Introduction to R/RStudio

Alexander Ploner

2020-11-19

This document

This is a short introduction to working with R/RStudio, covering the basics of working with the commandline, data types and data structures, and organising a simple analysis workflow. No previous exposure to R or RStudio is assumed. I strongly recommend that as a first-time reader, you work through the examples shown in the text at a computer.

The latest version of this document, as well as the accompanying data files and example scripts, is available from Github.[1]

[1] github.com/alexploner/QuickIntro2R

[2] creativecommons.org/licenses/by-nc-sa/4.0/

Background

R is a free software environment for statistical computing and graphics.

Specifically, R is

1. a program for statistical data analysis. Compared to software with similar functionality, R has a strong focus on interactive analysis (unlike the typical batch analysis in SAS) at a command prompt (unlike the graphical user interface of SPSS).
2. a general-purpose programming language. R is an interpreted language (like Python, and unlike C or C++). Essentially, any kind of data handling or -analysis method can be implemented in R. This also means that existing methods can be combined into complex analysis workflows through R script files.
3. the infrastructure for a large number of add-on packages that implement methods in statistics, data processing, bioinformatics, machine learning and many other disciplines.[3] [4]

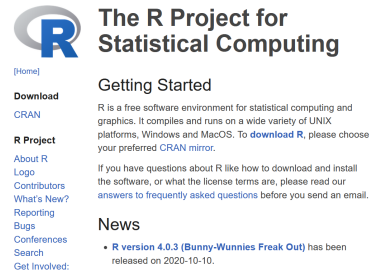

Figure 1: R homepage `https://www.r-project.org/`

[3] cran.r-project.org/web/packages

[4] www.bioconductor.org/packages

RStudio is an integrated development environment built on top of R. It provides a graphical user interface around the R console, including a code editor, a file browser, a help viewer, a tab for local data management and other utilities. In principle, all of this functionality is also available in R via commands and add-on packages, but the clean structure and tight integration of RStudio makes the learning process easier for many beginners. RStudio has an additional focus on data science and related software development, some aspects of which can be useful for data analysis (e.g. version control integration via git), whereas others are more developer-oriented (e.g. javascript-integration via shiny).

Both R and RStudio are open source software, and freely available online[5], but where R is community-developed and non-profit[6], RStudio is the product of a commercial entity[7].

[5] Installers can be run with defaults.

[6] www.r-project.org/foundation/

[7] linkedin.com/company/rstudio-inc

The RStudio program window is typically split into up to four different quadrants or *panes*. Figure 1 shows a typical configuration:

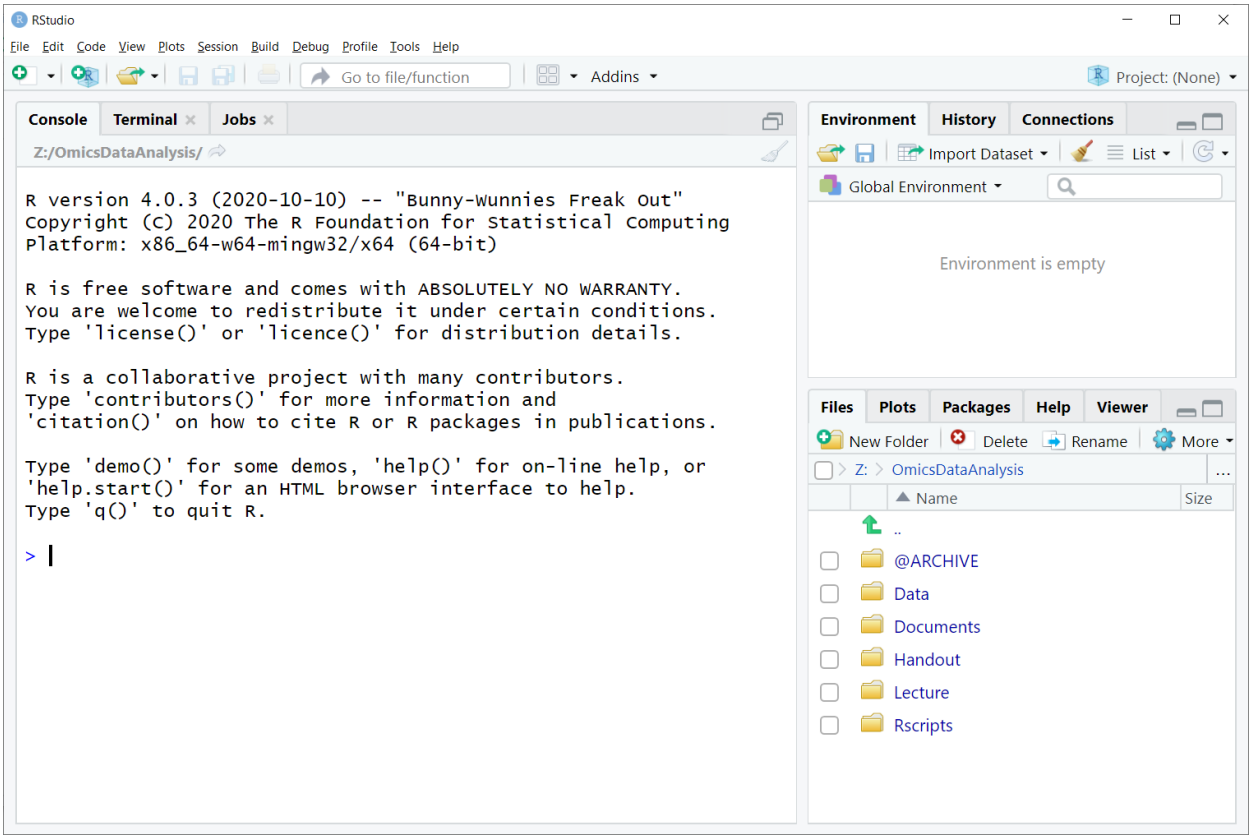

Figure 2: RStudio running R 4.0.3

- The large pane to the left showing the R start-up message is the **console**. This is the main window, where the user types commands, and numerical results are displayed. The RStudio console is identical in function and appearance to the console window in the underlying R software.
- The smaller top right pane shows the **Environment** tab, which displays all currently defined objects (like data sets or analysis results) in a the current R session. Figure 1 shows RStudio at the start of a session, so objects are defined yet, and the environment is empty.
- In the bottom right panel, we see the **Files** tab, which lists the files in the current working directory. This pane works as an internal file browser, where the user can travel through the directory hierarchy on the hard disk and inspect and manipulate files.

