
R-hints for the lecture Applied Multivariate Statistics

Werner Stahel, Seminar für Statistik, April 2003 / 2005

Jürg Schelldorfer, 2009

1 General Remarks

a These hints

... do not replace an introduction. They provide you with an overview of the most important things in R. It should be enough in order to solve the exercises and it should encourage you to read a full introduction. You will find several longer introductions on the internet.

b Start, quit

How to start R you will find in another tutorial available from <http://stat.ethz.ch/stahel/courses/multivariate/>.

You quit the program by typing `q()`.

c Functions generate objects

Basically: You type something that generates a function call, i. e. `c(2,-1,9,0.4,100)`.

The function generates an „object“, generates a graphical output in a graphical window, or writes a numerical output in the window where R is running.

d Assignment

You can assign a name to a new object with the assignment operator `<-`, which consists of the two symbols `<` and `-`. (In the ESS-mode from emacs the underline `_` generates the assignment operator.)

```
t.v <- c(2,-1,9,0.4,100)
```

e print

When you do not use an assignment operator, the function `print` is evaluated on the generated object and you see the result on the screen.

```
t.v : you see [1] 2.0 -1.0 9.0 0.4 100.0
```

(`[1]` at the beginning says that the first number is the first element of the vector `t.v`. (That is useful when the result requires more than one line on the screen.)

f help

For all functions, there is a description available if you type `?functionname`. For example: `?c`

When you are searching for a function, type `help.search("name").help.start()` guides you to a web-based help system.

g Error messages

Error messages are not always understandable. The S language is flexibel enough that you can write a lot what the system understands partly. Therefore the error message may be generated long after the mistake was made..

h Think in terms of matrices, not numbers!

If you have learnt a classical programming language, then you think immediately about the declaration of types and dimensions. Please forget the latter. Besides you may have got the habit to program loops. Loops which are running over elements of arrays are the wrong start in a matrix-based language. Think about it five times before using a loop! (You require loops for iterative procedures.)

i Environment

The system waits for an input on the command line. In general, we strongly recommend typing the commands in another window (script window). Then with the help of an intelligent text editor like emacs with ESS or Tinn-R sent the command to the R-window (see R-tutorials).

2 Objects

a Vectors

`t.v <- c(2,-1,9,0.4,100)` generates a vector $[2, -1, 9, 0.4, 100]$ and saves it under the name `t.v`

`rep(0,10)` : vector with 10 zeros.

`rep(t.v,3)` : repeat `t.v` three times.

`t.vs <- seq(0,2,0.5)` : Sequence of numbers from 0 to 2 with increment 0.5: $[0, 0.5, 1, 1.5, 2]$

`t.i <- 1:7` : Abbreviation for $[1, 2, \dots, 7]$ (this is abbreviated with a special sign because it is often used.).

`rep` and `seq` can do far more, see `?rep`, `?seq`

b Matrices

`t.m <- matrix(t.v,5,3)` : Matrix of dimension 5×3 , column-wise filled with the elements of `t.v`.

`t.m <- cbind(1:5,t.v,c(0,0,2,2,3))` : concatenate column-wise,
`rbind` : concatenate row-wise.

c Lists

Lists combine objects into a new object. They are used by functions which generate multiple results - e.g., by all function which fit a statistical model.

`list(t.v,t.m)` : generates a list.

d mode

The objects which we have studied comprise only numbers. Furthermore, there exists other types of „content“. `mode(t.m)` shows the „type“, namely `numeric`, `character`, `logical`, `complex` or the type of the object, `list`, `function`. There exist more types ...

e Conversion

You can converse some types into other types:

`numeric` into `character` (und sometimes vice versa),

`logical` into `numeric` (0 for FALSE, 1 for TRUE).

`as.character(1:3)` : gives `[1] "1" "2" "3"`.

`as.logical(c(0,1,-1,3,0.5))` : What is = 0 gives FALSE, everything else TRUE.

f Classes of objects

The basic types are complemented by the possibility to define arbitrary more types. The latter are called classes of objects. They facilitate a specific kind of „object-oriented programming“. Each object has a class. See `class(t.m)`.

g Naming objects

Names of objects are alphanumeric. They start with a letter and they can contain letters as well as numbers and points. Upper case and lower case letters are different.

There are many names which are used by the system itself. Especially the names of the available functions. However, you can use these names and you do not get a warning that you have just overwritten an object of the system.

`pi <- 2` is not harmless...

On the other hand, when the function `c()` is used, the system finds it, even if you have an object with the same name – unless this object is a function itself.

