Computational Statistics

An Introduction to C



Supplement

Günther Sawitzki





Supplement to

Computational Statistics

An Introduction to



Günther Sawitzki StatLab Heidelberg November 20, 2011

 $in\ preparation$

 $http://sintro.r\hbox{-}forge.r\hbox{-}project.org/$





Introduction

This is a supplement to

G. Sawitzki

 $Computational\ Statistics:\ An\ Introduction\ to\ R$

CRC Press, Boca Raton, 2009

ISBN: 978 14 20 08 6782



The supplement contains 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: November 20, 2011



Contents

In	Introduction		\mathbf{v}	
1	Bas	ic Dat	ta Analysis	1
	1.5	R Co	mplements	1
		1.5.3	Functions	1
			Vectorisation	1
			Compilation	5
		1.5.6	Search Paths, Frames and Environments	6
2	Reg	gressio	on	11
		2.2.4	Least Squares Estimation	11
		2.2.5	Regression Diagnostics	18
		2.2.6	Gauss-Markov Estimator	26
	2.5	Beyon	nd Linear Regression	28
		2.5.1	Generalised Linear Models	28
	2.6	R Co	mplements	29
		2.6.4	Classes and Polymorphic Functions	29
3	Cor	nparis	sons	31
	3.1	Shift/	Scale Families, and Stochastic Order	31
	3.3	Tests	for Shift Alternatives	32
4	Din	nensio	ons $1,2,3,\ldots,\infty$	33
	4.1	R Co	mplements	33

	$\alpha \alpha r r r$	TINTER
V111	CONT	ENTS

R as a Programming Language and Environment	Suppl.A-39
A.1 Help and Information	Suppl.A-39
A.2 Names and Search Paths	Suppl.A-41
A.3 Administration and Customisation	Suppl.A-43
A.4 Basic Data Types	Suppl.A-44
A.5 Output for Objects	Suppl.A-46
A.6 Object Inspection	Suppl.A-47
A.7 System Inspection	Suppl.A-48
A.8 Complex Data Types	Suppl.A-49
A.9 Accessing Components	Suppl.A-51
A.10 Data Manipulation	Suppl.A-54
A.11 Operators	Suppl.A-57
A.12 Functions	Suppl.A-58
A.13 Debugging and Profiling	Suppl.A-60
A.14 Control Structures	Suppl.A-62
A.15 Input and Output to Data Streams; External Data	Suppl.A-64
A.16 Libraries, Packages	Suppl.A-67
A.17 Mathematical Functions; Linear Algebra	Suppl.A-69
A.18 Model Descriptions and Diagnostics	Suppl.A-70
A.19 Graphic Functions	Suppl.A-73
A.19.1 High-Level Graphics	Suppl.A-73
A.19.2 Low-Level Graphics	Suppl.A-74
A.19.3 Annotations and Legends	Suppl.A-75
A.19.4 Graphic Parameters and Layout	Suppl.A-76
A.20 Elementary Statistical Functions	Suppl.A-78
A.21 Distributions, Random Numbers, Densities	Suppl.A-79
A.22 Computing on the Language	Suppl.A-82
References	83
Functions and Variables by Topic	85
Function and Variable Index	01



CHAPTER 1

Basic Data Analysis

1.5 R Complements

1.5.3 Complements: Functions

Vectorisation

In R, functions preferably are vectorised. 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 vectorised. So, for example, <code>curve()</code> or <code>integrate()</code> only accept a function passed as first argument if it is vectorised.

Unfortunately there is no easy way to check whether a function is vectorised. 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 vectorised. You can check whether an argument or any object is a vector using is.vector().

	_ Input
v <1:3	
<pre>is.vector(v)</pre>	
	Output
[1] TRUE	- output
Remember that in R numbers are just v	vectors of length 1. So is.vector(7) will return
a TRUE value.	

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

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

	Output
[1] TRUE	- Odopao
sqrtv	Input
-	Output
[1]	NaN 0.000000 1.000000 1.414214 1.732051

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

If you want to implement a function

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

the definition

does not work as hoped if x is a vector.

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

Exercise 1.1	Vectorisation
	Write sqrt0() as a vectorised function using ifelse().

