Computational Statistics

An Introduction to 

Supplement

Günther Sawitzki

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

Supplement to

Computational Statistics

An Introduction to 

Günther Sawitzki
StatLab Heidelberg
January 27, 2010

in preparation

http://sintro.r-forge.r-project.org/

Introduction

This is a supplement to

G. Sawitzki

Computational Statistics: An Introduction to R

CRC Press, Boca Raton, 2009

It will contain additions, corrections and other supplementary material.

The complete reference appendix of the book is included, with kind permission of CRC Press.

The most recent version of this supplement is on the web site accompanying this book:

`http://sintro.r-forge.r-project.org/`.

Additional material and updates will be available at the web site.

This version: January 27, 2010

Contents

CHAPTER 1

Basic Data Analysis

1.5 R Complements

1.5.3 Complements: Functions

Vectorization

In R, functions preferably are vectorized. If reasonable, they should accept vectors as parameters, and if appropriate, they should return a vector as result. This is a convenience for calling the function. In some contexts, a function can only be used if it is vectorized. So, for example, `curve()` or `integrate()` only accept a function passed as first argument if it is vectorized.

Unfortunately there is no easy way to check whether a function is vectorized. You must rely on the documentation provided by the author, check the source code, or run some test examples.

If you are providing a function, please document clearly where and how it is vectorized.

You can check whether an argument or any object is a vector using `is.vector()`.

Input

```
v <- -1:3
is.vector(v)
```

Output

```
[1] TRUE
```

Remember that in R numbers are just vectors of length 1. So `is.vector(7)` will return a `TRUE` value.

Operators in R are functions, and the default operators are vectorized. Most elementary functions are vectorized as well.

Input

```
sqrtv<-sqrt(v)
is.vector(sqrtv)
```

Output
```
[1] TRUE
```

Input
```
sqrtv
```

Output
```
[1]       NaN 0.000000 1.000000 1.414214 1.732051
```

The first example where vectorization usually breaks in your code are logical conditions, because *if* is not vectorized. So the following line takes a vector *v*, but returns only a vector of length 1.

Input
```
sqrtv <- if (v >=0 ) sqrt(v) else 0
is.vector(sqrtv)
```

Output
```
[1] TRUE
```

Input
```
sqrtv
```

Output
```
[1] 0
```

If you want to implement a function

$$f(x) = \begin{cases} 0 & x < 0 \\ sqrt(x) & x \geq 0 \end{cases}$$

the definition

Input
```
sqrt0 <- function(x){if (x >=0) sqrt(x) else 0}
```

does not work as hoped if *x* is a vector.

In this special case, you can use *ifelse()* which provides a vectorized result.

Exercise 1.1	**Vectorization**
	Write *sqrt0()* as a vectorized function using *ifelse()*.

In more general cases, where a vectorized variant is not available, you have to iterate over the components of *x*. For performance reasons, the iterators provided with R (Appendix

A.9 Accessing Components (page Suppl. A-23)) are to be preferred over *for*-loop on the indices or other ad-hoc solutions.

***Example* 1.1: Vectorization with iterators**

Input

```
sqrt1 <- function(v) {
        sapply(v,
                function(x) {
                        if (x >=0) sqrt(x) else 0}, simplify=TRUE
                )
}
```

To illustrate some technical tricks here, we included the function definition as an anonymous function inline. *sapply()* only relies on the position of the first argument . Argument names do not matter. The elements of *v* are matched to the formal first argument, named *x* in this example.

Of course it is good programming practice to enarmour a general purpose function with guards that check for the appropriate types.

Exercise 1.2	**Vectorization**
	Enhance *sqrt1()* above to check that the type of *v* can be handled correctly. What are the example data types you are using in your test battery?

To facilitate vectorization, a function *Vectorize()* is provided. *Vectorize()* generates an environment (see section Section 1.5.6 (page 4)) and hides your original function there as an object called *FUN*. It then generates a general purpose wrapper to check the arguments and provides vectorization on selected arguments (See Example 1.2 (page 3)).

***Example* 1.2: Vectorize**

Input

```
sqrt2 <- Vectorize(sqrt0)
sqrt2
```

Output

```
function (x)
{
    args <- lapply(as.list(match.call())[-1L], eval, parent.frame())
    names <- if (is.null(names(args)))
        character(length(args))
    else names(args)
    dovec <- names %in% vectorize.args
    do.call("mapply", c(FUN = FUN, args[dovec], MoreArgs = list(args[!dovec]),
        SIMPLIFY = SIMPLIFY, USE.NAMES = USE.NAMES))
}
<environment: 0xfd4f20>
```

Input

```
ls(environment(sqrt2))
```

Output

```
[1] "arg.names"      "FUN"            "FUNV"
[4] "SIMPLIFY"       "USE.NAMES"      "vectorize.args"
```

Input

```
environment(sqrt2)$FUN
```

Output

```
function(x){if (x >=0) sqrt(x) else 0}
```

`Vectorize()` is a general purpose tool. If you can provide a special case solution, this may be a chance for optimization. On the long run, general purpose support for vectorization may be enhanced in R. On the other side, more optimizations will be built into the base machine which may make vectorization less critical. But for now, vectorization is a chance for optimization in your code.

1.5.6 Search Paths, Frames and Environments

To evaluate an expression, the formal terms occurring in the expression must be related to actual terms which can be ultimately evaluated. This requires a search process. As the system has evolved, this search process has become rather complex. We try to give a description here, starting with a very simplified picture, and adding details and variations one by one.

In the R documentation, you find several term which are closely related: frames, environments, closures, etc .... the usage is not always consistent. In particular, ***environment*** may be used in R documentation where ***frame*** would be used in older S terminology. In R terminology, environments can be thought of as consisting of two things. A frame, consisting of a set of symbol-value pairs, and an enclosure, a pointer to an enclosing environment. We take the freedom here to define out own usage of these terms.

If you are starting R, several functions and variables are already pre-defined. These come organized in a chain of ***environments***. The chain starts with an invisible NULL environment. Next *"package:base"* is the basic environment created upon start of the system. Other environments are populated by loading packages. *search()* gives you a list of the currently active search environments. *searchpaths()* gives you information about the path to the underlying package, if appropriate. Using *ls()*, you can inspect any other environment in this list down to *"package:base"*. So *ls("package:base")* gives you a list of the intestines.

For performance reasons, each of these environments is implemented as a data base, called an ***environment*** in R terminology, and a reference to the predecessor environment. You can think of the data base as a list of names, but actually it contains support for caching and other techniques to improve performance. Moreover this environment does not only contain the name of functions and variables, but it contains name/value pairs.

You start working on the top level. R provides a work space for you, the global environment. Functions and variables you define are added here. This work space environment be accessed as *".GlobalEnv"* and *ls(".GlobalEnv")* should return a list of your functions and variables. Just calling *ls()* should give the same the same result. *ls.str()* gives you a look at the structure of each entry and its value.

The path can also be modified under program control. For example, a complex data structure like a *data.frame* can be inserted in the search path using *attach()*. After attaching, the components can be found directly. The components are removed from the search path using *detach()*.

Input

```
search()
```

Output

```
 [1] ".GlobalEnv"        "tools:RGUI"
 [3] "package:stats"     "package:graphics"
 [5] "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"
 [9] "Autoloads"         "package:base"
```

Input

```
#ls()
expl <- data.frame(x=1:3, y= 11:13) # just an example
ls()
```

Output

```
[1] "expl"
```

Input

```
try(x)                          # component not in search path
try(expl$x)                   # component accessible using full name
```

Output

```
[1] 1 2 3
```

Input

```
attach(expl)
search()                  # expl is added after .GlobelEnv
```

Output

```
 [1] ".GlobalEnv"         "expl"
 [3] "tools:RGUI"         "package:stats"
 [5] "package:graphics"   "package:grDevices"
 [7] "package:utils"      "package:datasets"
 [9] "package:methods"    "Autoloads"
[11] "package:base"
```

Input

```
ls()
```

Output

```
[1] "expl"
```

Input

```
try(x)                          # now in search path
```

Output

```
[1] 1 2 3
```

Input

```
ls("expl")              # list objects in the environment attached
```

Output

```
[1] "x" "y"
```

Input

```
detach(expl)
search()
```

Output

```
 [1] ".GlobalEnv"        "tools:RGUI"
 [3] "package:stats"     "package:graphics"
 [5] "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"
 [9] "Autoloads"         "package:base"
```

Input

```
ls()
```

Output

```
[1] "expl"
```

Input

```
try(x)                          # not in search path any more
```

Functions are first class objects in R. Functions in R can have formal parameters. They can also have local variables, and functions can be nested in R. As R is an interpreted language, the effective environment can vary. In particular, there is an environment in which a function is defined, and a (usually different) environment in which the function is called. In R, the preference is that variables in a function are evaluated to the values they have when the function is defined. This is called ***lexical scoping***. An example is given below.