From my perspective, I keep order by using the following rules: The names start with one letter, followed by a dot, followed from further numbers and letters. The letters before the point are used to distinguish the kind of the object. `t.d` is only a temporary used object, i.e. the currently used data set

`d.` : data sets
`t.` : temporary objects
`f.` : own functions with numerical results
`g.` : own graphical functions

3 Functions

a Arguments

Arguments are separated by a `,` . Identification by the position (first, second, ... argument) or by the name of the argument:

```
matrix(t.v, nrow=5, ncol=3)
```

„High-level“ function have many „auxiliary“ arguments for which the function defines a default value.

b Functions are usually very flexible!

Many standard functions generate very different results depending on the kind of the arguments
 Example:

```
diag(1:3) : generates a diagonal matrix with diagonal elements 1, 2, 3.
```

```
diag(t.m) : gives the diagonal of a matrix.
```

c Generic functions

This flexibility is enhanced in the context of generic functions. There are two generations of elements from the object-oriented programming in S. In the older, called S3, the generic function examines the “class” of the first argument and calls the corresponding “method” (an ordinary function).

There are three basic generic functions, which have a method for all objects classes:

```
print : numerical display on the screen
```

Attention: Some objects comprise much more information than you see when you print the object (by not doing an assignment).

```
summary : writes a numerical summary of the object to the R window.
```

```
plot : if there is a meaningful graphical result.
```

The new generation, S4, can react to the class of all arguments. S4-classes are more restrictive and defined more precisely than S3 classes. Needless to say, more care is needed to define them.

d Do not forget parentheses!

A function is only evaluated if the name is followed by `()` . Typing a function name without parentheses produces the definition of the function. Hence, you can not exit the session with `q`, only with `q()` .

e Options

The output of a function does not only depends on the arguments of the function, but also from arguments which are affected and set by `options` (affects the numerical result) and `par` (affect the graphical output).

f You can write your own functions, see later.

4 Operations

a Arithmetic Operations

The arithmetic operations `+` `-` `*` `/` `^` are applied to two vectors elementwise, or even to matrices. `c(2, 4)*c(3, -1)` provides `[1] 6 -4`.

b Recycling

If one argument has more elements than another, the elements of the shorter are cycled repeatedly. `c(2, 4)*3` provides `[1] 6 12`

c Improper values

S knows the improper values

`Inf` and `-Inf` : infinity

`NaN` : not defined

`NA` : missing

d Mathematical functions

I.e. `log10(t.v)` is carried out elementwise. Negative elements of `t.v` give `NA`.

The usual functions are available: `log`, `exp`, `sin`, `cos`, `tan`, `ctg`, `asin`, `...`,

`min(t.v, 500, 1:10)` provides the minimum of all elements.

`which.min(t.v)` : Index of the (first) element which is equal the minimum.

`pmin` elementwise minimization of vectors.

e Matrix multiplication

`t.v %**% t.m`

S does not differentiate between row and column vectors. It tries for both versions if the operation is executable. If you want explicitly a row or a column vector, then you have to save it as a matrix with one row or one column, respectively.

f Logical operations

The logical values are `TRUE` and `FALSE`. You can use the abbreviation `T` and `F`. By overwriting these objects (`T <- FALSE`) you can produce nonsense...

The logical operators are

`!` : not

`&` : and

`|` : or

Logical objects are crucial for `if(logical)` - constructions.

g Comparisons

`==` , `!=` , `>` , `<` , `>=` , `<=`

Attention with `t.v<-3`! This overwrites `t.v`. The comparison with `-3` you get as follows: `t.v<-3`

h Conversion

The arithmetic operations are only defined for `numeric` objects. Logical objects are coerced to numerical objects. In general, S tries very hard to use conversion in order to produce a results.

i Priorities

Generally, the priority is arranged as follows:

function evaluation before `^` : `*/+-` before comparisons before logical operations.

j Summary functions

`sum`, `mean`, `median`, `var`, `...`

k **apply**

You often want to apply a function to all columns (rows) of a matrix. To this end, use `apply`, see `?apply`,

```
apply(t.m,2,mean)
```

Do not write loops!

(For lists there exist the functions `lapply` and `sapply`.)

5 Names and selection of elements

a **Elements of vectors and matrices**

```
t.v[2:3], t.m[c(3,5),3],
```

`t.m[,3]` : the whole third column

b **names**

You can allocate names to the elements of a vector:

```
names(t.v) <- c("Hans","Fritz","Elsa","Trudi","Olga")
```

and then select the element with `t.v["Elsa"]`.

(The syntax is surprising because it looks like the assignment of a value to a function!)

`t.vn <- names(t.v)` displays the names.