Note that the two smaller panes are *tabbed*, i.e. other functionality is available in separate tabs (like *History* on the top and *Packages* and *Plots* on the bottom). This will be discussed in detail below.

Working interactively

At the prompt

The basic workflow for interactive data analysis looks like this:

1. Start RStudio
2. In the console pane (at the command prompt >), a steady loop:
   - Type input (**expression**, see below), hit return
   - R evaluates input and displays result (text output, plot)
   - Based on results, provide new input (expression)
3. Quit (and save, if necessary)

Note: you can browse through previous input via arrow keys, edit and re-use it. This minimzes typing and allows for rapid incremental refinement of the analysis.

Simple expressions

At the most basic level, R can work as a glorified calculator for numerical expressions that consist simply of numbers and basic arithmetic operations:[8]

[8] Text after the prompt > is typed by you, text below, usually starting with [1], is output generated by R.

```
> 3
[1] 3
> 3+4*2
[1] 11
```

R also uses character expressions (strings), e.g. to keep track of group identifiers in data or to define labels for plotting. Character values are enclosed in single ' or double " quotation marks:

```
> "My name is"
[1] "My name is"
> 'Earl'
[1] "Earl"
```

R also uses logical (or boolean) expressions, e.g. to select subsets of data or to test for conditions. These expressions have only two possible values, `TRUE` and `FALSE`. We can use comparison operations on numbers or characters (as well as other data types) to create logical expressions:

```
> TRUE
[1] TRUE
> 3 < 4
[1] TRUE
> "Earl" == "earl"
[1] FALSE
```

Functions

Everything interesting in R is done via *functions*. These are pre-packaged pieces of code that can perform a specific task. Functions have names (generally), and zero, one or more *arguments*, i.e. slots for pieces of information

that the function either operates upon (data), or that tell the function what to do (flags, parameters).

In order to run (execute) a function, you type its name at the console, followed by a pair of parentheses with the arguments for what you want the function to do, separated by commas.

EXAMPLE: FUNCTIONS

The function `sqrt` calculates the square root. It takes one numerical argument, and returns one numerical value, which is the desired result:

```
> sqrt(4)
[1] 2
```

The function `paste` takes two or more character arguments, and returns one character result, which is just the joined arguments, separated by an empty space:

```
> paste("My name is", "Earl")
[1] "My name is Earl"
```

The function `log` calculates by default the natural logarithm (for base e). Like the square root, it takes one numerical argument, and returns the numerical result:

```
> log(10)
[1] 2.302585
```

However, `log` has a second argument, called `base`, which you can use to specify a different base for the logarithm. So you can calculate the common logarithm (for base 10) and the binary logarithm (for base 2) like this:

```
> log(10, base = 10)
[1] 1
> log(10, base = 2)
[1] 3.321928
```

The function `getwd` takes no arguments, and returns a string with the current working directory, which is the path to the directory (folder) where R by default reads and writes files[9]:

```
> getwd()
[1] "Z:/@ownCloudKI/OmicsDataAnalysis"
```

Note that you still need to add an (empty) pair of parentheses to indicate that you want the function to run, even though no argument is given.

The functions in the examples so far do their job by returning a numerical or string result. This is the most common way to work with functions in R: you feed some data to a function, which returns a result, which you then can feed into another function, and so on. In some situations however, you want a function to simply *do* something, e.g. create a plot or write data to a file[10] An example for this kind of function is `browseURL`, which takes a character

[9] Note: R uses slashes (`/`) not backslashes (`\\`) for separating file- and folder names in a path.

[10] This is generally described as the function being run for its *side effect*. This is somewhat misleading, as this is often the only thing you are interested in.

argument with the URL we want to visit, and opens the default browser on your system pointed at that address:

```
> browseURL("https://cran.r-project.org")
```

Finding functions

The functions in the examples above are all part of base R, and are immediately available at the command prompt after starting R/RStudio. Most other functions however live in specialized add-on packages, which need to be loaded before you can use their functions; see the section on R packages below for details.

Learning how to use functions, and what functions to use in order to get a task done, is the hardest part when starting with R. To help you with this, R and its add-on packages have large amounts of documentation available. In the simplest case, when you remember the name of the function, you can just type

```
> ?t.test
```

at the command line; this will open the corresponding documentation in a separate window.

If you only kind of remember the name of the function, say something with "test", you can type

```
> apropos("test")
```

to list all currently available functions with "test" in their name.

A very useful feature is also *tab-expansion*: when typing the function name in RStudio, hitting the tabulator-key will open a context menu at the cursor that lists all available function names that complete the already typed text; this list is scrollable, reactive (i.e. more typing narrows the list), and displays the help text for each entry in a second-level pop-up. Furthermore, once you have typed or selected the function name with the trailing parentheses, you can move the cursor between these parentheses and hit the tabulator-key again to get a scrollable, reactive list of the function arguments (with documentation).

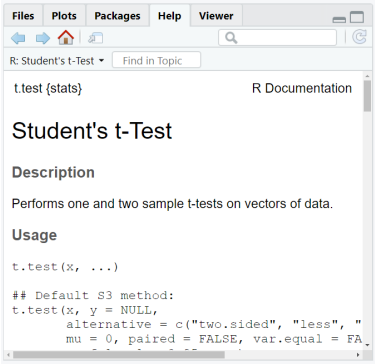

Figure 3: RStudio Help tab

Note: `apropos` is again just an R function, which takes a character expression with the string of interest as argument, and returns a series of character expressions, which are the names of the functions that match the string.

