

1 Introduction

This tutorial will give you some basic knowledge about working with *R*.

R is free software (copyright: GNU public license) and is available from <http://stat.ethz.ch/CRAN/>. At this URL you find a comprehensive **Documentation**, **Manual**, “An Introduction to *R*” (about 100 pages) and a shorter introduction **Contributed**, “*R* for Beginners” (31 pages).

1.1 *R*-environments

A “professional” way of working with *R* is to edit *R*-script files in an editor and to transfer the written code to a running *R* process. This can be set up on any platform. There are many editors that support this. We recommend the use of *R Studio*, which is available for all common platforms (<http://rstudio.org>).

Alternatives are the editor that comes bundled with *R* (syntax highlighting exists only on Mac OS X), *TinnR* (<http://www.sciviews.org/Tinn-R/>) and *WinEdt* on Windows (<http://www.winedt.com/>), or *Kate* on Linux. Under all platforms, you can also use *Emacs* with the add-on package *Emacs Speaks Statistics* (<http://stat.ethz.ch/ESS/>), which needs a bit of familiarization in the beginning.

This tutorial will focus on working with *R Studio*. *R Studio* combines all resources required for programming *R* in a single tidy window, see Fig. 1. The pane *console* contains a instance of *R*. It is not necessary to start *R* separately.

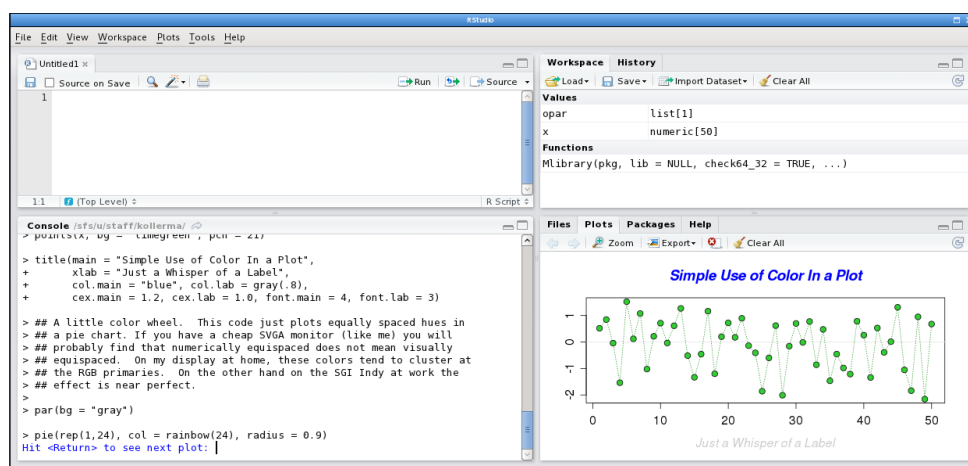


Figure 1: The working environment provided by *R Studio*. The standard pane layout consists of (clockwise, starting top left) the source editor, the workspace and history pane, the files, help and graphics browser, and the *R*-console.

2 *R*-basics

Type in the *R*-console:

```
> x <- 2 (press <RETURN>)
```

```
> x (press <RETURN>)
```

Result: [1] 2

The assignment operator `<-` has created an object `x`. *R* is vector-oriented, so `x` is a vector with one element of value 2.

Remark: You can write `<-` using the shortcut “`<Alt>+-`” (i.e., the keys `<Alt>` and “`-`” pressed at the same time).

Next try (all commands are followed by `<RETURN>`; this is omitted from now on):

```
> y <- c(3,5) (c for combine)
```

```
> y
```

Result: [1] 3 5, a vector with two elements.

`ls()` shows all objects you have already generated. To remove `x`, use `rm(x)`.

Note that many functions are already defined in *R* (for example `c`, `t`, `max`, ...). We advise you to use different names for your variables to avoid confusion.






R includes demonstrations for many functions. You can get a list of all demonstrations with `demo()`. For example, take a look at the graphics demo of *R*: `demo(graphics)`. This will display a variety of plots generated by *R*. Hitting `<RETURN>` in the console will allow you to go from one graphic to the next one.

2.1 Working with an *.R* (script-)file


Create a new script file via **File** → **New** → **R Script**. You should now see four panes just as in Fig. 1. Save the file as *tutorial.R* via **File** → **Save**. From now on, your *R* instructions should be typed in this script-file. Make sure to comment your code (with the symbol `#`) as you go on.

In the editor pane *tutorial.R*, type `z <- c(8,13,21)` as first line and `2*z` as second line.

You have several options to send your *R*-code to the *R*-window:

1. Click on  **Source**. All the code of your script is sent to the *R*-console.
2. Point the cursor on the first line. Then click on  **Run**. Only the selected line (first) is sent to the *R*-console. The cursor now points on the next line (second). Redo  **Run** to send the second line to the *R*-console, and so on.
3. Select the code to be sent to the *R*-console. Then click on  **Run**. This will send the whole selection to the *R*-console.
4. Instead of clicking on  **Run** with your mouse, you can press “`<Ctrl>+<RETURN>`” (i.e., the `<Ctrl>` key and `<RETURN>` at the same time). Both are equivalent.

Remark:

Sometimes the evaluation of an *R*-file takes too long (usually if you have errors in some loops). At any time you can interrupt the evaluation by clicking  (this will only show up when *R* is calculating something) or clicking in the *R*-console and pressing `<Esc>`.