Analogue for matrices: `dimnames(t.m) <- list(t.vn, c("V1","V2","V3"))`

Selection: `t.m[c("Elsa","Fritz"),"V1"]` or mixed: `t.m["Elsa",1]`.

c **Direct assignment of names**

```
t.v <- c(Hans=2,Fritz=-1,Elsa=9,Trudi=0.4,Olga=100)
```

```
t.m <- cbind(V1=1:5, V2=t.v)
```

d **paste**

For the generation of names (and character vectors for other things), the function `paste` is very useful:

```
paste("V", 1:3, sep=".")
```

generates `[1] "V.1" "V.2" "V.3"`

The argument `sep` has to be written with its name, since otherwise all arguments without a name are combined. Default is a space " ", hence `paste("AB","CD","ef")` gives `[1] "AB CD ef"`.

6 Data

a **data.frame**

Statistical data often contain numerical variables and „factors“, which have to be saved as characters. These variables are saved together in one object of the class `data.frame`.

b **Reading data sets into R**

from an ASCII-file: `d.abst <-`

```
read.table("http://stat.ethz.ch/stahel/data/abst.dat",header=TRUE)
```

c **Data sets from the system**

```
data(iris)
```

d **Missing values**

As aforementioned, you can highlight values which were not observed, by the improper value `NA`. Statistical functions have to be able to deal with improper values.

e **Conversion**

```
t.d <- data.frame(t.m)
t.x <- matrix(iris[,1:4])
```

f **Selection of elements**

... as with matrices. Data.frames have always names for the rows (observations) and the columns (variables).

7 Basics for graphics

a **plot**

```
plot(d.abst[, "a"], d.abst[, "c"])
```

The function `plot` has many arguments for labelling, data ranges etc. `?plot` !

b **Driver**

In order to use graphical functions, it is required to open a graphical window. This is carried out automatically in R.

`dev.print()` prints the current graphic.

You can write the graphic into a postscript-file or PDF, ..., by using the corresponding driver functions. There exists the Sfs- (Seminar für Statistik-) functions `ps.do` and `ps.latex`; Close with `ps.end()`.

`dev.list()` shows the open graphical windows.

c **high und low level**

Beside `plot` there are many other graphical functions. With “low level” graphical functions you can add graphical elements to the current graphical window.

`text(3,5,"Fritz")` writes the name at the corresponding place in the plot.

d **par**

The graphic is determined by „graphical parameters“. They are saved in a special object, which can be changed by the function `par`.

`par(mar=c(3,3,4,1))` sets the width of the border (below and left to 3 lines, top to 4, right to 1).

There are graphical parameters which can only be set by `par`. On the other hand, there are parameters which can only be selected through the arguments of “high level” graphical functions. Finally, there are arguments which can be set in both ways.

e **Interactive and dynamical graphics**

`identify` : Identifying the points in a plot.

`brush` : Coloring interactively points and 3d-rotation

The choice in the basic package is limited because S intends to be platform independent. And because the graphic is archaic: The graphic is the oldest part of the system, more than 40 years!

8 Organisation of the workspace

a **Global environment, packages**

S seeks the required objects in the so called “environments” (in R) or “frames” (in S).

If an object from the command line is assigned a name, then the object is saved in the “global environment”. The search for an objects starts there and it is continued in the attached “packages” (and the attached data.frames or list of objects). The package `package:base` is always attached, like a few others.

`search()` displays the attached environments in the order of the search.

b Frames

The call of a function generates an environment with local variables, called a frame. During the evaluation, the system first searches this local environment and then in the aforementioned environments. Therefore, be careful: functions can use objects of your global environment which are not entered as arguments! (Well written functions do not do that, except using `options` and `par`)

Hierarchical calls generate a hierarchy of such frames. You need to inform yourself about this when writing your own functions. Attention: Concerning this topic, there are large differences between R and S-Plus!

c Libraries

Currently, there are hundreds of packages available. They can be downloaded from the R website.

`library()` (without an argument) : Directory of all installed packages on your system.

`help(package=fda)` information about the package fda.

`library(fda)` : a package is made available for the current session.

d .First

If you save a function `.First`, then it will be carried out at the beginning of each session.

`.First <- function() { options(digits=3) }` : Sets an option for the session.

e objects

`objects()` lists all objects in the global environment, `objects(5)` in the attached package with number 5.

`objects(pattern="^t\\.*)"` : all objects with names beginning with `t..`

f str

`str(object)` displays the key information about the object.

g rm

`rm(t.d)` deletes the object `t.d`

`rm(list=objects(pattern="t..*"))` deletes all objects which begin with `t..`

9 Distributions and random variables

a Normal distribution

`dnorm(7,5,2)` : density of the normal distribution for $x = 7$, $\mu = 5$, $\sigma = 2$.

`pnorm(7,5,2)` : $P\{X \leq 7\}$ for $X \sim \mathcal{N}(5, 2^2)$.

`qnorm(c(0.025, 0.5, 0.975))` : quantile of the standard normal distribution.

