(x)
```

```
## [1] 0.1313942
```

Some functions can get very complex, with many arguments and running over hundreds of lines of code. This can seem overwhelming, but it is important to keep in mind that functions are generally not written in one go, it is generally best to start from "inside" the function, with a single line that performs a single operation, and make sure it works before adding complexity.