This tutorial will give you some brief basic knowledge 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 dbutants" (31 pages, english/french). This tutorial is based on the UNIX/Emacs installation of the ETH-Seminar für Statistik. If you want to get a similar working environment on your private UNIX/Linux computer, you have to install the free software ESS ("emacs speaks statistics"), see `http://stat.ethz.ch/ESS/`. *R* works on MS-Windows and Macintosh systems as well. A (german) version of this tutorial for MS-Windows (recommended editors are Tinn-R or WinEdt) is available upon request.

Getting started with Emacs

We use *R* from within the Emacs editor. Emacs is a large and mighty tool, but we restrict ourselves to minimal basics.

Emacs is started by typing **emacs** in the shell followed by <*RETURN*>, as always.

The following two keys are of particular importance: The control-key (<*Ctrl*> or <*Strg*>), abbreviated as **C-**, and the meta-key (next to the space key, with a small square on it; on many computers you can use the Alt- and/or Esc-key for this) abbreviated as **M-**. **C-x** means: type the control-key, type shortly on x, let go the control-key; analogously **M-x**.

Text can be marked by use of the left mouse button in Emacs. The middle mouse button copies a marked part of text to the current position of the cursor. A double click on the right mouse button deletes the marked text.

Leave Emacs by selecting *File / Exit Emacs* in the menu.

Important keys

C-g : terminates an Emacs-command
: try, if emacs seems to do something different than what you want...
C-] : in case of messages concerning the minibuffer including "abort it with '^]' "
_ : (underscore) produces `<-` (*R* assignment) in ESS
C-c C-c : terminates *R*-commands in ESS.

Getting started with *R*

To start *R* inside of Emacs, type **M-x R**, then <*RETURN*>. (Attention! As the whole "world of UNIX", Emacs is "case sensitive": capital R is not the same as small r!). Simply type <*RETURN*> when asked for the **Starting Directory** (at the bottom line of Emacs).

Creating and deleting objects

Type in the *R*-window (***R*** is written on the bar below):
> 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.

Warning: Do not use names of *R*-commands as object names, for example: **c**, **t**, **T**, **F**, **max**, ...

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

***R*-demonstrations**
Get a list of all demonstrations with **demo()**. For example, take a look at the graphics demo of *R*: **demo(graphics)**.

Working with an .R (script-)file
Split the Emacs-window into two buffers (**File / Split Window**) or create a second emacs-frame (**File / New Frame**). Open the file **tutorial.R** via **File / Open**. You are asked for the filename in the minibuffer below. (Take care that the Emacs-window is activated. If not, move the mouse to its region and click, if necessary.) Since the file does not already exist, a new file is created. Now you have two buffers: **R** and *tutorial.R*, which you can see on the bars below the buffers.

In buffer *tutorial.R*, type **z <- c(8,13,21)** as first line and **2*z** as second line. Place the cursor on the first line and type C-c C-n. This evaluates the command on the corresponding line in *R* and moves to the next line. If you don't want to move to the next line simply use C-c C-j. In buffer **R** you find an additional >. Repeat C-c C-n. The command on the second line is evaluated, i.e. you get the value of **2*z**.

Computing with vectors
Type **fib <- c(1,1,2,3,5,z)** as third line of *tutorial.R* (gives the first eight Fibonacci-numbers). Evaluate the line (C-c C-n), 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 type C-c C-r. This evaluates all marked lines. Check the results. Do you understand them?

From now on you should write (almost) all *R*-commands into the *.R-file to evaluate them from there. At the end, you may save it by **File / Save (current buffer)**. If you open the file next time, type C-c C-l, you restore your whole work. An alternative to execute all commands of the file *(path/)tutorial.R* is to execute **source("(path/)tutorial.R")** from *R*.

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]**.

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,]** und **mat2[,1]**.

Computation with matrices using +, ***** etc. follows the same rules as computation with vectors, namely elementwise. 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:

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

You may examine the created object directly by **no2**. 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 or Mozilla. 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 **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 command **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("Titel 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 commands (such as **abline**). The former make up a new graphic, while the latter add something to existing graphics.

Getting R-help

If you want to know the details about commands, you can use the *R*-online help. For example, **help(plot)** explains the **plot**-command. You can execute the example at the end of the help page by **example(plot)**. Note that it is a good idea to execute **par(ask=TRUE)** first, to give you time to observe the graphics. You may check **help(par)** to understand this.

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

If you look for help about some topic without knowing the command names, e.g., about histograms, **help.search("histogram")** delivers a list of commands which correspond

to the keyword. In parantheses you find the name of the package to which the command belongs. Most commands used by us in the beginning are contained in the package "base", which is automatically loaded. Other packages must be loaded by **library(Libname)**, before their commands and help pages are accessible.

Ending *R*

You can save your work by saving the file of commands *tutorial.R* (see above; of course it is useful to use new files for new projects, e.g., *exercise1.R*, *exercise2.R*, ...). The commands have to be evaluated again to restore your work. *R*-objects may be saved also by **save** and **write** or by creating a new output file and use of the copy, cut and paste facilities of Emacs (see above, or via **Edit** in the menu bar).

The command **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 commands or at **help(function)** for creating functions.