In more detail, functions have three basic components: a formal argument list, a containing environment, and a body. The combination of these three parts forms what is called the ***function closure***. This set defines the lexical scoping. The environment is linking back to the enclosing environment at definition time.

When a variable is requested inside a function, it is first sought in the evaluation environment, then in the enclosure, the enclosure of the enclosure, etc.; once the global environment or the environment of a package is reached, the search continues up the search path to the environment of the base package. If the variable is not found there, the search will proceed next to the empty environment, and will fail.

This construction allows for some optimization. In general, variables are passed by value in R, that is a local copy is generated for each function argument and R functions only operate on the local copy. This causes some time and memory overhead for the copy process. Internally, R uses a lazy evaluation scheme, that is an argument is only evaluated if it is actually needed. Until then, the variable may be treated as a ***promise***, defined by an expression to be evaluated, and the environment to be used for evaluation. If the R interpreter recognizes that an argument is unchanged, the copy step may be omitted.

As a special case, environments are never copied, but passed "as is". So if you have a large data structure, it may be worth considering to hide it as part of the environment, like in the following code fragment which makes use of lexical scoping:

Input

```
definef <- function (x,y, ...) {

setuphugedata <- function()  {
# some function using x, y, ...
}
myhugedata <- setuphugedata()

return( function () {
# some function, possibly using myhugedata
return(myhugedata) # should be some condensed data
})
}
```

The function `Vectorize()` discussed in Section 1.5.3 (page 1) is an example where this facility is exploited.

After this, `f()` will be a function, accessing `myhugedata` without copy. For a detailed discussion of lexical scoping and more examples, see [3].

At a later stage, a function may be called. This may be from the top level, or from within another environment. When a function is called, a new environment is created, whose enclosure is the environment from the function closure. The run time environment from which a function is called is accessible using `sys.calls()` and `sys.frame()`.

Lexical scoping is the preferred (and default) scoping rule. But expression evaluation is under complete control for the programmer. If you want to use the calling environment as a scope, `sys.frame()` allows to accessing variables and functions by call order, and other rules of scope then R's preferred lexical scoping can be used.

The next detail to add is that on the package level, there is a possibility to fine-tune search paths entries. A package may define a ***name space***. Variables and functions are entered into this name space. They may be exported, which will add a reference to the enclosing environment.

With all these possibilities, it is possible that some names are redefined and thus hidden in the search hierarchy. You can however regain hidden definitions by using an explicit reference. So if you have accidentally defined `pi <- 4` and later discovered that the world is not square, you can access the definition of `pi` as given in the base package by using `base::pi`.

CHAPTER 2

Regression

2.5 Beyond Linear Regression

2.5.1 Generalised Linear Models

We want to proceed to practical work. But at this point we should consider how to overcome the limiting assumptions of linear models. Linear models are among the best-investigated statistical models. Theory and algorithms are far advanced. So it is tempting to try to extend this class of models while still allowing ourselves to use the theoretical and algorithmic know-how.

We have formulated the linear model as

$$\begin{aligned} Y &= m(X) + \varepsilon \\ & \quad Y \text{ with values in } \mathbb{R}^n \\ & \quad X \in \mathbb{R}^{n \times p} \\ & \quad E(\varepsilon) = 0 \\ & \quad \text{with } m(X) = X\beta, \quad \beta \in \mathbb{R}^p. \end{aligned}$$

An important extension is to remove the linearity assumption. As an intermediate step, we do not suppose any longer that m is linear, but only that it can be factored using a linear function. This results in a generalised linear model

$$\begin{aligned} Y &= m(X) + \varepsilon \\ & \quad Y \text{ with values in } \mathbb{R}^n \\ & \quad X \in \mathbb{R}^{n \times p} \\ & \quad E(\varepsilon) = 0 \\ & \quad m(X) = \overline{m}(\eta) \text{ with } \eta = X\beta, \ \beta \in \mathbb{R}^p. \end{aligned}$$

The next generalisation at hand is to allow for a transformation for Y. Many more generalisations have been discussed. A small number of them have proven tractable. Most important among these is a group of models called generalised linear models (GLM).

An introduction to generalized linear models is is [2], and an extensive classical survey is [7].

Generalised linear models have extensive support in R. For most of the functions in R for linear models, there is a corresponding function for generalised linear models. For more information see `help(glm)`.

2.6 R Complements

2.6.4 Classes and Polymorphic Functions

Polymorphis and classes are concepts from object oriented programming. Object oriented programming is a programming style that uses ***objects*** as basic elements. Objects conceptually consist of ***data slots*** and ***methods***. Encapsulating data and methods as an object is one aspect.

Object oriented programming uses abstract data types, called ***classes*** which define the data structure and the methods for an object. Classes are arranged in a hierarchy: derived classes inherid the structure of theri predecessor, but can add slots and methods. This inheritance is used to generate specific variants. In object oriented programming, variables are instances of a class. The general structure is defined by the class, but the contents of the data slot may be specific to the instance.

Since functions are first class members of R and functions can be stored for example in components of a list, object oriented programming is a style that can be used with R as presented here. Beyond this, R has support for object oriented programming on the language level. `setClass()` allows to define the structure of a class. `new()` is available to create an instance of a class. For details, see chapter 5 of [12].

R as a Programming Language and Environment

R is an interpreted expression language. Expressions are composed of objects and operators.

A.1 Help and Information

Some R functions such as *library()* or *data()* serve a dual purpose. With minimal arguments, they provide help and information. With specific arguments, they give access to certain components.

R ***Help***	
help()	information about an object/a function. *Example:* *help(help)*
help.start()	starts browser access to R's online documentation. The reference section includes a search engine to search for keywords, function and data names and text in help page titles.
args()	shows arguments of a function.
example()	executes examples, if available. *Example:* *example(plot)*
help.search()	searches for information about an object/a function.
RSiteSearch()	searches for keywords or phrases in the R-help archives or documentation.
apropos()	locates by keyword.
demo()	executes demos for a topic area. *Example:* *demo(graphics)* *demo()* lists all topic areas that provide a demo.

(cont.)→

R ***Help*** (cont.)	
`library()`	gives information about libraries. *Example:* `library()` gives a list of all libraries. `library(help=`⟨package⟩`)` gives information about a package. *Example:* `library(help="stats")` gives information about the basic statistics package.
`data()`	gives information about data sets. *Example:* `data()` lists available data sets.
`vignette()`	lists or views vignette information about a topic. `vignette(all = TRUE)` lists vignettes from all installed packages. *Example:* `vignette("grid")` shows a vignette for the grid graphics.

See also Appendix A.6 “Object Inspection” (page Suppl. A-19) and Appendix A.7 “System Inspection” (page Suppl. A-20).

A.2 Names and Search Paths

Objects are identified by names. By the name objects are searched in a search path, a chain of search environments. The search path in effect can be inspected with `search()`.