In more general cases, where a vectorised variant is not available, you have to iterate over the components of \mathbf{x} . For performance reasons, the iterators provided with R (such

R COMPLEMENTS 3

as those listed in Appendix A.9 Accessing Components (page Suppl.A-51)) are to be preferred over *for*-loop on the indices or other ad-hoc solutions.

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 armour a general purpose function with guards that check for the appropriate types.

Exercise 1.2	Vectorisation
	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 vectorisation, a function Vectorize() is provided. Vectorize() generates an environment (see section Section 1.5.6 (page 6)) and hides your original function there as an object called FUN. It then generates a general purpose wrapper to check the arguments and provides vectorisation on selected arguments (See Example 1.2 (page 3)).

```
Example 1.2: Vectorize
sqrt2 <- Vectorize(sqrt0)</pre>
sqrt2
                                        Output
function (x)
    args <- lapply(as.list(match.call())[-1L], eval, parent.frame())</pre>
    names <- if (is.null(names(args)))</pre>
        character(length(args))
    else names(args)
    dovec <- names %in% vectorize.args</pre>
    do.call("mapply", c(FUN = FUN, args[dovec], MoreArgs = list(args[!dovec]);
        SIMPLIFY = SIMPLIFY, USE.NAMES = USE.NAMES))
<environment: 0x101df0208>
                                        Input -
ls(environment(sqrt2))
                                        Output __
"FUNV"
[1] "arg.names"
                        "FUN"
[4] "SIMPLIFY"
                        "USE.NAMES"
                                            "vectorize.args"
                                         Input
 environment(sqrt2)$FUN
                                        Output

\frac{1}{\text{function}(x)\{\text{if }(x \geq 0) \text{ sqrt}(x) \text{ else } 0\}}
```

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

R COMPLEMENTS 5

Compilation

As of R version 2.13, R supports compilation and optimisation on a compiler level, and with version 2.14 this is widely used for many packages.

The basic packages already come in byte compiled form as of version 2.14. No further action is necessary.

Additional packages may come in original (interpretable) form, or in compiled form. A listing of a function in compiled form will show it marked as bytecode at the end of the listing.

Uncompiled packages can be compiled on demand by using <code>install.packages()</code> with appropriate options method="source", INSTALL_opts="--byte-compile".

Usage

Arguments

pkgs

character vector of the names of packages whose current versions should be downloaded from the repositories.

If repos = NULL, a character vector of file paths of '.tar.gz' files. These can be source archives or binary package archive files (as created by R CMD build --binary). Tilde-expansion will be done on the file paths.

If this is missing or a zero-length character vector, a listbox of available packages is presented where possible.

. . .

INSTALL_opts

an optional character vector of additional option(s) to be passed to R CMD INSTALL for a source package install. E.g. c("--html", "--no-multiarch"). Use INSTALL_opts="--byte-compile" for compilation

To compile a package and install it on command level, use: R CMD INSTALL

If you are writing a package, you can mark it for compilation upon build by inserting

ByteCompile=TRUE

in the Description file.

Individual functions can be compiled by using cmpfun() in library(compiler). Other compilation facilities are available in library(compiler).

Byte-compiling can be used with various optimisation levels. These are subject to change. The current default level is 2. Level 3 can be used with the functions provided in library(compiler), but should be used with caution.

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. This section goes into some technical details and can be skipped on first reading.

In the R documentation, you find several terms which are closely related: frames, environments, closures, 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 organised in a chain of <code>environments</code>. 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. <code>search()</code> gives you a list of the currently active search environments. <code>searchpaths()</code> gives you information about the path to the underlying package, if appropriate. Using <code>ls()</code>, you can inspect any other environment in this list down to "package:base". So <code>ls("package:base")</code> 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

R COMPLEMENTS 7

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 <code>data.frame</code> can be inserted in the search path using <code>attach()</code>. After attaching, the components can be found directly. The components are removed from the search path using <code>detach()</code>.

