



Introduction

This tutorial will give you some basic knowledge about working with *R*. It will also help you to familiarize with an environment to work with *R* as it is provided in the computing labs in the ETH main building.

About *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 pdf) and a shorter introduction Contributed, “R for Beginners / R pour les débutants” (31 pages, English/French).

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. Below, we will describe one possible setting under Linux which was chosen for this exercise class. However, this is not the only possibility. Depending on which platform you are working on, we recommend the following:

linux Emacs with the add-on package Emacs Speaks Statistics (<http://stat.ethz.ch/ESS/>)

mac The built-in editor is already quite powerful (<http://cran.r-project.org/bin/macosx/>)

windows: TinnR (<http://www.sciviews.org/Tinn-R/>) and WinEdt (<http://www.winedt.com/> and <http://cran.r-project.org/web/packages/RWinEdt/index.html>) both support R syntax highlighting and direct submission of *R* code

platform independent Java GUI for *R* (<http://jgr.markushelbig.org/JGR.html>) or Eclipse (<http://www.eclipse.org/>) with StatEt (<http://www.walware.de/goto/statet/>)

This tutorial will focus on working under Linux with the editor **gedit**. It will also introduce a web-application (LEMUREN) to submit the code which you want to hand in. Your exam will use *R* in this Linux environment (without internet access).

Getting started on a dual-boot machine

The computers in the computing labs of the ETH main building are dual-boot machines with Windows and Linux installed. If your computer is running under Windows, restart the machine and choose Linux when prompted to specify the operating system.

When the login screen is displayed, enter your account information and log in (nethz name and password).

Getting started with gedit

We use *R* from within the gedit text editor. To start it:

1. Click on the "Fedora applications" in the Taskbar (the big blue "f" in a corner of your screen). Choose **Accessories** → **gedit Text editor**¹. (If it is not the first time you use gedit with *R* integration, then skip points 2-4).

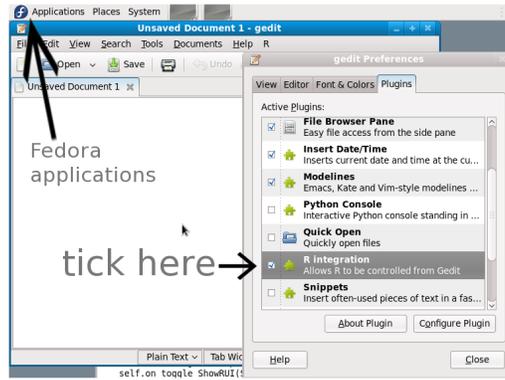


Figure 1: Fedora applications menu, opened gedit and gedit Plugins window. Make sure *R*-integration is ticked (as in the figure).

2. Choose in the menu bar **Edit** → **Preferences**. A window will appear, click on **Plugins** (upper right), then tick the package "R-integration", and click on **Close**. The menu **R** will appear in the menu bar.
3. close and restart gedit.
4. Click on **R** → **Show/hide toolbar** → **Show toolbar only for R files...** (it will be not ticked!)
5. From now on you can use gedit as *R* compiler! You should have two windows in gedit, the editor-window and the *R*-window (see Figure 2)

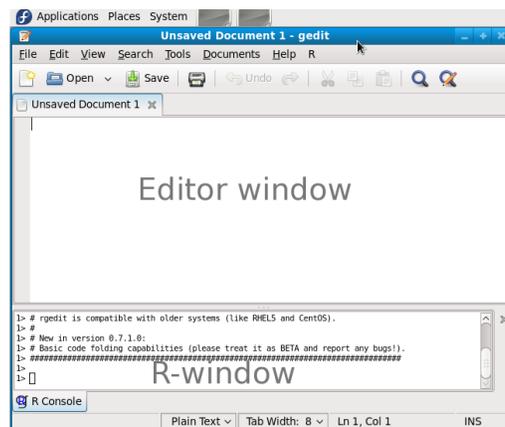


Figure 2: The working environment is set up: *R*-code can be executed in the *R*-window.