Both function- and argument expansion also work in plain R, though you may have to hit the tab-key twice, and only the names (and not the help) are displayed.

Objects

While the arrow-key navigation of previous expressions is helpful, we do not want to re-calcuate the same expression repeatedly. Therefore, R allows you to store results as *objects* under a freely chosen name: R saves a copy of the specified result in its memory, and from there on, you can use the name to refer to the result.

To assign a *value* to a chosen *name* in R, you can use the operator <-:[11]

name <- *value*

Note that there are some limitations for object names: they can only contain letters, numbers, dots . and underscores _, and they cannot start with a number.[12]

[11] Note: you can also use a sinle equal sign, as in *name* = *value*, though this is less common and potentially confusing

[12] It is also generally a good idea to avoid non-ASCII characters like ä, å etc.

Example: objects

Let's define two objects called `a` and `A` that both store numerical results:

```
> a <- 3
> A <- 3 + 1
```

In order to see the content of an object, you just type its name at the command prompt - to R, this is just another type of expression:[13]

[13] Also, R clearly distinguishes between small `a` and capital `A` as two different objects, with different content.

```
> a
[1] 3
> A
[1] 4
```

In the same way, you can define objects that store the value of a character expression. Also, it can be helpful to have longer, more informative names for objects:

```
> myWorkDir <- getwd()
> myWorkDir
[1] "Z:/@ownCloudKI/OmicsDataAnalysis"
```

Note that this uses the function `getwd` from the previous example to return an informative character expression. The value of this function call is saved under the chosen object name, exactly as for the numerical expressions above.

What has happened with the previously defined objects? Nothing; they are around until further notice (see below), or until the user quits R. So you get the same value for object `a` as before:

```
> a
[1] 3
```

Managing objects

During a typical interactive session, you will generate a number of different objects, containing different types of data and results. In order to get an overview of what objects are currently defined (and therefore available for analysis and inspection), you can either check the Environment tab in RStudio, which shows a list of the currently defined objecsts, or run a command at the prompt:

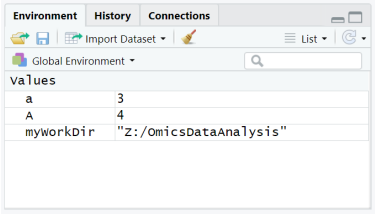

Figure 4: RStudio Environment tab

```
> ls()
[1] "a"          "A"          "myWorkDir"
```

Once an object is defined, it will stay defined and available at the command prompt for the rest of the R session. However, you can change the *content* of the object through a new assignment to the same name:

```
> a <- 3.18
> a
[1] 3.18
```

If you want to remove an object completely, so that its name is no longer defined, you can use the function `rm`:

The names `rm` (for remove) and `ls` (for list) are taken from corresponding Unix shell commands for files

```
> rm(a)
> ls()
[1] "A"          "myWorkDir"
> a
Error in eval(expr, envir, enclos): object 'a' not found
```

If not deleted, objects will stay in memory until the end of the R session; if not saved to disk when R shuts down, they will be lost (see below for shutting down R safely).

All together now: general expressions

As you may have foreseen, you can combine simple expressions (with arithemtic- or comparison operators), functions and objects in a completely natural and flexible manner.

EXAMPLE: GENERAL EXPRESSIONS

```
> a <- -1
> x <- 3*sqrt(16) - abs(a)
> x
[1] 11
```

Working with data

Basic data types

The data type of an expression determines what kind of information has been collected or measured. We have already seen examples for three important data types, namely `numerical`, `character` and `logical`, corresponding to numerical, discrete and binary data, respectively.

Here, I want to introduce a fourth data type, *factor*, for keeping track of discrete (grouping) data. This is similar to `character`, but has some advantages over `character` for statistical analysis and modelling. Data of other types (`numerical`, `character` etc.) can be converted manually to a `factor` using the function of the same name, but often this is done automatically when reading data into R from an external source like a file. I will show examples for both uses below.

Basic data structures

While a data type describes what a single item of information looks like, a *data structure* describes how multiple items of information can be arranged and used together. This is of course crucial for any data analysis with $n > 1$ samples, but has not been discussed so far.

Linear structure: vector

The simplest data structure for multiple observations is linear, presenting measurements in order from first to last. Statistically, this corresponds to a study where you collect information on a single variable and in a single population. In R, this structure is known as a *vector* (in other programming languages as a one-dimensional array).

We can construct a vector manually at the command line, by using the function `c` (for *combine*) and listing all elements of the vector as arguments:

```
> c(2.57, 3.14, 3.78, 1.90, 2.45)
[1] 2.57 3.14 3.78 1.90 2.45
```

R just displays the elements of the vector in the given order. The same general rules apply for vectors as for other expressions, so you can e.g. save a vector as an object under a freely chosen name:

```
> x <- c(2.57, 3.14, 3.78, 1.90, 2.45)
> x
[1] 2.57 3.14 3.78 1.90 2.45
```

You can also use the vector as argument to functions, e.g. for calculating basic descriptive statistics: function `mean` accepts a numerical vector with one or more elements, and returns a single number, the arithmetic mean of the elements of the argument. Function `sd` does the same for the standard deviation:

```
> mean(x)
[1] 2.768
> sd(x)
[1] 0.7169868
```

In the same way, you can define character vectors for keeping track of non-numeric observations for a set of subjects, e.g. identifiers or group labels. A possible grouping variable for the same five subjects as above could be

```
> grp <- c("case", "control", "control", "case", "control")
> grp
[1] "case"    "control" "control" "case"    "control"
```

Again, we can use functions to extract information from the vector, e.g. the frequency of the observed groups:

```
> table(grp)
grp
   case control
      2       3
```

As mentioned above, it is generally preferable to store grouping variables as factors instead of character strings, which is as easy as:

```
> grp2 <- factor(grp)
> grp2
[1] case    control control case    control
Levels: case control
```

`grp2` has of course the same information about the five subjects, but note the differences in display: no quotation marks around the vector entries, and the levels of the factor (the different values allowed) are displayed explicitly below the data.