	Input
search()	
	Output
[1] ".GlobalEnv" [3] "tools:RGUI" [5] "package:graphics" [7] "package:utils" [9] "package:methods" [11] "package:base"	Output "package:compiler" "package:stats" "package:grDevices" "package:datasets" "Autoloads"
	Input
#ls()	
ls()	3, y= 11:13) # just an example
	Output
[1] "expl"	Output
So far, x is hidden in exp	1.
try(x)	Input # component not in search path
try(expl\$x)	# component not in search path # component accessible using full name
	Output
[1] 1 2 3	
Now we attach expl to the	e search paths.
	· .
attach(expl)	Input
search()	# expl is added after .GlobalEnv
	Outroot
[1] ".GlobalEnv"	"expl" Output
[3] "package:compiler"	"tools:RGUI"
[5] "package:stats"	"package:graphics"
[7] "package:grDevices"	
[9] "package:datasets"	
[11] "Autoloads"	"package:base"

The global environment st	cill only contains expl.	
ls()	Input	
[1] "expl"	Output	
But now x is found by tra	eversing the search paths	
try(x)	Input # now in search path	
[1] 1 2 3	Output	
ls("expl") #	Input Inst objects in the environment att	ached
[1] "x" "y"	Output	
An here we clean up again	1.	
<pre>detach(expl) search()</pre>	Input	
[1] ".GlobalEnv" [3] "tools:RGUI" [5] "package:graphics" [7] "package:utils" [9] "package:methods" [11] "package:base"	"package:stats"	
ls()	Input	
[1] "expl"	Output	
trv(x)	Input # not in search path any more	

R COMPLEMENTS 9

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 optimisation. 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 recognises 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:

```
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 })
}

#called as
# f <- definef(actualx, actualy, ...)</pre>
```

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 [8].

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 <code>sys.calls()</code> and <code>sys.frame()</code>.

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, <code>sys.frame()</code> 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 $pi \leftarrow 4$ 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

2.2.4 Least Squares Estimation

A first idea of estimation in a linear model can be gained from the following relation: given X, we have $E(Y) = X\beta$. As X is a matrix, we cannot simply solve this relation for β using a division by X. But we can expand the relation to $X^{\top}E(Y) = X^{\top}X\beta$. $X^{\top}X$ is a positive semi-definite symmetric matrix. If X has rank p, the full rank, this matrix is invertible. In general at least a pseudo-inverse exists and we can calculate $(X^{\top}X)^{-}X^{\top}E(Y) = \beta$. This equation motivates the following estimator:

$$\widehat{\beta} = (X^{\top} X)^{-} X^{\top} Y. \tag{2.1}$$

Using the model relation $Y = X\beta + \varepsilon$ 2.2 and $E(\varepsilon) = 0$, in the full rank case we get

$$E(\widehat{\beta}) = E\left(\left(X^{\top}X\right)^{-}X^{\top}\left(X\beta + \varepsilon\right)\right) = \beta, \tag{2.2}$$

so $\widehat{\beta}$ is an unbiased estimator for β . It is a topic in statistics lectures to discuss whether there are other qualities of this estimator. The Gauss-Markov theorem is a theorem from statistics characterising the estimator 2.2. We will come back to this estimator frequently. We give it a name for reference: the **Gauss-Markov estimator**. In the case of a linear model, such as the regression model, this estimator has a series of optimality properties. For example, this estimator minimises the mean quadratic deviation, that is, it is a **least squares estimator** in this model.

The least squares estimator for linear models is implemented as function Im().

We generate an example data set to be used for illustration.

```
x <- 1:100
err <- rnorm(100, mean = 0, sd = 10)
y <- 2.5*x + err
```

For this example data set, we get the least squares estimator using

¹ Here X^{\top} means the transposed of the matrix X and $(X^{\top}X)^{-}$ denotes the (Penrose-Moore generalised) inverse of $(X^{\top}X)$.