R *Search Paths*	
`search()`	lists the search areas in effect, beginning with `.GlobalEnv` down to the base package `package:base`. *Example:* `search()`
`searchpaths()`	lists the access paths for the search areas in effect. *Example:* `searchpaths()`
`objects()`	lists the objects in a search path. *Examples:* `objects()` `objects("package:base")`
`ls()`	lists the objects in a search path. *Examples:* `ls()` `ls("package:base")`
`ls.str()`	lists the objects and their structure in a search path. *Examples:* `ls.str()` `ls.str("package:base")`
`find()`	locates by keyword. Also finds overlaid entries. *Syntax:* `find(what, mode = "any", numeric = FALSE, simple.words = TRUE)`
`apropos()`	locates by keyword. Also finds overlaid entries. *Syntax:* `apropos(what, where = FALSE, ignore.case = TRUE, mode = "any")`

Functions can be nested. This may occur at definition time as well as at execution time. This requires an extension of the search paths. The dynamic identification of objects uses environments to resolve local or global variables in functions.

R *Search Paths (cont.)*	
`environment()`	current environments. *Example:* `environment()`

(cont.)→

R ***Search Paths (cont.)*** (cont.)	
`sys.parent()`	preceding environments. *Example:* `sys.parent(1)`

A.3 Administration and Customisation

objects() *ls()*	lists the objects in the current search path.
rm()	removes indicated objects. *Syntax:* *rm(*⟨object list⟩*)*

R offers a series of possibilities to configure the system so that certain commands are executed upon start or termination. When starting, the files *.Rprofile* and *.RData* are read and executed if available. Details can be system specific. The appropriate information is given by:

```
help(Startup)
```

Various parts of the system keep global information and can be configured by setting options and parameters.

Some System Components with Global State	
basic system	see *help(options)*.
random numbers	see Appendix A.21 (page Suppl. A-49).
basic graphics	see *help(par)*.
lattice graphics	see *help(lattice.options)*.

For information on how to configure memory available for data storage, see:

```
help(Memory)
```

See also Appendix A.7 “System Inspection” (page Suppl. A-20).

A.4 Basic Data Types

Basic* R *Data Types	
numeric	*real* or *integer*. In R: real numbers are always in double precision. Single precision is supported for external call to other languages with .C or .FORTRAN. Functions *mode()* and *typeof()* can show the storage modus (single, double ...), depending on the implementations. *Examples:* *1.0* *2* *3.14E0*
complex	complex, in Cartesian coordinates. *Example:* *1.0+0i*
logical	TRUE, FALSE. In R, T and F are predefined variables provided as an alternative. In S-Plus, T and F are basic objects.
character	character strings. Delimiter are alternatively " or '. *Example:* *"T"*, *'klm'*
list	general list structure. List elements can be of different types. *Example:* *list(1:10, "Hello")*
function	R function. *Example:* *sin*
NULL	special case: empty object. *Example:* *NULL*

is.⟨type⟩*()* tests for a type, *as.*⟨type⟩*()* converts to a type.

In addition to TRUE and FALSE there are three special values for exceptional situations:

Special Constants	
TRUE	alternative: *T*. Type: logical.
FALSE	alternative: *F*. Type: logical.

(cont.)→

Special Constants (cont.)	
NA	"not available". Type: logical. NA is different from TRUE and FALSE.
NaN	"not a valid numeric value". Implementation dependent. Should follow the IEEE Standard 754. Type: numeric. *Example:* `0/0`
Inf	infinite. Implementation dependent. Should follow the IEEE Standard 754. Type: numeric. *Example:* `1/0`

Test Functions	
is.na()	returns *TRUE* if the argument has the value *NA* or *NaN*.
na.omit()	returns an object with the cases containing *NA* removed.
na.fail()	returns its argument if no the case contains *NA*; signals an error message otherwise.
is.nan()	returns *TRUE* if the argument has the value *NaN*.
is.inf()	returns *TRUE* if the argument has the value *Inf* or *-Inf*.

A.5 Output for Objects

Revised from Appendix **??** (page **??**): *str* added.

The object attributes and content can be queried or displayed using output routines. The output routines generally are ***polymorphic***, that is they come with variants adapted to the given object type. To list all available methods for an generic function, or all methods for a class, use *methods()*, for example *methods(print)*.

R ***Inspection***	
print()	standard output.
cat()	outputs the objects, concatenating the representations. *cat()* is useful for producing output in user-defined functions, with minimal formatting.
format()	formats an R object for pretty printing.
structure()	output, optional with attributes.
str()	compact output, optional with attributes.
summary()	standard output as summary, in particular for model fits.
plot()	standard graphic output.

For converting tables to a HTML or LaTeX format, *library(xtable)* [1] is available.

Output of objects to files is discussed in Appendix A.15 “Input and Output to Data Streams” (page Suppl. A-36).

A.6 Object Inspection

Objects have two implicit attributes that can be queried with *mode()* and *length()*. The function *typeof()* gives the (internal) storage modus of an object.

A *class* attribute gives the class of an object.

The following table summarises the most important information possibilities about objects.

Object Inspection	
str()	shows the internal structure of an object in compact form. *Syntax:* *str(*⟨object⟩*)*
structure()	shows the internal structure of an object. Attributes for the display can be passed as parameters. *Example:* *structure(1:6, dim = 2:3)* *Syntax:* *structure(*⟨object⟩, ...*)*
class()	object class. For object classes defined in newer R versions, the class is stored as an attribute. For vintage object classes, the class is determined implicitly by type and other attributes.
mode()	mode (type) of an object.
storage.mode()	storage mode of an object.
typeof()	mode of an object. May be different from the storage mode. Depending on the implementation a numerical variable, for example, can be stored in double precision (the default) or in single precision.
length()	length = number of elements.
attributes()	reads/sets attributes of an object, such as names, dimensions, classes.
names()	names attribute for elements of an object, for example, a vector. *Syntax:* *names(*⟨obj⟩*)* gives the *names* attribute of ⟨obj⟩. *names(*⟨obj⟩*)<-*⟨charvec⟩ sets the *names* attribute. *Example:* *x<-values* *names(x)<-* ⟨charvec⟩

A.7 System Inspection

The following table summarises the most important information possibilities about the general system environment. When used with an argument, these functions generally serve specific purposes, such as setting parameters and options. When used with an empty argument list, they provide inspection.

System Inspection	
`search()`	current search path.
`ls()`	objects in current or selected search path.
`methods()`	generic methods: *Syntax:* `methods(`⟨`fun`⟩`)` shows specialised functions for ⟨`fun`⟩, `methods(class =` ⟨`c`⟩`)` the class-specific functions for class ⟨`c`⟩. *Examples:* `methods(plot)` `methods(class = lm)`
`data()`	accessible data.
`library()`	accessible packages.
`help()`	general help system.
`options()`	global options.
`par()`	parameter settings for the graphics system.
`capabilities()`	reports availability of optional features.

The options of the `lattice` systems can be controlled with `trellis.par.set()` resp. `lattice.options()`.

R is anchored in the host operating system. Some variables such as access paths, encoding, etc. are imported from there.

System Environment	
`getwd()`	gets current working directory.
`setwd()`	sets current working directory.
`dir()`	lists files in the current working directory.
`system()`	calls system functions.

A.8 Complex Data Types

The interpretation of basic types or derived types can be specified by one or more `class` attributes. Polymorphic functions such as `print` or `plot` evaluate this attribute and call a variant for this class if available (see Section 2.6 (page 105)).

For the storage of dates and times, special classes are provided. For more information on these data types see

```
help(DateTimeClasses)
```

and Appendix A.15 (page Suppl. A-36).

R is vector based. Individual constants or values just are vectors with the special length 1. They do not get a special treatment.

Compound Data Types	
Vectors	basic R data types.
Matrices	vectors with two-dimensional layout. *See also* Appendix A.10 "Data Manipulations" (page Suppl. A-26).
Arrays	vectors with higher-dimensional layout. `dim()` defines a dimension attribute. *Example:* `x <- runif(100)` `dim(x) <- c(5, 5, 4)` `array()` generates a new vector with specified dimension structure. *Example:* `z <- array(0, c(4, 3, 2))` *See also* Appendix A.10 "Data Manipulations" (page Suppl. A-26).
Factors	special case for categorical data. `factor()` converts a numeric vector into a factor. *See also* Section 2.2. `ordered()` converts a vector into a factor with ordered levels. This is a shortcut for `factor(x, ..., ordered = TRUE)`.

(cont.)→

Compound Data Types (cont.)	