Rectangular structure: data frame

In the example above, you have constructed two vectors with information about five subjects, one with numeric measurements and one with grouping information. In a situation like this, where more than one variable is measured on a group of subjects, it is extremely useful to be able to save all that information as one object in R (instead of as a collection of unrelated vectors). This is the motivation for the default structure for data analysis, the *data frame*.

A data frame is a rectangular arrangement of information, where each row corresponds to a single subject under study, and each column to a single variable that is measured. A simple way to build such a data frame is to use the function `data.frame`. For the example above with five subjects and two variables, this could be

```
> dat <- data.frame(grp2, x)
> dat
     grp2    x
1    case 2.57
2 control 3.14
3 control 3.78
4    case 1.90
5 control 2.45
```

The data is displayed arranged as rows and columns, with the names of the vectors used as column names and a running number as row name. Object `dat` now contains all necessary information for a basic analysis, and is the only object you have to keep track of from here on.

Importantly, a data frame is not a black box: you can still extract the information for individual processing. A simple way of extracting a variable is the $-notation:

```
> dat$grp
[1] case    control control case    control
Levels: case control
> table(dat$grp)

   case control
      2       3
> dat$x
[1] 2.57 3.14 3.78 1.90 2.45
> mean(dat$x)
[1] 2.768
```

In short, a data frame is the R implementation of the standard format for analysis data in statistical software, the data matrix, given as subject rows $\times$ variable columns.[14]

[14] Note that the convention is different in e.g. bioinformatics, where generally rows correspond to features (variables), and columns to samples (subjects). This is the source of much hilarious confusion.

Getting data into R

Building a data frame from hand at the command prompt is neither usual nor recommended. The most common way to import rectangular data into R for analysis is to read it from an external file. With add-on packages, R can read (and often write) a large number of formats, including data files from Stata or SAS as well as Microsoft Excel formats. Import functions of this type generally take as argument the name of the file to be read, and return as result a data frame with its content.[15]

In base R, `read.table` can read data from a wide range of delimited text formats.

The `Environment`-pane offers shortcuts for importing other types of common data files.

[15] Import functions also generally have a large number of optional arguments that allow fine control over how the data is read; see e.g. ?read.table

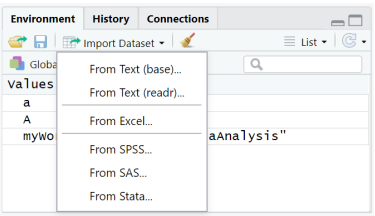

Figure 5: RStudio Environment with import menu

Example: qPCR data

This document comes with a data set `qPCR.txt`[16] that contains the results of a small qPCR experiment, where the expression of a target gene

[16] link?!

in treated *Arabidopsis thaliana* plants was compared to untreated controls plants, at four different concentrations, each with three replicates.[17] Consequently, the file contains 24 measurements of the relative expression level of the target gene compared to a reference gene (`DeltaCt`), together with the corresponding treamtent status (`Sample`) and concentration (`Con`); variables names are listed in the first row of the file.

[17] PMC1395339

We use `read.table` to import the data file, which we directly save as object `ex1`. The argument `header=TRUE` tells `read.table` to use the first row as column names, and the argument `stringsAsFactors = TRUE` to import the character coumn directly as a factor variable:

```
> ex1 <- read.table("Z:/OmicsDataAnalysis/Data/qPCR.txt",
+                   header = TRUE, stringsAsFactors = TRUE)
```

Inspecting data

I recommend some basic quality checks after each data import. We can use the function `head` to display the first few rows of the data:

```
> head(ex1)
   Sample Con DeltaCt
1 Control  10  3.3687
2 Control  10  3.4063
3 Control  10  3.5066
4 Control   2  4.5963
5 Control   2  3.7704
6 Control   2  3.4906
```

The data looks ok, and the column names were properly imported as variable names.

More formally, we can use the function `str` to look at the *structure* of the object `ex1`:

```
> str(ex1)
'data.frame':   24 obs. of  3 variables:
 $ Sample : Factor w/ 2 levels "Control","Treatment": 1 1 1 1 1
 $ Con    : num  10 10 10 2 2 2 0.4 0.4 0.4 0.08 ...
 $ DeltaCt: num  3.37 3.41 3.51 4.6 3.77 ...
```

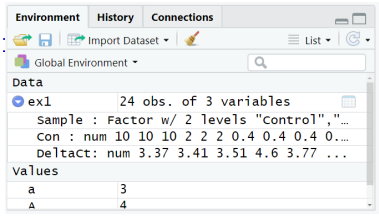

Figure 6: RStudio Environment tab also showing structure of ex1

This is quite informative: we learn that this is indeed a data frame with 24 rows and three columns, all of which have the right type and name, and the values appear to be correct, too.[18]

[18] And the grouping variable Sample was indeed automatically converted to a factor variable by read.table

Everything used in the small 5×2 toy example above works here, too, including the $-notation:

```
> ex1$Del
 [1] 3.3687 3.4063 3.5066 4.5963 3.7704 3.4906 3.3016 3.7572
 [9] 3.3607 3.5453 3.6461 3.9351 3.3345 2.9337 3.3349 2.5397
[17] 2.6615 2.8767 3.1472 3.2965 2.7206 3.0662 2.7814 2.7741
```

Note that we only have to type enough of the column name to uniquely identify the variable.[19]

[19] This is actually quite lazy, as both R and RStudio support tab-expansion for column names.