Example 2.1: Least Squares Estimator			
lm(y ~ x)	Input		
Call: lm(formula = y ~ x)	Output		
Coefficients: (Intercept) x -0.5194 2.4885			

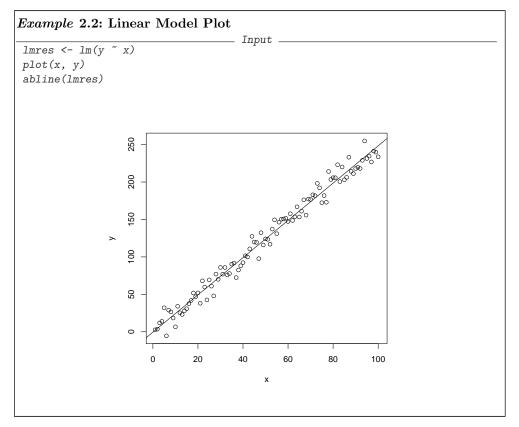
Exercise 2.1	
	When we generated the data, we did not use a constant term. The model specified for estimation, however, did not exclude the constant term. Repeat the estimation using the model without a constant term. Compare the results.

The estimator $\widehat{\beta}$ immediately yields an estimation \widehat{m} for the function m in our original model:

$$\widehat{m}(x) = x^{\top} \cdot \widehat{\beta}.$$

The evaluation at the measurement points results in the vector of the fitted values $\widehat{Y}=X\widehat{\beta}.$

In our example, the fit gives a regression line. Using <code>plot()</code> we can plot the data points. If we store the result of the regression, we can use it with <code>abline()</code> to add the regression line.



Function abline() is a function to draw lines, using various parametrisations. For more information, see help(abline).

Technically, we can apply the least squares estimation to any data set. The algorithm does not know whether the model assumptions apply, and it does not give us any information about the quality of the result. It is optimal, but optimality may not mean much if you are in dire straits. To judge the quality of the estimation, we need additional work.

The first step in this direction is to get information about the variance of the estimator. Equation 2.1 tells us that the estimator $\hat{\beta}$ is a linear function of the observations Y. The matrix X is assumed to be known, hence the linear function is considered a known function. So the stochastic variation comes from the error terms contained in Y, and we have to reconstruct this.

Equation (2.1) tells us how to calculate the fit at the measurement points:

$$\widehat{Y} = X(X^{\top}X)^{-}X^{\top} \cdot Y. \tag{2.3}$$

The matrix

$$H := X(X^{\top}X)^{-}X^{\top} \tag{2.4}$$

is called the *hat matrix*.² It is the main tool for analysing the Gauss-Markov estimator for a given design matrix X. The design matrix, and hence the hat matrix, depends only on the experimental conditions, not on the result of the experiment. The fit on the other side always refers to a specific outcome of the experiment, the random sample of observed values Y.

Writing Equation 2.3 as

$$\widehat{Y} = HY \tag{2.5}$$

highlights that the fit is a weighted average, a linear combination of the observations. Not all observations need to have the same weight. The coordinate representation

$$\widehat{Y}_i = H_{ii} \cdot Y_i + \sum_{j \neq i} H_{ij} \cdot Y_j \tag{2.6}$$

points to a potential problem. If all contributions H_{ii} are about equal, the fit is a balanced average and stochastic errors have a chance to balance out. Values of H_{ii} about Rk(X)/n are the best case. If some contributions H_{ii} are relatively large, the fit at data point i is dominated by these observations. The extreme would be one large value of H_{ii} , where the fit \hat{Y}_i would be dominated by Y_i . This is a sensitivity which by itself is not a problem, but it can lead to gross errors if there is some problem at data point i. The diagonal elements H_{ii} are called *leverages*.

For the simple linear regression Example 2.1 (page 61)

$$H_{ij} = \frac{1}{n} + \frac{(x_i - \overline{x})(x_j - \overline{x})}{\sum_{k=1}^{n} (x_k - \overline{x})^2}.$$

So H_{ii} is just the physical leverage you know from the physics of a seesaw with masses Y_i placed at position x_i .

The leverage does not depend the experimental outcome - it can be calculated based on the design matrix X only. High leverage is a feature of the experimental design, not of the observations. Design points with high leverage can contribute most information and are used in "optimal" designs, but on the other side they have the potential to be most misleading if the corresponding observation is a stray one. The leverages, or hat values, are used as diagnostics for this kind sensitivity by design.

The statistics of the experiment comes in by the stochastic error. The linear model contains a term ε , representing the measurement error or the experimental fluctuation. We cannot observe this error directly. If we could, we would subtract it and get exact information about the model function. But since the error is not observable, we have to resort to indirect inference.