	`levels()` returns the levels of a factor.
	Example:
	`x <- c("a", "b", "a", "c", "a" )`
	`xf <- factor(x)`
	`levels(xf)`
	results in
	`[ 1 ] "a" "b" "c"`
	`tapply()` applies a function separately for all levels of factors in a list.

Lists	analogous to vectors, with elements of possibly different types.
	`list()` generates a list.
	Syntax: `list`(⟨`components`⟩)
	`[[ ]]` access to components of a list by index.
	⟨`list$component`⟩ access by names.
	Example:
	`l <- list(name = "xyz", age = 22, fak = "math")`
	> `l`[[2]]
	`22`
	`>l$age`
	`22`

Data Frames	**data frames** analogous to arrays resp. lists, with column-wise uniform type and uniform column length. `data.frame()` analogous to `list()`, but restrictions have to be satisfied. `attach()` attaches a database to the current search list. For access to components the component name will be sufficient. `detach()`

A.9 Accessing Components

The length of vectors is a dynamic attribute. It is extended or shortened as needed. In particular, an implicit "recycling rule" applies: if a vector does not have the length necessary for some operation, it is repeated periodically up to the length required.

Vector components can be accessed by index. The indices can be specified explicitly or in the form of an expression rule.

Accessing Components	
`x[⟨indices⟩]`	indicated components of x. *Example:* `x[1:3]`
`x[-⟨indices⟩]`	x omitting indicated components. *Example:* `x[-3]` x omitting the 3. component.
`x[⟨condition⟩]`	components of x, for which the ⟨`condition`⟩ holds. *Example:* `x[x<0.5]`
`which()`	give the indices of a logical object, allowing for array indices.
`subset()`	is a polymorphic function and returns subsets of vectors, matrices or data frames by specified conditions.

Vectors (and other objects) can be mapped to higher-dimensional constructs. The layout is described by a additional `dim` attribute. By convention the imbedding goes by column, that is, the first index varies first (FORTRAN convention). Operators and functions can evaluate the dimension attribute.

R ***Index Access***	
`dim()`	gets or sets dimensions of an object. *Example:* `x <- 1:12; dim (x) <- c(3, 4)`
`dimnames()`	gets or sets names for the dimensions of an object.
`nrow()`	gives the number of rows = dimension 1.
`ncol()`	gives the number of columns = dimension 2.
`matrix()`	generates a matrix with given specifications. *Syntax:* `matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)` *See also* Example 1.11 (page 23)

(cont.)→

R ***Index Access*** (cont.)	
`array()`	generates a possibly higher-dimensional matrix. *Example:* `array(x, dim = length(x), dimnames = NULL)`

`NCOL()` and `NROW()` are variants treating a vector as a one-column resp. as a one-row matrix.

R ***Iterators***	
`apply()`	applies a function to the rows or columns of a matrix. *Syntax:* `apply(x, MARGIN, FUNCTION, ...)` MARGIN = 1: rows, MARGIN = 2: columns *See also* Example 1.11 (page 23).
`lapply()`	applies a function to the elements of a list. *Syntax:* `lapply(X, FUN, ...)`
`sapply()`	applies a function to the elements of a list, of a vector or a matrix. If possible, dimension names are carried over. *Syntax:* `sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)`
`mapply()`	applies a function to multiple list or vector arguments. *Syntax:* `mapply(FUN, ..., MoreArgs = NULL,simplify = TRUE, USE.NAMES = TRUE)`
`Vectorize()`	returns a new function that acts as if `mapply` was called. This can be used as a stepping stone to make a function vectorized. *Syntax:* `Vectorize(FUN, vectorize.args = arg.names,simplify = TRUE, USE.NAMES = TRUE)`
`tapply()`	applies a function to components of an object depending on a list of controlling factors.
`by()`	object-oriented variant of `tapply`. *Syntax:* `by(data, INDICES, FUN, ...)`
`aggregate()`	calculates statistics for subsets. *Syntax:* `aggregate(x, ...)`

(cont.)→

R *Iterators* (cont.)	
`replicate()`	evaluates an expression repeatedly (for example, with generating random numbers for simulation). *Syntax:* `replicate(n, expr, simplify = TRUE)`
`outer()`	generates a matrix with all pair-wise combinations from two vectors, and applies a function to each pair. *Syntax:* `outer(vec1, vec2, FUNCTION, ...)`

A.10 Data Manipulation

Array Access	
`cbind()`	combines by columns.
`rbind()`	combines by rows.
`split()`	splits a vector by factors.
`table()`	generates a table of counts.
`prop.table()`	expresses table entries as fraction of marginal table, i.e., gives relative counts.
`t()`	transposes rows and columns. *Syntax:* `t(x)`
`aperm()`	generalised permutation. *Syntax:* `aperm(x, perm)` where `perm` is a permutation of the indices of `x`.

Transformations	
`duplicated()`	checks for duplicate or multiple values.
`unique()`	generates a vector without multiple values.
`match()`	gives first position of a value in a vector.
`pmatch()`	partial matching

Character String Transformations	
`casefold()`	translates characters, in particular from upper - to lowercase or vice versa.
`tolower()`	translates to lowercase.
`toupper()`	translates to uppercase.
`chartr()`	translates characters in a character vector.
`substr()`	extracts or replaces substrings in a character vector.

(cont.)→

Character String Transformations (cont.)	
`substring()`	extracts or replaces substrings in a text (respects encoding and other attributes)
`paste()`	concatenates vectors after converting to character. See also `cat()`.
`strsplit()`	splits the elements of a character vector into substrings.
`grep()`	pattern matching.
`gsub()`	pattern substitution, by regular patterns.
`abbreviate()`	abbreviates strings.

Transformations	
`table()`	generates a table of counts.
`expand.grid()`	generates a data frame with all combinations of the factors given.
`gl()`	generates factors by specifying the pattern of their levels.
`reshape()`	converts between a cross classification table (column per variable) and a long table (variables in rows, with additional indicator column).
`merge()`	merges data frames. See `help(merge)` for examples. `merge()` supports various versions of data base `join` operations.

Vector Manipulation	
`seq()`	generates a sequence.
`stack()`	concatenates multiple vectors from a data frame or list into a single vector and generates a factor indicating the source of each item. *Syntax:* `stack(x, ...)`
`unstack()`	splits a vector by an indicator variable, i.e., reverses the operation of `stack()`. *Syntax:* `unstack(x, ...)`

(cont.)→

Vector Manipulation (cont.)	
`split()`	splits a vector into the groups defined by a factor. *Syntax:* `split(x, f, drop = FALSE,...)`
`unsplit()`	combines components to a vector, i.e., reverses `split()`. *Syntax:* `unsplit(value, f, drop = FALSE)`
`cut()`	converts a numeric to factor. `cut()` divides the range of a vector into intervals and creates a factor indicating the interval for each value. *Syntax:* `cut(x, ...)`

A.11 Operators

Expressions in R can be composed of objects and operators. The following table of operators is ordered by precedence (highest rank on top). See *help(Syntax)*.

Basic* R *operators	
`$`	select component by name. *Example:* `list$item`
`[ [[`	indexing, access to elements. *Example:* `x[i]`
`^`	exponentiation (right to left). *Example:* `x^3`
`-`	unary minus.
`:`	sequence generation. *Examples:* `1:5` `5:1`
`%`⟨name⟩`%`	special operators. Can also be user defined. *Examples:* `"%deg2%"<-function(a, b) a + b^2` `2 %deg2% 4`
`* /`	multiplication, division.
`+ -`	addition, subtraction.
`< > < = > = == !=`	comparison operators.
`!`	negation.
& | && ||	and, or . &&, || are "shortcut" operators.
`<- ->`	assignment.

If operands do not have the same length, the shorter operand is repeated cyclically.

Operators of the form %⟨name⟩% can be defined by the user. The definition follows the rules for function definitions.

Expressions can be written as a sequence with separating semicolons. Expression groups can be combined by enclosing braces {...}.

A.12 Functions

Functions are special objects. Functions can return objects as results.

