R Tutorial

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

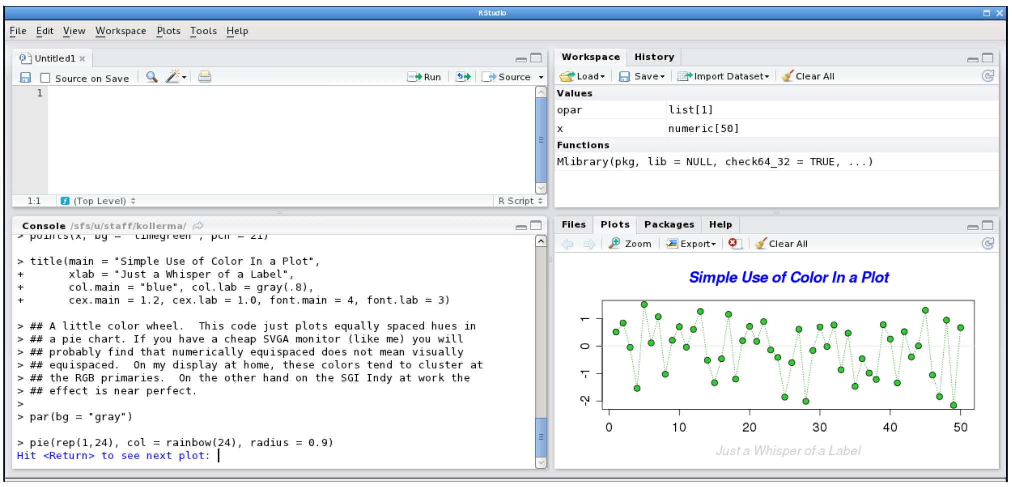
A "professional" way of working with *R* is to edit *R*-script files in an editor and to transfer the written code to a running *R* process. This can be set up on any platform. There are many editors that support this. We recommend the use of *R Studio*, which is available for all common platforms (`http://rstudio.org`).

Alternatives are the editor that comes bundled with *R* (syntax highlighting exists only on Mac OS X), *Emacs* with the add-on package *Emacs Speaks Statistics* (`http://stat.ethz.ch/ESS/`), *TinnR* (`http://www.sciviews.org/Tinn-R/`) and *WinEdt* on Windows (`http://www.winedt.com/`). This tutorial will focus on working with *R Studio*.

Getting started with R Studio

We use *R* from within R Studio. To start R Studio, find it in the applications menu or type `rstudio` in a terminal.

R Studio combines all ressources 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.


RStudio
File Edit View Workspace Plots Tools Help
Untitled1
Source on Save
Run
Source
1:1
(Top Level)
R Script
Workspace History
Load
Save
Import Dataset
Clear All
Console /sfs/u/staff/kollerma/
Files Plots Packages Help
Zoom
Export
Clear All
Simple Use of Color In a Plot
Just a Whisper of a Label

```
Values
opar        list[1]
x           numeric[50]
Functions
Mlibrary(pkg, lib = NULL, check64_32 = TRUE, ...)
```

```
> points(x, bg = "limegreen", pch = 21)

> title(main = "Simple Use of Color In a Plot",
+       xlab = "Just a Whisper of a Label",
+       col.main = "blue", col.lab = gray(.8),
+       cex.main = 1.2, cex.lab = 1.0, font.main = 4, font.lab = 3)

> ## A little color wheel.  This code just plots equally spaced hues in
> ## a pie chart. If you have a cheap SVGA monitor (like me) you will
> ## probably find that numerically equispaced does not mean visually
> ## equispaced.  On my display at home, these colors tend to cluster at
> ## the RGB primaries.  On the other hand on the SGI Indy at work the
> ## effect is near perfect.
>
> par(bg = "gray")

> pie(rep(1,24), col = rainbow(24), radius = 0.9)
Hit <Return> to see next plot: 
```

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.

_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.

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 be 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`.

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
```

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 `read.table`, which generates a data frame. `read.table` works also for datasets from the web. Try:

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

The dataset stream contains the concentration of zinc scaled in four levels (variable ZINC) from different rivers (variable STREAM). In addition the variable DIVERSITY, which depicts the species diversity of the river. The variable ZNGROUP encodes the zinc levels numerically. You may examine the object directly by `stream`. Single variables are accessible by `stream[,"ZINC"]`. 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 or Mozilla. The parameter `header=TRUE` of `read.table` tells *R* that the variable names are in the first line of the data frame. `stream` is a small dataset so it can be displayed by calling it directly, but in general it is useful to use the `str` function, which displays the structure and type of an object, but not all elements: `str(stream)`. We see that the variable ZINC is already identified as a factor. However the variable ZNGROUP is identified as a continuous variable.

`summary(stream)` displays information about the columns of `stream`. `summary` extracts the most important information from lots of *R*-objects, e.g., the results of statistical tests or model fits.

Graphics

Draw a histogram of the DIVERSITY-values of the stream-data.

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

Now draw a scatter-plot of the diversity against ZNGROUP:

```
plot(stream[,"ZNGROUP"],stream[,"DIVERSITY"])   # produces the scatter-plot.
```

Compute an analysis of variance in order to test if there is a difference in diversity for different amounts of zinc: Therefore you first have to change ZNGROUP into a factor.

```
stream[,"ZNGROUP"] <- as.factor(stream[,"ZNGROUP"])
fit.av <- aov(DIVERSITY ~ ZNGROUP, data = stream)   # ANOVA.
summary(fit.av)                                     # returns the model summary.
```

In order to check the normality assumptions draw a normal Q-Q plot:
You can extract the residuals of fit.av with **resid(fit.av)** and the fitted values with **fitted(fit.av)**.

```
par(mfrow = c(1,1))      # setting device back to one picture.
qqnorm(resid(fit.av))    # draws the normal Q-Q plot.
qqline(resid(fit.av))    # adds the corresponding diagonal.
```

`title("Titel xy")` adds a title to your graphic and clicking the print button in the *R*Graphics window prints the graphic.

Note that there is a distinction between "high-level"- (such as `plot`, `hist`) and "low-level"-graphic functions (such as `qqline`). 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*-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.

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?*.

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.