¹Alternative: Open a terminal by clicking on the corresponding icon in the Taskbar. Type **gedit** followed by **<RETURN>**

R-basics

Type in the *R*-window:

```
> x <- 2 <RETURN>
```

```
> x <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.

Next try (all commands have to be confirmed 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 *R*-window will allow you to go from one graphic to the next one.

Working with an .R (script-)file

Click in the editor window. Save the file as *tutorial.R* via **File** → **Save As...** 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 window *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 the menu **R** → **Run the whole file through R...** All the code of your script is sent to the *R*-window.
2. Point the cursor on the first line. Then click on **R** → **Run current line through R...** Only the selected line (first) is be sent to the *R*-window. The cursor now points on the next line (second). Redo **R** → **Run current line through R...** to send the second line to the *R*-window, and so on.
3. Select the code to be sent to the *R*-window. Then click on **R** → **Run selection line through R...**

In the toolbar you can find icons with the same meaning as the commands cited above.

It might be handy to define a shortcut for some commands. To set a new shortcut, go to **R** → **Configure R interface**, then click on **Keyboard Shortcuts**. Enter a shortcut of your choice by writing the corresponding key combination (e.g. `<Shift><Ctrl>R`). Do not choose a key combination that is already in use for another command. Once you have chosen your combination, quit the setup window by clicking twice on **OK**. Close and restart *gedit*.

From now on you can send your *R*-script (or highlighted parts of it) to the *R*-window by pressing the defined key combination (e.g. `<Shift><Ctrl>R`).

Remark 1:

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 at the *R*-window and then typing on `<Ctrl>Q` or on `<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**.

Remark 2:

By closing gedit, all what you have in the *R*-window is lost (variables, functions,...). Therefore you should resend your *R*-script to the *R*-window every time you open it.

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 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
```

Matrices: creation and computation

Create two vectors `x <- 1:4` and `y <- 5:8` and the matrices `mat1 <- cbind(x,y)` and `mat2 <- rbind(x,y,x+y)` (`cbind` means column-bind, `rbind` means row-bind). Take a look at the whole matrices `mat1` and `mat2` and try `mat2[3,2]`, `mat2[2,]` and `mat2[,1]`.

Computation with matrices using `+`, `*` etc. follows the same rules as computation with vectors, namely element-wise. For the matrix product, use `%*%`, e.g. `mat2 %*% mat1`.

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.

Reading and looking at datasets

ASCII-data is most easily read by the function `read.table`, which generates a data frame. `read.table` works also for datasets from the web. Try:

```
no2 <- read.table("http://stat.ethz.ch/Teaching/Datasets/no2Basel.dat",
                 header=TRUE)
```

You may examine the created object directly by typing `no2` in the *R*-window. Single variables are accessible by `no2["NO2"]`. You may take a look at the original file, in particular its first line, to understand why *R* knows the name of the variable. This can be done by calling the above URL from a web browser, e.g. Firefox. The parameter `header=TRUE` of `read.table` tells *R* that the variable 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.

An alternative to `read.table` is the function `scan`, which reads vectors and lists. A list is a more general structure which may contain elements of different types and sizes, e.g. vectors of varying lengths, data frames, sublists, etc.

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 `dev.print()` prints the graphic.

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.

Getting help

If you want to know the details about functions, you can use the *R*-online help. For example, `help(plot)` explains the `plot`-function. Type `q` to exit the help window. You can execute the example at the end of the help page by `example(plot)`.

An alternative to the `help`-function: `help.start()` starts the html-help of *R* in a web browser.

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 `library(package)`, before their functions and help pages are accessible.

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` or by creating a new output file and use of the copy, cut and paste facilities of the editor.

The function `q()` terminates the *R*-session. Answer `n` to the question Save workspace image? [`y/n/c`] or use `q("no")`.

More to come

R can be used to create complex programs and functions. You may take a look at `help(for)` for control-flow constructs or at `help(function)` for creating functions.

Handing in your homework

If you want us to correct your solutions to the exercises, you should:

1. Print out important *R*-output and plots. Don't forget to add your interpretation of the *R*-output and plots and to answer the questions on the exercise sheet. A very elegant way to hand in your solution is to combine everything in a single file (for example by generating a pdf with L^AT_EX or OpenOffice).
2. Submit your code on the LEMUREN web-interface. We will only correct code that has been submitted through this interface (don't print it out, don't send it to us by email). Also, we will only correct well documented code.

You have to open a web browser to access the LEMUREN interface: click on the Firefox icon in the Taskbar. Use the URL that was sent to you by email (something like `http://karoline.ethz.ch/dka/?role=student&key=xxxxx`).

In the upper left corner of the browser window you will see your name. In the menu below you can access the exercise sheets. You can download/open an exercise sheet by clicking on the corresponding pdf link. If you move the cursor of the mouse on to the name of an exercise sheet (for example “Exercise 1”) you will see the deadline to hand in your solution. By clicking on the + sign in front of an exercise you can access the different tasks.