`rnorm(20)` : Generates 20 standard normal distributed random numbers

b Other distributions

There exists the following functions for all well known distributions.

`d... (x)` : Density, for discrete distributions probability of the value `x`.

`p... (x)` : cumulative distribution function, useful for the calculation of p-values: `1-pchisq(3.6,2)`.

`q... (x)` : quantiles (inverse cumulative distribution functions), useful for the calculation of critical values in a test: `qchisq(0.95,2)`

`r... (n)` : n Random numbers according to the distribution (see below).

The following distributions exist:

beta, binom, cauchy, chisq, exp, f, gamma, geom, hyper, lnorm, logis, nbinom, norm, pois, t, unif, weibull, wilcox

and many more in other packages.

c **Random numbers**

`runif(5)` : generates 5 random numbers according to the uniform distribution in (0,1).
`rnorm(20, mean=5, sd=2)` : 20 normal distributed random numbers with expectation 5 and standard deviation 2.
`sample(8)` : Random ordering of the numbers 1 to 8
`sample(3, size=10, replace=TRUE)`
`set.seed(28)` : Starting value for a (pseudo) random number generator. This enables you to get reproducible results.

10 Graphical functions

a **Important graphical (high level) functions**

One-dimensional:

`hist`, `barplot` : histogram and barplot
`qqnorm`, `qqplot` : QQ-diagram for the assessment of distributions

Two-dimensional:

`plot` : scatterplot
`sunflowerplot` : scatterplot with multiple points
`image` : two-dimensional histogram

Multi-dimensional:

`persp` : Perspective view for 3-dimensional data
`coplot` : For 3- and 4-dimensional data: Scatterplot of Y versus X , conditioned on the values of A and B
`matplot` : Several scatterplots in one diagram (i.e. several y -Variables for the same x -values)
`pairs` : scatterplot matrix
`brush` : Interactive version of `pairs`, with 3d-rotation
`symbols` : Scatterplot with symbols representing further variables

b **Important general elements of graphical functions**

`type="p"` : Draw points, `type="l"` : lines, `type="b"` : both
`xlim=`, `ylim=` : Range of the coordinates.
`xlab="x"`, `ylab="y"` : axis labelling
`main="Title"` : title

c **Important graphical “low level”-functions**

Typically you block parts of the result of high level functions by writing `plot(..., type="n", axes=FALSE)` (no points and no axis, respectively). Then you call the corresponding function:

Points:

`points` : draw points
`lines`, `arrows`, `segments` : draw lines, ...
`text` : draw points with character labels

Axis:

`axis` : axis labelling
`box` : border
`mtext`, `title` : text in the border of the figure

Additional arguments:

`abline` : drawing a line
`polygon` : drawing a polygon
`legend` : add a legend for the line types, the plotting characters or colors to the plot

Identifying points:

`identify` : index of the point you click

locator : coordinates

d **Most important graphical parameters**

pch= : plotting symbol (numerical) or character
 lty, lwd : line type and line width
 cex, ps : font size
 mar=c(3,3,2,1) : Width of the border (below, left, top, right)
 col : color

e **Several diagrams (frames) on one sheet**

par(mfrow=c(2,3) : Split into 2 rows and 3 columns, filled row-wise.
 par(mfcol=c(2,3) : ... filled column-wise
 frame() : Go to the next subplot (the high-level functions do it automatically)

The preceding two arguments of `par` determine the old, less flexible way of the allocation on the „screen“. The modern way is as follows:

split.screen : flexible determination of plotting areas,
 screen : choice of the plot

11 Writing functions

a A function for a simple simulation:

```
f.simmed <- function(n, mu=0, sig=1, nrep=1000, seed=1) {
  ## Simulation der Verteilung des Medians
  ## n      Stichprobenumfang
  ## mu     Erwartungswert
  ## sig    Standardabweichung
  ## nrep   Anzahl Replikate
  ## seed   Startwert fuer Zufallszahlen
  ## -----
  set.seed(seed)
  smed <- rep(NA,nrep)
  for (li in 1:nrep) {
    dat <- rnorm(n, mu, sig)
    smed[li] <- median(dat)
  }
  smed
}
```

b **Loops**

The functions comprises a `for`-loop. The syntax is as follows

```
for( loop variable in value vector ) { statements }
```

There are also `repeat` and `while`-loops. Such loops are used in iterative calculations and sophisticated simulations.

(The `for`-loop can often be avoided by using `apply`.)

c **if ... else**

```
if (abs(eps)<0.001) r <- NA else r <- 1/eps
if (nrep <= 2) stop("Such a simulation does not make sense")

for (li in 1:niter) {
  ...
  if ( abs(diff) < tolerance ) break
}
```