Workflow

Let's put all parts together to look at a standard situation: you have some data in a file, which you want to analyse; the results from this analysis are to be included in a report. For this exercise, let's use the qPCR data from before to demonstrate how this can be done in R.

I suggest breaking down the workflow as follows:

1. Prepare: put the data file(s) into a folder on the harddisk that is easy to find & access.
2. Start R/RStudio
3. Set the working directory, load the data
4. Run some descriptives: this includes calculating descriptive statistics like mean or standard deviation, as well as plotting the data. Descriptives can be a simple quality control of the data, or they may be included in the report.
5. Run the analysis: the main analysis of interest, generally involving some statistical inference (estimates, confidence intervals, p-values).
6. Export results: extract the numerical and graphical output you want to include in the report.
7. Shut down R/RStudio: this requires a decision whether to save the R objects generated during the workflow.

Finding & loading data

As demonstrated above, you can specify the full path to any file on the harddisk for reading it in via `read.table`. However, I recommend setting R's default directory, the *working directory*, right at the start of the analysis. This makes it easier to keep track of files that you generate while working on this specific data.

The working directory can be set at the command prompt:

```
> setwd("Z:/OmicsDataAnalysis/")
> dir()
[1] "@ARCHIVE"  "Data"      "Documents" "Handout"   "Lecture"   "Rscripts"
```

Note that my working directory has a number of sub-folders, which is generally a good idea if your project involves more than a handful of files.

After setting the working directory, all further file operations that do not explicity start at the root of the file system will be *relative* to the working directory. So in order to read the file `qPCR.txt` in sub-folder `Data`, it is sufficient to run[20]

```
> ex1 <- read.table("Data/qPCR.txt", header=TRUE,
+                   stringsAsFactors = TRUE)
> str(ex1)
```

[20] Another round of tab-expansions: once you have placed the cursor between the quotation marks, you can start to type the file name, and hit tab to show a list of available files in the working directory.

```
'data.frame':   24 obs. of  3 variables:
 $ Sample : Factor w/ 2 levels "Control","Treatment": 1 1 1 1 1 1 1 1 1 1 ...
 $ Con    : num  10 10 10 2 2 2 0.4 0.4 0.4 0.08 ...
 $ DeltaCt: num  3.37 3.41 3.51 4.6 3.77 ...
```

Alternatively, you can use the GUI to set the working directory: both R and RStudio have menu items for that (`File/Change dir` and `Session/Set working dir`, respectively). RStudio also offers the Files pane, where you can explore the directory tree interactively to set a working directory.

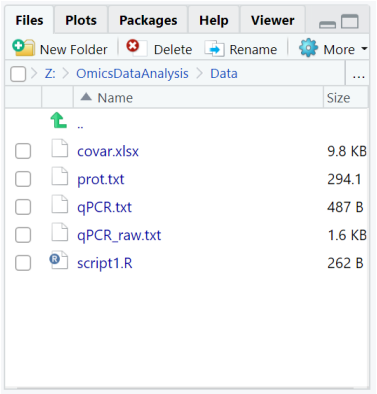

Figure 7: RStudio Files pane

Descriptive statistics

A useful function for generic descriptives is `summary`: if the argument is a numerical vector, it will display a number of location measures (minimum, maximum, mean, median and the first and third quartile); if the argument is either a factor vector, the function will return a frequency table. Coveniently, when the argument is a data frame, it will return a summary for each variable, depending on type:

```
> summary(ex1)
      Sample        Con           DeltaCt
 Control  :12   Min.   : 0.08   Min.   :2.540
 Treatment:12   1st Qu.: 0.32   1st Qu.:2.919
                Median : 1.20   Median :3.335
                Mean   : 3.12   Mean   :3.298
                3rd Qu.: 4.00   3rd Qu.:3.516
                Max.   :10.00   Max.   :4.596
```

This can be combined with other summary statistics of interest, like `sd` for standard deviations and `IQR` for the interquartile range.

Plots

R has a very rich set of graphical functions, including three independent systems for generating high-level plots. The examples shown here are from the oldest system, the *base graphics*, which is directly available in R.[21]

The function `hist` takes a numerical vector as argument and draws a histogram in the current plotting window:

```
> hist(ex1$DeltaCt)
```

[21] The other two systems are ggplot2 and lattice, respectively. Both require that an add-on package of the same name is installed and loaded.

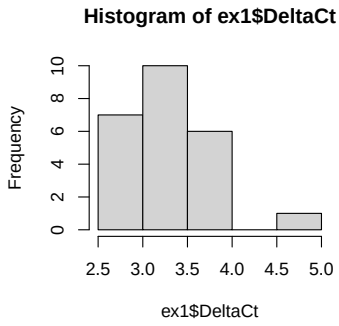

Figure 8: Histogram of DeltaCt

The function `boxplot` can draw boxplots of values of a given variable for separate groups of subjects, given by another variable. This draws separate boxplots for the relative strength of gene expression `DeltaCt` for the two groups of measurements as seen in grouping variable `Sample`:

```
> boxplot(DeltaCt ~ Sample, ex1)
```

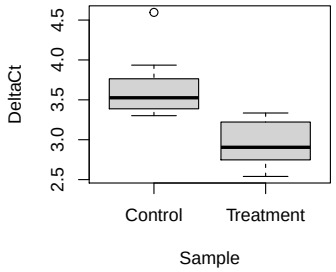

Figure 9: Boxplot of DeltaCt by group

The first argument in this function call is a *formula*, an R construction that is widely used for specifying e.g. plots and regression models; the tilde character ~ is best read "as a function of", so here "plot `DeltaCt` as a function of `Sample`" (where both variables are taken from the data frame `ex1`, as specified by the second argument).

With regard to the plot itself, we see that the expression levels are clearly higher among controls than among cases, and that there is a suspiciously large value among the controls - interesting for the report?