R *Function Declarations*	
Declarations	*function* (⟨formal argument list⟩) ⟨expression⟩ *Example:* *fak <- function(n) prod(1:n)*
Formal argument	⟨argument name⟩ ⟨argument name⟩ = ⟨default value⟩
Formal argument list	list of formal argument, separated by commas. *Examples:* *n, mean = 0, sd = 1*
...	variable argument list. Variable argument lists can be propagated to imbedded functions. *Example:* *mean.of.all <- function (...)mean(c(...))*
Function result	*return* ⟨value⟩ stops function evaluation and returns value.
	⟨value⟩ as last expression in a function declaration: returns value.
Assignments	In general, assignments operate only on local copies of variables. Assignments done within a function are temporary. They are lost after exit from the function. The assignment with <<-, however, looks for the target in the complete search chain. It can be used if global and permanent assignments are intended within a function. *Syntax:* ⟨Variable⟩<<-⟨value⟩

R *Function Call*	
Function call	⟨name⟩(⟨Supplied (actual) argument list⟩) *Example:* *fak(3)*
Supplied argument list	Values are matched by position. Deviating from this, names can be used to control the matching. Initial parts of the names suffice (exception: after a variable argument list, names must be given completely). Function *missing()* can be used to check, whether a corresponding actual argument is missing for a formal argument. *Syntax:* ⟨list of values⟩ ⟨argument name⟩ = ⟨values⟩ *Example:* *rnorm(10, sd = 2)*

Arguments for functions are passed by value. This helps consistency, but involves overhead for memory management and copying. If this overhead needs to be avoided, the information provided by *environment()* allows direct access to variables. Techniques to use this are described in [3].

Special case: Functions with names of the form *xxx<-* extend the assignment function. ***Example:***

Input

```
"inc<-" <-function (x, value) x+value
x <- 10
inc(x) <- 3
x
```

Output

```
[1] 13
```

In R assignment functions the value argument **must** be called "value".

A.13 Debugging and Profiling

Revised from Appendix **??** (page **??**): changes in "Profiling Support".

R provides a collection of tools for identification of errors. These are particularly helpful in connection with functions. *browser()* can be used to switch to a browser mode. In this mode, the usual R instructions can be used. Besides this, there is a small number of special instructions. With *debug()*, the browser mode is activated automatically upon entry to a function. The browser mode is marked by a special prompt *Browse[xx]>*.

⟨return⟩	Goes to the next instruction, if the function is under control of *debug*. Continuous with the expression evaluation if *browser* has been called directly.
n	Goes to the next instruction (also if *browser* has been called directly).
cont	Continuous with the expression evaluation.
c	Short for *cont*. Continues the expression evaluation.
where	Shows call nesting.
Q	Stops execution and jumps back to base state.

Debug Help	
browser()	suspends execution and enters the browser mode. *Syntax:* *browser()*
recover()	shows a list of the current call hierarchy. An entry from this list can be chosen for inspection by *browser()*. With *c* you leave the *browser* and return to *recover*. With *0* you leave *recover()* *Syntax:* *recover()* *Hint:* With *options(error = recover)*, error handling for a function is directed to call *browser()* automatically in case of an error.
debug()	marks a function for debugger control. On subsequent calls to the function, the debugger is activated and switches to browser mode. *Syntax:* *debug(*⟨function⟩*)*
undebug()	cancels debugger control for a function. *Syntax:* *undebug(*⟨function⟩*)*
trace()	marks a function for trace control. On subsequent calls to the function, the call is signalled together with its arguments. *Syntax:* *trace(*⟨function⟩*)*
untrace()	cancels trace control for a function. *Syntax:* *untrace(*⟨function⟩*)*

(cont.)→

Debug Help (cont.)	
traceback()	in case of error inside of a function the current calling stack is stored in a variable *.Traceback*. *traceback()* evaluates this variable and displays its content. *Syntax:* *traceback()*
try()	Calls a function. Allows for user-defined error handling. *Syntax:* *traceback(*⟨expression⟩*)*

To measure execution time in selected code ranges, R provides a "profiling". This is only available if R has been compiled with the appropriate options. The options installed at compiling can be queried using *capabilities()*. See Appendix A.7 "System Inspection" (page Suppl. A-20).

Profiling Support	
system.time()	returns the execution time of an expression. This function is available in all implementations. *Syntax:* *system.time(*⟨expr⟩, ⟨gcFirst⟩*)*
Rprof()	records active functions periodically. This function is only available if R has been compiled for "profiling". With *memory.profiling = TRUE*, in addition to the timing the memory usage is recorded periodically. This option is only available if R has been compiled correspondingly. *Syntax:* *Rprof(filename = "Rprof.out", append = FALSE, interval = 0.02, memory.profiling = FALSE)* Use *Rprof(NULL)* to switch off profiling.
Rprofmem()	records memory requirements on demand. This function is only available if R has been compiled for "memory profiling". *Syntax:* *Rprofmem(filename = "Rprofmem.out", append = FALSE, threshold = 0)* Use *Rprofmem(NULL)* to switch off profiling.
summaryRprof()	summarises the output of *Rprof()* and reports the timing by function. *Syntax:* *summaryRprof(filename = "Rprof.out", chunksize = 5000, memory = c("none", "both", "tseries", "stats"), index = 2, diff = TRUE, exclude = NULL)* As an alternative, the Perl script *R CMD Rprof* can be used. See *R CMD Rprof -help* for usage information.

A.14 Control Structures

R *Control Structures*	
if	conditional execution. *Syntax:* *if* (⟨log. expression1⟩) ⟨expression2⟩ The logical *expression1* may return only one logical value. For vector-oriented access use *ifelse*. *Syntax:* *if* (⟨log. expression1⟩) ⟨expression2⟩ *else* ⟨expression3⟩
ifelse	element wise conditional execution. *Syntax:* *ifelse*(⟨log. expression1⟩, ⟨expression2⟩, ⟨expression3⟩) evaluates the logical *expression1* element wise on a vector, and returns *expression2* if the evaluation gives true, else *expression3*. *Example:* `trimmedX <- ifelse (abs(X)<2,  x, sign(X)*2)`

switch	evaluates an expression and executes an instruction based on the result.
	Syntax: *switch*(⟨expression1⟩, ...)
	expression1 must return a numeric value or a character string. ... is an explicit list of alternative actions.
	Example:
	`centre <- function (x , type) { switch(type,`
	`mean = mean(x),`
	`median = median(x),`
	`trimmed =  mean(x,  trim = .1)}`

for	iteration (loop). *Syntax:* *for* (⟨name⟩ *in* ⟨expression1⟩) ⟨expression2⟩

repeat	iteration. Must be terminated explicitly, for example with *break*.
	Syntax: *repeat* ⟨expression⟩
	Example:
	`pars<-init`
	`repeat { res<- get.resid  (data, pars)`
	`if (converged(res) ) break`
	`pars<-new.fit (data, pars)}`

(cont.)→

R *Control Structures* (cont.)	
while	conditional repetitions. *Syntax:* *while* (⟨log. expression⟩) ⟨expression⟩ *Example:* *pars<-init; res <- get.resid (data, pars)* *while (!converged(res))* { *pars<-* *new.fit(data, pars)* *res<- get.resid}*
break	terminates the current loop and exits.
next	terminates the current loop cycle and advances to next cycle.

Note: In R loops should be avoided if possible in favour of more efficient language constructs (see [6]).

A.15 Input and Output to Data Streams; External Data

R *Input/ Output*	
write()	writes data to a file. *Syntax:* *write(val, file)* *Example:* *write(x, file = "data")*
source()	executes the R instruction from the file indicated. *Syntax:* *source("⟨file name⟩")* *Example:* *source("cmnds.R")*
Sweave()	executes the R instruction from the file indicated and entangles embedded text. Sweave can be used for automatic report generation. *Syntax:* *Sweave("⟨file name⟩", ...)*
sink()	redirects output in the file specified. *Syntax:* *sink*("⟨file name⟩") *Example:* *sink()* redirects the output back to the console.
dump()	writes the commands defining an object. The object can be regenerated from this output using *source()*. *Syntax:* *dump(list, file =* "⟨dumpdata.R⟩", *append = FALSE)*

R can access data from local files indicated by a usual file path or from remote files accessed by an URL reference. On most systems, direct access to a clipboard is available as well. More system-specific information is available using *help(connections)*.