From now on you should write (almost) all *R*-instructions into the *.*R*-file to evaluate them from there. At the end, you can save your script file by clicking on **File** → **Save**.

2.2 Computing with vectors

Type `fib <- c(1,1,2,3,5,z)` as next line of *tutorial.R* (gives the first eight Fibonacci-numbers). Evaluate the line, and take a look at `fib`. Type `2*fib+1`, `fib*fib` and `log(fib)` as next three lines of *tutorial.R*. Mark all three lines with the left mouse button and send them to the *R*-console. This evaluates all marked lines. Check the results. Do you understand them?

Now create the sequence 2, 4, 6 as object `s`: `s<-2*(1:3)`, alternatively `s<-seq(2,6,by=2)`. Take a look at `fib[3]`, `fib[4:7]`, `fib[s]`, `fib[c(3,5)]` and `fib[-c(3,5)]`.

Create a vector `x` with 8 elements, some of which are positive, some negative. Check `x > 0` and `fib[x > 0]`.

Don't forget to put comments in your script file. Up to now, it could for example look like this:

```
## Computational Statistics -- R tutorial
## Author: Hans Muster
## Date: 26 Feb 2010

## getting started
z <- c(8,13,21)
2*z

## computing with vectors
fib <- c(1,1,2,3,5,z)      # vector with first 8 Fibonacci numbers
fib
2*fib + 1                 # element-wise operations
fib*fib                   # element-wise multiplication
log(fib)                  # takes the log of each element
s <- 2*(1:3)               # vector holding 2, 4, 6
s1 <- seq(2,6,by=2)        # same vector as s
fib[3]                    # 3rd element of vector fib
fib[4:7]                  # 4th, 5th, 6th and 7th element of fib
fib[s]                    # 2nd, 4th and 6th element of fib
fib[c(3,5)]               # elements 3 and 5 of fib
fib[-c(3,5)]              # vector fib without elements 3 and 5
x <- c(1,-3,5,-1,8,9,-2,1) # new vector x
x > 0                      # elements 1, 3, 5, 6 and 8 of x are > 0
fib[x > 0]                 # elements 1, 3, 5, 6 and 8 of fib
```

2.3 Data Frames

A data frame is a generalized matrix. The main difference between data frames and matrices is that matrices need all elements to be of the same type (e.g. numeric, character), while data frames allow every column to have another type.

ASCII-data is most easily read by the function `read.table`, which generates a data frame. On the ILIAS course page, you find a data set in a file called `no2Basel.dat`. Download it and read it into *R* using

```
no2 <- read.table("no2Basel.dat", header=TRUE)
```

You may examine the created object directly by typing `no2` in the *R*-console. Single columns are accessible by `no2[, "N02"]` (or `no2$N02`, but not for matrices). You may take a look at the original file in an editor, in particular its first line, to understand why *R* knows the name of the columns. The parameter `header=TRUE` of `read.table` tells *R* that the column names are in the first line. `no2` is still small enough, but in general it is useful to use the function `str` first, which displays the structure and type of an object, but not every single element: `str(no2)`. `summary(no2)` displays information about the columns of `no2`. `summary` extracts the most important information from lots of *R*-objects, e.g., the results of statistical tests or regression fits.

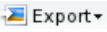
2.4 Graphics

Draw a histogram of the NO₂-values of the no2-data.

```
par(mfrow = c(1,2))  # Number of pictures one below the
                      # other [1] or side by side [2]
                      # Important to save paper!
hist(no2[, "NO2"])    # draw histogram.
```

Now compute the regression line of the NO₂-content against temperature and show it graphically next to the histogram:

```
lm.T <- lm(NO2 ~ Temp, data = no2)  # fits regression.
plot(NO2 ~ Temp, data = no2)
abline(lm.T, col = 4, lty = 2)      # col: colour; lty=2: dashed line
summary(lm.T)                       # regression summary (details later)
```

`title("Title xy")` adds a title to your graphic and the button  **Export** can be used to save the plot as an image or pdf.

Note that there is a distinction between “high-level”- (such as `plot`, `hist`) and “low-level”-graphics functions (such as `abline`). The former make up a new graphic, while the latter add something to existing graphics.

2.5 Getting help

If you want to know the details about functions, you can use the *R*-help-system. For example, `help(plot)` explains the `plot`-function (also try `?plot`). You can execute the example at the end of the help page by `example(plot)`.

If you look for help about some topic without knowing the function names, e.g., about histograms, `help.search("histogram")` delivers a list of functions which correspond to the keyword. In parentheses you find the name of the package to which the function belongs. Most functions used by us in the beginning are contained in the package “base”, which is automatically loaded. Other packages must be loaded by `require(package)`, before their functions and help pages are accessible.

2.6 Ending *R*

You can save your work by saving the file of instructions *tutorial.R* (see above; of course it is useful to use new files for new projects, e.g., *exercise1.R*, *exercise2.R*, ...). The instructions have to be evaluated again to restore your work. *R*-objects may be saved also by the functions `save` and `write.table`.

The function `q()` terminates the *R*-session (this is the same as **File** → **Quit R**). Answer **n** to the question *Save workspace image?*.

3 Online tutorial

If you don't feel comfortable with basic *R* commands yet, go through the *R* introduction provided by the “CodeSchool”: <http://tryr.codeschool.com/>

It is suggested that you sign up for a CodeSchool account first; otherwise, you cannot pause the course since you have to restart from the beginning each time you access the web page. With an account, you can also download and print out the completed worksheet in the end.