Base graphics look a rather old-fashioned, but are straightforward to create, and can be prettified in many ways, using titles, labels, colors, fonts etc., all of which goes beyond the scope of this document.

Statistical tests

For the current example, a simple t-test for the null hypothesis that the mean gene expression level (averaged across all concentrations) is the same for both treated plants and untreated controls seems a reasonable sort of inference. This can be done via the function `t.test`:

```
> t.test(DeltaCt ~ Sample, data = ex1)

    Welch Two Sample t-test

data:  DeltaCt by Sample
t = 5.2577, df = 20.684, p-value = 3.429e-05
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 0.413698 0.955952
sample estimates:
  mean in group Control mean in group Treatment
               3.640408                2.955583
```

The function call and the result for this test are very similar to most other basic tests, so let's have a look:

- The outcome and grouping variable as well as the data frame where they live are specified as before via a formula.
- The first row of the output states the test used: here, the Walch t-test, which does not assume equal variances between groups (which the Student t-test does).
- This is followed by information on the data used, the test statistic, and the p-value for the null hypothesis; here, a p-value of less than 3.5E-5 indicates that the difference is statistically significant at most conventional significance levels.

- The test does not state the null hypothesis, but the alternative hypothesis: here, a simple two-sided alternative, i.e. we reject the null hypothesis of equal means for both large negative and large postive mean differences.
- This is followed by a number of descriptive statistics, here: a 95% confidence interval for the mean difference between groups, and the respective group means.

The results here are very similar to the results in Table 3 of the original publication (and would be identical if we had chosen the Student t-test instead of Walch.)

Export results

Results to be included in a report can be exported manually: text output can be selected in the console, copied to the clipboard, and pasted into the target document. Graphs can be extracted via the `Export`-menu in the plot tab, and either be saved to an image file, or again copied to the clipboard.[22]

Manual export / copying and pasting is reasonably fast and convenient for a few numbers and plots, but generally not recommended, because it breaks the connection between the R code use to generate the results and the results themselves: this is a common source of errors that hard to track. It also has to be repeated whenever the data or the analysis changes, and formatting even moderately large result tables becomes very time consuming.

Therefore, results should generally be exported as part of an analysis script that automates both analysis and extraction of the desired results, see R Scripts below.

[22] In base R, the same functionality is available via right-clicking the plot window

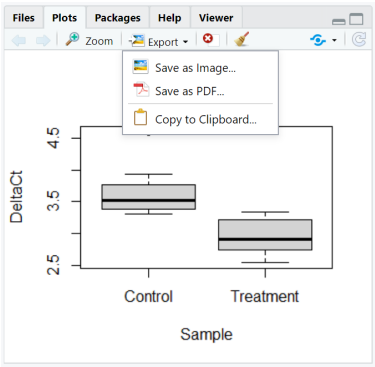

Figure 10: RStudio Files pane

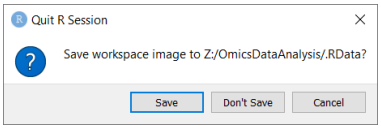

Figure 11: R Quit dialogue

Shut down R

Terminating an R session is easy: select the `Quit`-entry in the `File`-menu, or kill the program window, or use the command `q()` in the console.

However, by default, all data, results, and plots generated during an R/RStudio session will be lost when quitting R, unless you take steps to save them for later. Therefore, by default, R will always offer to save the *workspace image* when you quit a session: this means that all currently defined objects will be saved to a file called `.RData` in the current working directory. This file can later be loaded via the `Load workspace`-entry in the `Session`-menu.[23], or via the `load()` command in the console.[24]

Note that you can save the current workspace image at any time via the `Environment`-tab or the `save.image()`-command.

In addition, RStudio will also save all the R commands used in the current R session (as seen in the `History`-pane) to a file `.Rhistory` in the current working directory.[25] This is a simple text file that can be opened in any text editor, or re-loaded into R via `History`-pane or the `loadhistory()`-command.

[23] In base R, in the File-menu

[24] If the current working directory is also the start-up directory for R/RStudio, this .RData file will be loaded automatically at the start of the next session, with a message in the console.

[25] By default, RStudio will always save the history, but base R only when also saving the workspace image.

Adding functionality with R packages

Background

R supports a mechanism through which user-written software can be distributed and used in (almost) the same manner as the commands built into R: *R packages*. These packages are written using R commands, similar to the R scripts discussed below, but may also include code from other programming languages like e.g. C++. While R is a powerful statistical package in its own right, it is the huge collection of R packages that contribute the majority of methods available through R.

Packages are mostly available through curated online collections, i.e. *repositories*:

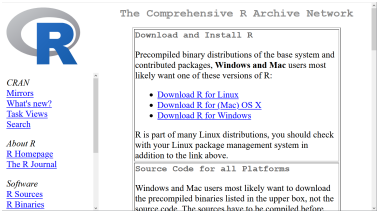

Figure 12: CRAN https://cran.r-project.org/

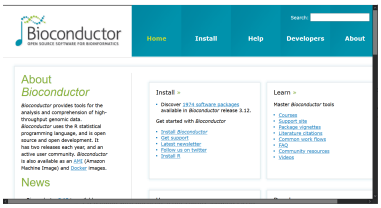

Figure 13: Bioconductor https://www.bioconductor.org/

1. The most important and default repository is CRAN, the Comprehensive R Archive Network, with currently more than 16,000 packages covering an extremely wide range of applications.

2. Bioconductor is another large (currently ca. 2000 packages) collection, focused on high-throughput genomic data.

The curation process for these repositories ensures minimal standards of software quality and documentation. Other sources which also host many R packages for download, like github, lack these standards, leading to more variable package quality.

Installation vs Loading

In order to make use of the commands and data sets in an R package, two steps are necessary:

1. The package needs to be *installed*: this means that there has to be a local copy of the data and software on a file path known to R. Packages available on CRAN can be downloaded via the command `install.packages`, e.g. as in

```
> install.packages("ggplot2")
```

   Note that installation is only required *once*: after the package has been installed, the local copy remains on the local hard disk for all future R sessions.