To edit or enter data, R provides *edit()*. This is a polymorphic function For the special case of matrix-like data, *data.entry()* is provided, using a spreadsheet model.

For exchange, the data formats have to be harmonised between all parties. For import from data bases or other systems, several packages are available, for example *library(foreign)* for Stata, SAS, Minitab and SPSS, *library(RODBC)* for SQL. For more information, see the manual "Data Import/Export" [8].

Within R, prepared data are usually provided as *data frames*. If additional objects such as functions or parameters are necessary, they can be made accessible in bundled form as packages. See Appendix A.16 (page Suppl. A-39).

For the exchange from R to R, a special exchange format can be used. Files in this format can be generated with *save()* and conventionally have the name suffix *.Rda*. These files can be loaded again using *load()*.

A general purpose function to load data us *data()*. Depending on the suffix of the input file name, *data()* branches for several special cases. Besides *.Rda* usual suffixes for data input files are *.tab* or *.txt*. The online help function *help(data)* gives additional information.

Data Input/Output for R	
save()	stores data in an external file. *Syntax:* *save(*⟨names of the objects to be stored⟩*, file =* ⟨file name⟩*, ...)*
save.image()	is a short-cut and stores data of the workspace in an external file.
load()	loads data from an external file. *Syntax:* *load(file =* ⟨file name⟩*, ...)*
data()	loads data. *data()* can handle various file formats, if the access paths and filenames follow the R conventions. *Syntax:* *data(... , list = character(0), package = c(.packages(), .Autoloaded), lib.loc = .lib.loc)* *Example:* *data(crimes) # loads the data set 'crimes'*

For the flexible exchange with other programs in general text-based files are provided. Some conventions can make exchange easier:

- in table form
- only ASCII characters (for example, no umlaut!)
- variables arranged in columns
- columns separated by tabulator stops
- possibly a column header in row 1
- possibly a row label in column 1

For reading the function *read.table()* is provided, and for writing, there is *write.table()*. Besides *read.table()* there are several variants that are adapted to usual data formats. These are documented under *help(read.table)*.

Input and Output of Data for Exchange	
read.table()	reads data tables. *Syntax:* *read.table(file, header = FALSE, sep = "\t", ...)* *Examples:* *read.table(*⟨file name⟩*, header = TRUE, sep = "\t")* headers in row 1, row labels in column 1 *read.table(*⟨file name⟩*, header = TRUE, sep = '\t')* now row number, headers in row 1,

(cont.)→

Input and Output of Data for Exchange (cont.)	
write.table()	writes data table. *Syntax:* *write.table(file, header = FALSE, sep = '\t', ...)* *Examples:* *write.table(*⟨data frame⟩, ⟨file name⟩*, header = TRUE, sep = '\t')* headers in row 1, row labels in column 1 *write.table(*⟨data frame⟩, ⟨file name⟩, *header = TRUE, sep = '\t')* now row number, headers in row 1.
read.csv()	reads comma-separated data tables.
write.csv()	writes comma-separated data tables.
read.csv2()	reads semicolon-separated data tables, using a comma as decimal separator.
write.csv2()	writes semicolon-separated data tables, using a comma as decimal separator.

By default, *read.table()* converts data to *factor* variables if possible. This behaviour can be modified with the argument *as.is* when calling of *read.table()*. This modification is, for example, necessary to read date and time information as for example in the following example from [4]:

```
# date col in all numeric format yyyymmdd
df <- read.table("laketemp.txt", header = TRUE)
as.Date(as.character(df$date), "%Y-%m-%d")
# first two cols in format mm/dd/yy hh:mm:ss
# Note as.is = in read.table to force character
library("chron")
df <- read.table("oxygen.txt", header = TRUE,  as.is = 1:2)
chron(df$date, df$time)
```

For sequential reading, *scan()* is provided. Files with data in fixed format (by character columns) can be read with *read.fwf()*.

A.16 Libraries, Packages

External information can be stored in (text) files and packages. In general, additional functions are provided as packages. Packages may be installed as part of the basic installation or installed by the user. Once packages are installed, they are loaded with

```
library()
```

when needed. Data sets contained in the package are then included in the search path and can be listed using `data()` without arguments:

```
data()
```

Example:

```
library(nls)
data()
data(Puromycin)
```

If you use R packages, please treat them as you would treat any other scientific source of information. Credit should be given where credit is due, and proper citations should be included. The function `citation()` gives the bibliographic information to use.

Package Utilities	
`install.packages()`	installs add-on package in ⟨`lib`⟩, downloading it from the archive `CRAN` or from specified archives. *Syntax:* `install.packages(pkgs, lib, CRAN = getOption("CRAN"), ...)` *Example:* `install.packages("mypackage.tgz", repos=NULL` installs package from a local file.
`library()`	loads an installed add-on package into the current workspace. *Syntax:* `library(package, ...)` *See also* Section 1.5 "Packages" (page 54).
`require()`	tries to load an add-on package; gives warning on error. *Syntax:* `require(package, ...)`
`detach()`	releases an add-on package and removes it from the search path. *Syntax:* `detach(`⟨`name`⟩`)`
`package.manager()`	if implemented, interface for management of installed packages. *Syntax:* `package.manager()`
`package.skeleton()`	creates a skeleton for a new package. *Syntax:* `package.skeleton(name = "`⟨`anRpackage`⟩`", list, ...)`
`citation()`	gives bibliographic information for citing a package. *Syntax:* `citation(`⟨`package name`⟩`, lib.loc = NULL)`

For Unix/Linux/Mac OS X, the main tools are available as commands:

```
R CMD check <directory> # checks a directory for compliance with the R conventions
R CMD build <directory> # generates an R package
```

Detailed information for building R packages is in “Writing R Extensions” ([10]).

A.17 Mathematical Operators and Functions; Linear Algebra

For basic arithmetic operators, see *help(Arithmetic)*. For trigonometric functions, information is available using *help(Trig)*. For special mathematical functions, including *beta()*, *factorial()*, *choose()*, see *help(Special)*.

For linear algebra, the most important functions are widely standardised and implemented in C libraries such as BLAS/ATLAS and LAPACK. R makes use of these libraries and provides an interface to the most important functions.

Linear Algebra	
t()	transposes a matrix.
diag()	generates a diagonal matrix.
%%*	matrix multiplication.
rowsum()	gives row sums for a matrix.
colsum()	gives column sums for a matrix.
rowMeans()	gives row means for a matrix.
colMeans()	gives column means for a matrix.
eigen()	computes eigenvalues and eigenvectors of real or complex matrices.
svd()	singular value decomposition of a matrix.
qr()	QR decomposition of a matrix.
determinant()	determinant of a matrix.
solve()	solves linear equations, or computes inverse.

If possible, statistical functions should be used and direct access to the linear algebra functions should be avoided.

Optimisation and Fitting	
optim()	general purpose optimisation.
nlm()	carries out a minimisation of a function using a Newton-type algorithm.
lm()	fits a linear model.
glm()	fits a generalised linear model.
nls()	determines the non-linear (weighted) least-squares estimates of the parameters of a (possibly non-linear) model.
approx()	linear interpolation.
spline()	cubic spline interpolation.

Use the online help functions and search for the keyword *smooth* to find more fitting methods.

A.18 Model Descriptions

Mathematically, linear statistical models can be specified by a design matrix X and written generally as

$$\mathtt{Y} = \mathtt{X}\beta + \varepsilon,$$

where the matrix `X` has to be specified.

R allows us to specify models by giving the rules for how to build the design matrix.