Click on the + in front of “Tutorial”. Then click on “task”. You see four panels (see Figure 3). The upper left one corresponds to the editor panel. The upper right one will display the *R*-output. The third one will display the content of the “.R” and “.Rout” files in your working directory. The last one is a tool to easily access the R help.

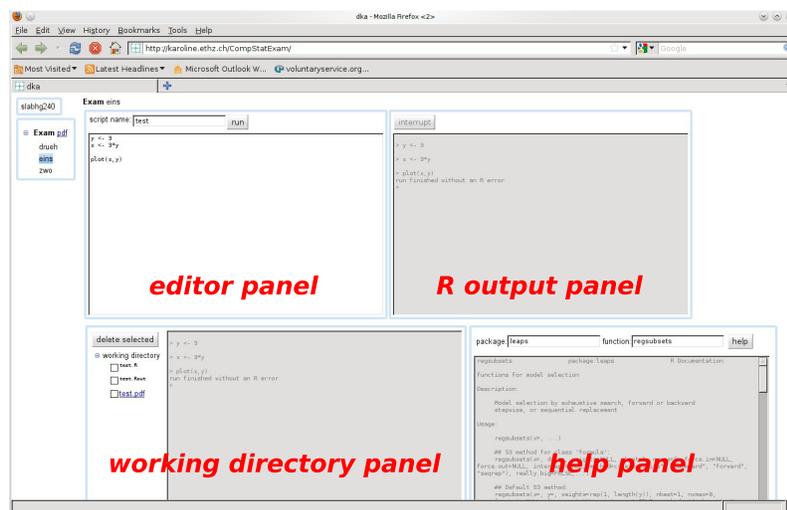


Figure 3: The LEMUREN interface: a handy tool to submit your solutions to the exercises and to get feedback regarding your code from the teaching assistants.

Switch back to the editor `gedit`. Copy all the instructions contained in your file `tutorial.R` and paste them into the editor panel in LEMUREN. Click in the field `script name:` and provide a name for your script. Do **not** include the “.R” in the name. Click on the button `run`. If there is no error in your script, all *R* instructions are carried out, and you see the output from *R* in the *R*-output panel. At the end of the buffer the message `run finished without an R error` should be displayed.

In addition, several files have been created and are now available in the working directory panel. Your script has been saved under the name you chose with the extension “.R”. The output of *R* has been saved under the same filename with the extension “.Rout”. If your code generated any plots, a file holding all the plots has been generated with the same filename with the extension “.pdf”.

You can click on the file names in the working directory. If you click on a “.pdf” file, you can open/download it. If you click on a “.R” or “.Rout” file, the content of the file is displayed in the buffer. You can delete files that you do not need anymore by checking the corresponding boxes and then clicking on `delete selected`.

The teaching assistants will be able to see all the files that you have in your working directories.

To use the help panel, simply click in the field `function:` and type the name of the R function for which you would like to read the help file. If the function does not belong to a standard package, you also have to specify the name of the package in the field `package:.` Try for example to call the help file for the function `lm`. Now try to call the help file for the function `regsubsets` without specifying a package, and then try again while specifying the package `leaps`.

A few things to keep in mind:

- When you click on the “run” button, the whole script is evaluated in *R* (not submitted line by line). Before being evaluated, your script is parsed (with the function `parse`). A syntax error (including an incomplete expression) will return an error.
- Instead of clicking on the “run” button, you can also type `<Shift> + <RETURN>` in the editor panel to evaluate your script.
- While your script is being evaluated, a wheel is turning next to the “run” button.
- If you want to stop the evaluation of a script, click on the “interrupt” button.
- If you want to save your plots one by one, you can use the function `pdf` in your script.
- If you need help from the teaching assistants for a particular problem, you can prepare a script called `help_request.R`. Once you submitted your help request in LEMUREN, send an email to `compstat@stat.math.ethz.ch` with the indication where we can find your code (for example in “Exercise 1”, “Task 2”). Make sure to structure your requests for help as it is mentioned in the “organizational sheet” of the lecture.
- LEMUREN is a nice tool to submit your code and requests for help. However, we strongly advise you not to develop your code on LEMUREN. It is better to become familiar with an editor to which you will still have access after this lecture is over. Therefore, develop your scripts with the editor of your choice, then submit your code via the LEMUREN web interface.