2. The package needs to be *loaded*: this tells R to activate the installed copy of the package and make its built-in commands and data sets available at the console. This is done via the command `library`, as in

```
> library(ggplot2)
```

   Note that loading is required *once per session*, namely the first time before you use one of the package commands or data sets; after the package has been loaded as above, it syats activated until the end of the R session.

Installation and loading of packages can also be done via the `Packages`-pane in RStudio: this tab lists all installed packages currently installed, which can simply be loaded via checking their selection box. Extra packages can be installed via the `Install`-dialogue in this pane.

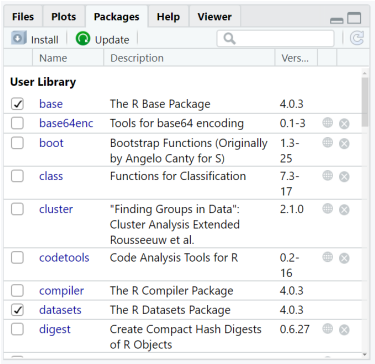

Figure 14: RStudio Package pane

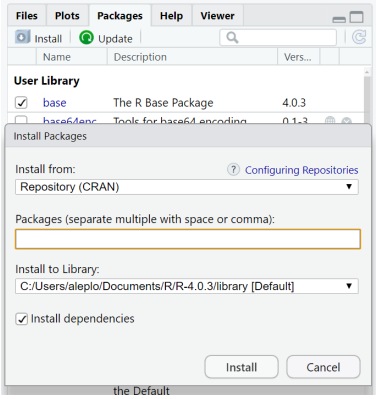

Figure 15: RStudio Package pane with Install dialogue

Example packages

Widely used packages include e.g.:

- `ggplot2`, a complete and modern plotting system
- `dplyr`, offering consistent data manipulation (SQL-style)
- `data.table`, which implements a high-performance alternative to data frames for large data sets.

Popular packages that provide useful tools for bioinformatics-type data and are available from CRAN:

- `matrixStats`: statistics for rows/columns of a data matrix
- `factoextra`: data reduction methods (e.g. principal components) and visualizations

Other popular packages for bioinformatics-type data, available from Bioconductor:

- `limma` allows fitting of linear models for many features in parallel
- `biomaRt` interfaces to online repositories for feature annotation

Note that these packages have a different installation process, demonstrated on the package homepages.

Finding more useful packages

As it is often difficult identify just the right package required for a task simply through search engines, both CRAN and Bioconductor offer curated lists of packages that can be used for specific application areas:

- For CRAN, these are *task views*[26]: e.g. for genetic applications[27]
- For BioConductor, these are *BioCViews*[28], which organize packages hierarchically by application, domain, method etc., e.g. packages useful within metagenomics[29]

[26] cran.r-project.org/web/views/

[27] cran.r-project.org/web/views/Genetics.html

[28] www.bioconductor.org/packages/release/BiocViews.html#__

[29] www.bioconductor.org/packages/release/BiocViews.html#__

Building a simple script

R/RStudio make it very easy to translate an interactive data analysis into an *analysis script*: this is simply a text file containing a series of R commands that act together to achieve some goal, e.g. clean a data set, fit a regression model, create a plot etc.

Why scripts?

The main reason is **replicability**: given the original data and the corresponding analysis script, anyone (including yourself) can replicate the intended analysis. Given only the data and the analysis results, this is generally not possible.

Scripts have a number of other important practical advantages: a script can be

- *documented*, making it easy to understand how and why an analysis was performed, and how to interpret the results;
- *modified*: if the data changes, or if the analysis needs to be expanded, the relevant parts can be edited, and the modified script can be re-run;
- used to generate *draft reports* that contain all results and graphs.

Building a script

It is easiest to start with an interactive analysis, e.g. by running a workflow as described in the previous section.

All commands used in an interactive session are shown in the **History** tab. From there, some or all of the commands can be selected and copied to a text file via the `To Source`-button.

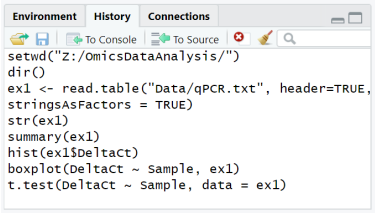

Figure 16: RStudio History pane showing the workflow example

This text file is then opened in the **Source**-pane of RStudio, which is an integrated multi-tabbed text editor with special support for editing and running R script files, as seen in Figure 17:

- tab-expansion for R functions, function arguments and file names works,
- some or all of the commands can be selected and run at the console, either via the `Run`-Button at the top of the editor pane, or by pressing `Ctrl-Return`at the same time.

Once the newly generated R script with the recycled commands has been generated, it should be save directly under a suitable file name with the extension `.R`.

Adding comments

Comments are text in the script that is ignored when the commands are run: this is information that helps the person reading, editing or running the script to understand what it does, and how it does it. Comments provide context and are crucial making an analysis script *easily* replicable and modifiable.