Operator	Syntax	Meaning	Example
$\sim$	$Y \sim M$	Y depends on M	$Y \sim X$ results in $E(Y) = a + bX$
$+$	$M_1 + M_2$	include M_1 and M_2	$Y \sim X + Z$ $E(Y) =$ $a + bX + cZ$
$-$	$M_1 - M_2$	include M_1, but exclude M_2	$Y \sim X - 1$ $E(Y) = bX$
:	$M_1 : M_2$	tensor product, that is, all combinations of levels of M_1 and M_2	
% in %	M_1% in %M_2	modified tensor product	$a + b\%in\%a$ corresponds to $a + a : b$
$*$	$M_1 * M_2$	"crossed"	$M_1 + M_2$ corresponds to $M_1 + M_2 + M_1 : M_2$
/	M_1/M_2	"nested": $M_1 + M_2$ $\%in\%$ M_1	
^	$M\hat{}n$	M with all "interactions" up to level n	
$I()$	$I(M)$	interpret M; terms in M retain their original meaning; the result determines the model	$Y \sim$ `(1 + I(X^2))` corresponds to $E(Y) = a + bX^2$

Table A.34 *Wilkinson-Rogers Notation for Linear Models*

The model specification is also possible for generalised (not linear) models.

Examples:

`y ~ 1 + x`	corresponds to $\mathtt{y_i} = (\mathtt{1}\ \mathtt{x_i})(\beta_1\ \beta_2)^\top + \varepsilon$
`y ~ x`	short for `y ~ 1 + x` (a constant term is assumed implicitly)

`y ~ 0 + x`	corresponds to $\mathtt{y_i} = \mathtt{x_i} \cdot \beta + \varepsilon$
`log(y) ~ x1 + x2`	corresponds to $\log(\mathtt{y_i}) = (\mathtt{1}\ \mathtt{x_{i1}}\ \mathtt{x_{i2}})(\beta_1\ \beta_2\ \beta_3)^\top + \varepsilon$ (a constant term is assumed implicitly)
`y ~ A`	one-way analysis of variance with factor `A`
`y ~ A + x`	covariance analysis with factor `A` and covariable `x`
`y ~ A * B`	two-factor crossed layout with factors `A` and `B`
`y ~ A/B`	two-factor hierarchical layout with factor `A` and sub-factor `B`

For an economic transition between models, for example for model comparison, the function `update()` is available. It updates and (by default) re-fits a model by extracting the call stored in the object, updating the call and evaluating that call, given the new information. In particular, it can be used to re-fit a model to a changed (possibly corrected) data set.

Model Administration	
`formula()`	extracts a model formula from an object.
`terms()`	extracts terms of the model formula from an object.
`contrasts()`	specifies contrasts.
`update()`	updates and re-fits, or changes a model.
`model.matrix()`	generates the design matrix for a model.

Example:

```
lm(y ~ poly(x, 4), data = experiment)
```

analyses the data set "experiment" with a linear model for polynomial regression of degree 4.

Standard Analysis	
`lm()`	linear model. *See also* Chapter 2.
`glm()`	generalised linear model.
`nls()`	non-linear least squares.
`nlm()`	general non-linear minimisation.
`update()`	update and re-fit, or change a model.
`anova()`	analysis of variance.

A.19 Graphic Functions

R provides two graphics systems: The basic graphics system of R implements a model that is oriented at pen and paper drawing. The lattice graphics system is an additional second graphics system that is oriented at a viewport/object model. For information about lattice see `help(lattice)`. For a survey about the functions in lattice see `library(help = lattice)`. Information about the basic graphics system follows here. Additional graphics systems are available as packages.

Graphic functions fall essentially in three groups:

"high level" functions.	These define a new output.
"low level" functions.	These modify an existing output.
parametrisations.	These modify the settings of the graphics system.

A.19.1 High-Level Graphics

High-Level Graphics	
`plot()`	generic graphic output.
`pairs()`	pair-wise scatterplots.
`coplot()`	scatterplots, conditioned on covariables.
`qqplot()`	QQ Plot.
`qqnorm()`	Gaussian QQ Plot.
`qqline()`	adds a line to a Gaussian QQ Plot, passing through the first and third quartile.
`hist()`	histogram. See also Section 1.3, page 16.
`boxplot()`	box-and-whisker plot.
`dotchart()`	draws a Cleveland dot plot.
`curve()`	evaluates a function or an expression and draws a curve. *Example:* `curve(dnorm, from = -3, to = 3)`
`image()`	colour coded z against x, y.
`contour()`	contour plot of z against x, y.
`persp()`	3D surface.
`matplot()`	plots the columns of one matrix against the columns of another.
`mosaicplot()`	mosaic displays to visualise (standardised) residuals of a log-linear model for the table.
`termplot()`	plots regression terms against their predictors, optionally with standard errors and partial residuals added.

Corresponding function names for the `lattice` graphics are in Table 4.5 (page 147).

A.19.2 Low-Level Graphics

Most high-level functions have an argument `add`. If the function is called with `add = FALSE`, it can be used to add elements to an existing plot. Moreover, there are several low-level functions that suppose that there is already a defined plot environment. This is usually set by high-level functions, but may be modified by `par()`: Besides the physical layout, information about the scales, such as range and possible logarithmic transformations, are part of the environment.

Low-Level Plotting	
`points()`	generic function. Marks points at specified positions. *Syntax:* `points(x, ...)`
`symbols()`	draws symbols at selected points.
`text()`	adds text labels at selected points.
`lines()`	generic function. Joins points at specified positions. *Syntax:* `lines(x, ...)`
`segments()`	adds line segments.
`abline()`	adds a line (in several representations) to a plot. *Syntax:* `abline(a, b, ...)`
`arrows()`	adds a line with arrows to a plot.
`polygon()`	adds polygon with specified vertices.
`rect()`	draws a rectangle.
`axis()`	adds axis.
`rug()`	adds a rug marking the data points.

Besides this, R has rudimentary possibilities for interaction with graphics.

Interactions	
`devAskNewPage()`	controls if a console prompt is given before starting a page of output.

`locator()`	determines the position of mouse clicks.
	A current graphics display has to be defined before `locator()` is used.
	Example:
	`plot(runif(19))`
	`locator(n = 3, type = "l")`

`Sys.sleep()`	suspends execution for a time interval. *Syntax:* `Sys.sleep(`⟨`seconds`⟩`)`

(cont.)→

Interactions (cont.)	
getGraphicsEvent()	waits for a keyboard or mouse event. Functions to respond to these events can be specified. This function needs a graphics display that supports graphics events.

For more interactive facilities, see additional packages, in particular:

- rgl implements OpenGL for real-time 3d rendering,
- rggobi interfaces to the ggobi system for higher-dimensional exploration of data.

A.19.3 Annotations and Legends

The high-level functions generally offer the possibilities to add standard annotations by using arguments:

main =	main title, above the plot,
sub =	plot caption, below the plot,
xlab =	label for the x axis,
ylab =	label for the y axis.

For documentation, see *help(plot.default)*.

High-level functions are complemented by low-level functions.

Low Level Annotation	
title()	adds main title, analogous to high-level argument main. *Syntax:* *title(main = NULL, sub = NULL, xlab = NULL, ylab = NULL, ...)*
text()	adds text at specified coordinates. *Syntax:* *text(x, y = NULL, text, ...)*
legend()	adds a legend block. *Syntax:* *legend(x, y = NULL, text, ...)* *Example:* *plot(runif(100)); legend(locator(1), legend = "You clicked here")*
mtext()	adds text to margin. *Syntax:* *mtext(text, side = 3, ...)*. The margins are denoted by 1 = bottom, 2 = left, 3 = top, 4 = right).

For annotations, texts some times has to be shortened. Function and variable names can be shortened using `abbreviate()`.

R gives (limited) possibilities for mathematical typesetting. If the text argument is a character string, it is taken directly. If the text argument is an (unevaluated) R expression, R tries to render the expression as usual in a mathematical formula. R expressions can be generated using the functions `expression()` and evaluated with `eval()` or `bquote()`.

Example:

```
text(x, y,  expression(paste(bquote("(", atop(n, x), ")"),
    .(p)\^{}x, .(q)\^{}\n-x\})))
```

`demo(plotmath)` gives several examples for mathematical typesetting in plots.

A.19.4 Graphic Parameters and Layout

Parametrisations	
`par()`	sets parameters for the basic graphics system. *Syntax:* see `help(par)`. *Example:* `par(mfrow = c(m, n))` splits the graphic area in m rows and n columns, to be filled row-wise. `par(mfcol = c(m, n))` fills the area column by column.
`lattice.options()`	sets parameters for the lattice graphics system. *Syntax:* see `help(lattice.options)`
`split.screen()`	splits the graphic area in parts. *Syntax:* `split.screen(figs, screen, erase = TRUE)`. If `figs` is a pair of two arguments, these will fix the number of rows and columns. If `figs` is a matrix, each row gives the coordinates of a graphic area in relative coordinates $[0 \ldots 1]$. `split.screen()` can be nested.
`screen()`	selects graphic area for the next graphical output. *Syntax:* `screen(n = cur.screen, new = TRUE)`.
`layout()`	divides the graphic area. This function is not compatible with other layout functions.

A.20 Elementary Statistical Functions

Statistical Functions	
sum()	sums up components of a vector.
cumsum()	calculates cumulated sums.
prod()	multiplies components of a vector.
cumprod()	calculates cumulated products.
length()	length of an object, for example a vector.
max() *min()*	maximum, minimum. See also *pmax, pmin*.
range()	minimum and maximum.
cummax() *cummin()*	cumulated maximum, minimum.
quantile()	sample quantile. For theoretical distributions, use *qxxxx*, for example *qnorm*.
median()	median.
mean()	mean, including trimmed mean.
var()	variance, variance / covariance matrix.
sort()	sorting.
rev()	reverse sorting.
order()	returns a permutation for sorting.
which.max()	index of the (first) maximum of a numeric vector.
which.min()	index of the (first) minimum of a numeric vector.
rank()	rank in a sample.

A.21 Distributions, Random Numbers, Densities...

The base generator for uniform random numbers is administered by *Random*. Several types of generators are available as base generator. **For serious simulation it is strongly recommended to read the recommendations of Marsaglia et al.** (see *help(.Random.seed)*). All non-uniform random number generators are derived from the current base generator. A survey of most important non-uniform random number generators, their distribution functions and their quantiles is given at the end of this section.

R *Random Numbers*	
.Random.seed	*.Random.seed* is a global variable that holds the current state of the basic random number generator. This variable can be stored and later be restored with *set.seed()*. Initially, there is no seed. Use *set.seed()* to define a seed. If no seed has been defined, a new one is created based on the current clock time when one is required. Random number generators may use variables other than *.Random.seed* to store their state information. To set a generator to a defined state, always use *set.seed()*. Never set *.Random.seed* directly.
set.seed()	initialises the random number generator. *Syntax:* *set.seed(seed, kind = NULL)*
RNGkind()	*RNGkind()* gives the name of the current base generator. *RNGkind(*⟨name⟩*)* sets a basic random number generator. *Syntax:* *RNGkind()* RNGkind(⟨name⟩) *Example:* *RNGkind("Wichmann-Hill")* *RNGkind("Marsaglia-Multicarry")* *RNGkind("Super-Duper")*
sample()	draws a sample from the values given in vector *x*, with or without replacement (controlled by the value of *replace*). Size is by default the length of *x*. Optionally, *prob* can be a vector of probabilities for the values of *x*. *Syntax:* *sample(x, size, replace = FALSE, prob)* *Example:* Random permutation: *sample(x)* Biased coin: *val<-c("H", "T"")* *prob<-c(0.3, 0.7)* *sample(val, 10, replace = TRUE, prob)*

If simulations shall be reproducible, the random number generator must be set to a well-defined initial state for a reproduction. So the initial state needs to be recorded. An example is the following statement sequence to store the current state:

```
save.seed <- .Random.seed
save.kind <- RNGkind()
```

These variables can be stored to a file and read from there when necessary. With

```
set.seed(save.seed, save.kind)
```

the state of the random number generator is then restored.

The individual function names for the common non-uniform generators and distribution functions are combined from a prefix and the short name of the distribution (see the list below). General pattern: if *xxxx* is the short name, then

rxxxx	generates random numbers
dxxxx	density or probability
pxxxx	distribution function
qxxxx	quantiles

Example:

x<-runif(100)	generates 100 random variables with $\mathtt{U}(0,1)$ distribution.
qf(0.95, 10, 2)	calculates the 95% quantile of the $\mathtt{F}(10,2)$ distribution.

Distributions	***Short Name***	***Parameter and Default Values***
Beta	*beta*	*shape1, shape2, ncp = 0*
Binomial	*binom*	*size, prob*
Cauchy	*cauchy*	*location = 0, scale = 1*
χ^2	*chisq*	*df, ncp = 0*
Exponential	*exp*	*rate = 1*
F	*f*	*df1, df2 (ncp = 0)*
Gamma	*gamma*	*shape, scale = 1*
Gauss	*norm*	*mean = 0, sd = 1*
Geometric	*geom*	*prob*
Hypergeometric	*hyper*	*m, n, k*
Lognormal	*lnorm*	*meanlog = 0, sdlog = 1*
Logistic	*logis*	*location = 0, scale = 1*
Negativ-Binomial	*nbinom*	*size, prob*
Poisson	*pois*	*lambda*
Student's t	*t*	*df*

(cont.)→

Distributions	*Short Name*	*Parameter and Default Values*
Tukey Studentised Range	`tukey`	
Uniform	`unif`	`min = 0, max = 1`
Wilcoxon Signed Rank	`signrank`	`n`
Wilcoxon Rank Sum	`wilcox`	`m, n`
Weibull	`weibull`	`shape, scale = 1`

Additional support for generating random numbers is provided by `library(distr)` [11].

A.22 Computing on the Language

The language elements of R are objects, as are data or functions. They can be read and changed like any other data or functions. Chapter 6 of the "R Language Definition" [9] gives details for computing on the language. See also Section 2.1.5, "Function objects" of [9].

Conversions	
`parse()`	converts input into a list of R expressions. `parse` executes the parse, but does not evaluate the expression.
`deparse()`	converts an R expression given in internal representation into a character string.
`expression()`	generates an R expression in internal representations. *Example:* `integrate <- expression(integral(fun, lims))` *See also* 1.3: mathematical typesetting in plot annotations
`substitute()`	R expression with evaluation of all defined terms.
`bquote()`	R expression with selective evaluation. Terms in `.()` are evaluated. *Examples:* `n<-10; bquote( n^2 == .(n*n))`

Evaluation	
`eval()`	evaluates an expression.

References

[1] David B. Dahl and contributions from many others. *xtable: Export Tables to LaTeX or HTML*, 2008. R package version 1.5-3.

[2] D. Firth. Generalized linear models. In D.V. Hinkley, N. Reid, and E.J. Snell, editors, *Statistical Theory and Modeling*, chapter 3, pages 55–82. Chapman and Hall, London, 1991.

[3] Robert Gentleman and Ross Ihaka. Lexical scope and statistical computing. *Journal of Computational and Graphical Statistics*, 9:491–508, 2000.

[4] Gabor Grothendieck and Thomas Petzoldt. R help desk: Date and time classes in R. *R News*, 4(1):29–32, June 2004.

[5] David James and Kurt Hornik. *chron: Chronological Objects Which Can Handle Dates and Times*, 2008. R package version 2.3-24. S original by David James, R port by Kurt Hornik.

[6] Uwe Ligges and John Fox. R help desk: How can I avoid this loop or make it faster? *R News*, 8(1):46–50, May 2008.

[7] P. McCullagh and J.A. Nelder. *Generalized linear models*. Number 37 in Monographs on statistics and applied probability. London : Chapman & Hall, 2nd edition, 1989.

[8] R Development Core Team. *R Data Import/Export*, 2008.

[9] R Development Core Team. *The R language definition*, 2008.

[10] R Development Core Team. *Writing R Extensions*, 2008.

[11] Peter Ruckdeschel, Matthias Kohl, Thomas Stabla, and Florian Camphausen. S4 classes for distributions. *R News*, 6(2):2–6, May 2006.

[12] William N. Venables and Brian D. Ripley. *S Programming*. Statistics and Computing. Springer, New York, 2000.