Note that the person that will read, edit etc. the script and needs to understand it may be an internal or external collaborator, but is most likely *yourself*

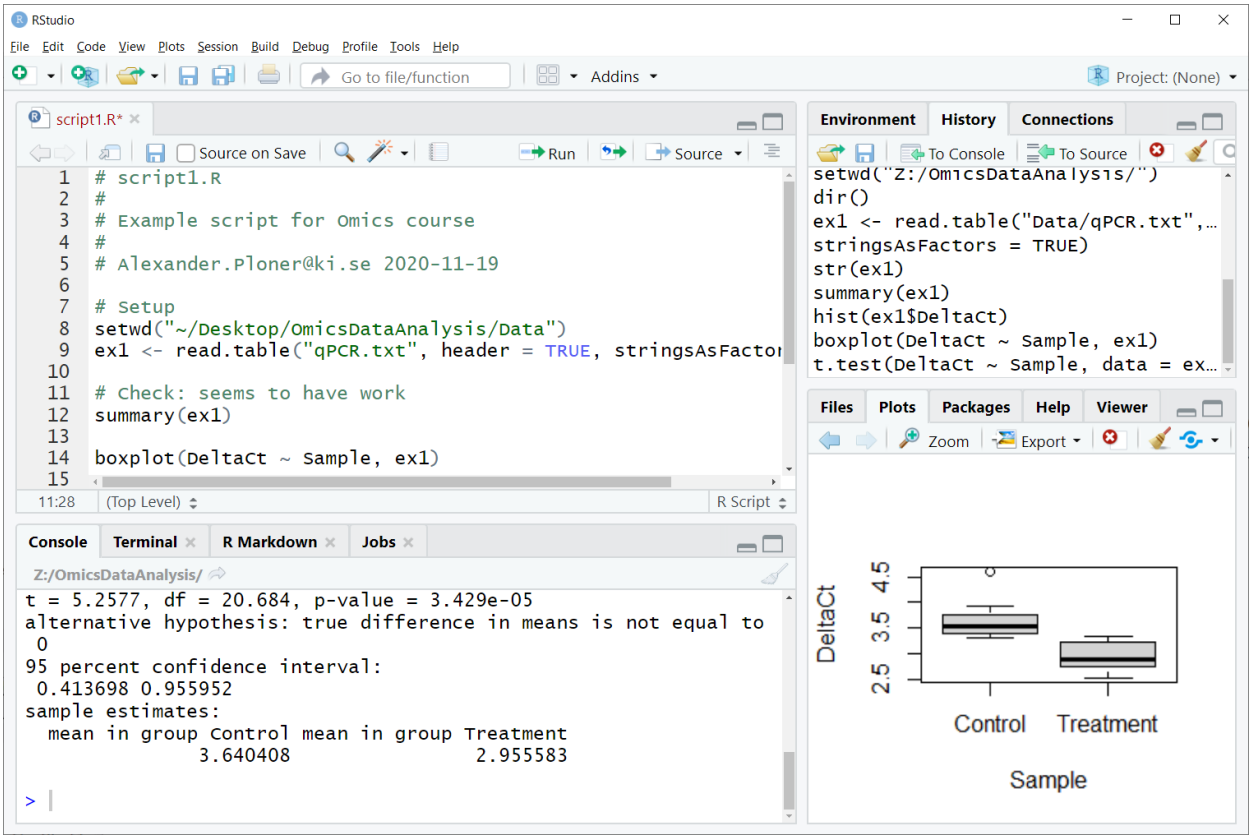

Figure 17: RStudio with open Source pane showing script script1.R

in six months: writing good comments means that you are making your own life easier.

Comments are easy to add: R/RStudio will ignore any text after a hastag # until the end of the current line.

Personally, I always include the filename, a short description, my email address and the original date of writing in any R script I save; I also add comments to separate major steps in the workflow, like setup, data import, data cleaning etc. and explain anything unusual or smart I have done in the code (see e.g. the script files included with this document).

Running a script

A saved R script can always be opened in RStudio (as a tab in the editor pane), the code selected and run via the `Run`-button. Alternatively, the whole script (from first to last command) can be run via the `Source`-menu, or the `source`-command in the console.

From script to draft report

In you have opened an R script in the editor pane, you can translate it into a draft report by clicking the *Compile report*-button [30] at the top of the pane. This allows you choose an output format ((HTML, PDF, Word), and will then

- run all commands in the script,
- collect all output generated by these commands, both graphical and numerical,

- combine all the comments, code and results in one document,
- convert that document to the chosen format.

This draft output can then be edited to generate a full report on the data analysis:

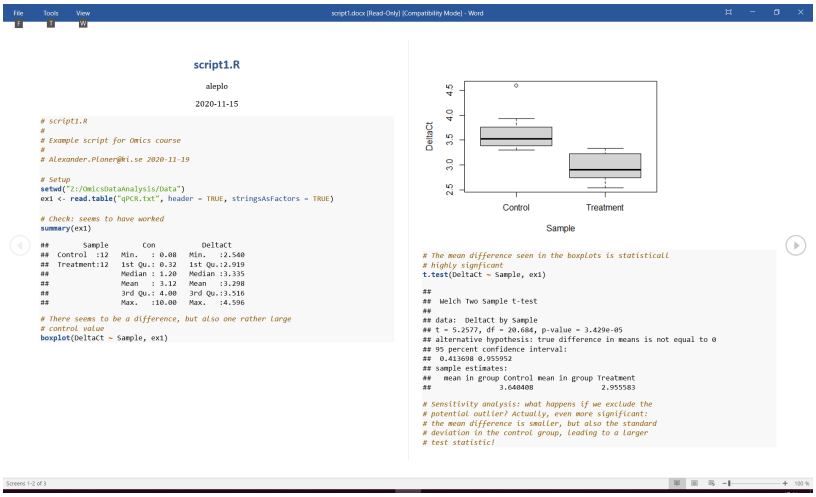

Figure 18: Compiled Script1.R as Word file

This technique can be greatly expanded to generate e.g. automated reports from data, see e.g. package `knitr`.

Other Resources

- CRAN user contributed documentation of varying length[31]
- RStudio cheatsheets: 1-page summaries on how to use RStudio, specific packages, or perform specific packages[32]
- Bioconductor common workflows show how Bioconductor packages can work together[33]
- Bioconducor user-contributed documentation[34]
- Bioconductor course material[35]
- Bioconductor featured publications[36]

[31] cran.r-project.org/other-docs.html

[32] rstudio.com/resources/cheatsheets/

[33] bioconductor.org/packages/release/workflows/

[34] `http://bioconductor.org/help/community/`

[35] bioconductor.org/help/course-materials/

[36] bioconductor.org/help/publications/index.html