We already introduced the notion of residuals. To repeat: in general, the value of the random observation Y is different from the fit \widehat{Y} . The difference

$$R_X(Y) := Y - \widehat{Y}$$

is called *residual*. The residual can be seen as an estimator for the non-observable error term ε .

² It puts the hat on top of $Y: \widehat{Y} = H \cdot Y$.

Residuals do not exactly match the error terms. This would only be the case if the estimation were exact. In our situation, the relation

$$R_X(Y) = Y - \widehat{Y}$$

$$= (I - H)Y$$

$$= (I - H)(X\beta + \varepsilon)$$

$$= (I - H)\varepsilon$$
(2.7)

shows that the residuals are linear combinations of the error terms. We have to infer back from these linear combinations to the error term.

If the variance of the error terms does exist, the variance matrix Σ of the error terms $Var\left(\varepsilon\right)=\Sigma$ determines the variance of the residuals:

$$Var(R_X(Y)) = Var((I - H)\varepsilon)$$

$$= (I - H)\Sigma(I - H)^{\top}.$$
(2.8)

So far we have only presumed that there is no systematic error. This was formalised as the assumption

$$E(\varepsilon) = 0.$$

We speak of a *simple linear model* if we have additionally:

$$\begin{split} &(\varepsilon_i)_{i=1,\dots,n} & \text{are independent} \\ &Var\left(\varepsilon_i\right) = \sigma^2 & \text{for a σ not depending on i}. \end{split}$$

For a linear model we try to estimate the parameter vector β . The variance structure of the vector of error terms introduces nuisance parameters, which complicate the estimation. For a simple linear model this nuisance reduces to just one unknown nuisance parameter σ . Equations like 2.8 can be simplified because now $\Sigma = \sigma^2 I$ and the parameter σ^2 can be pulled out from Formula 2.8. We can estimate this parameter from the residuals, because the **residual variance**

$$s^{2} := \frac{1}{n - Rk(X)} \sum_{i=1}^{n} (Y_{i} - \widehat{Y}_{i})^{2}$$
(2.9)

is an unbiased estimator for σ^2 , where Rk(X) is the rank of the matrix X. We write $\widehat{\sigma^2} := s^2$. (Taking the root is not a linear operation and does not preserve the expected value. The residual standard deviation $\sqrt{s^2}$ is not an unbiased estimator for σ .) Plugged into Equation (2.1), the residual variance estimator gives an estimator for the variance/covariance matrix of the estimator for β because in the simple model we have

$$Var\left(\widehat{\beta}\right) = \sigma^2(X^{\top}X)^{-},\tag{2.10}$$

which can be estimated by using the residual variance estimator as

$$\widehat{Var\left(\widehat{\beta}\right)} = s^2(X^{\top}X)^{-}. \tag{2.11}$$

If in addition we can assume that the errors have a normal distribution, s^2 and $\hat{\beta}$

are independent. We give a summary here, for simplicity for the full rank case where Rk(X) = p:

Theorem 2.1 For a simple linear model with independent Gaussian errors and full rank p = Rk(X) of the design matrix, the estimators $\widehat{\beta}$ and s^2 are stochastically independent:

$$\widehat{\beta} \sim N_p(\beta, \sigma^2 X^\top X)^{-1}) \tag{2.12}$$

and

$$(n-p)\frac{s^2}{\sigma^2} \sim \chi^1_{n-p}.$$
 (2.13)

If we standardise $\widehat{\beta}$ by $Var\left(\widehat{\beta}\right)$, each component has a t-distribution, that is, we can use t-tests for hypotheses such as $\beta_j=0$.

The standard output in Example 2.1 shows only minimal information about the estimator. More information about the estimator, residuals and derived statistics are returned if we ask for a summary.

```
Example 2.3: Linear Model Summary
                                  Input
summary(lm( y ~ x))
                                  Output
lm(formula = y
Residuals:
    Min
              1Q
                  Median
                                30
                                       Max
                           7.1822 21.4770
-19.8068 -6.2306 -0.4211
Coefficients:
          Estimate Std. Error t value Pr(>|t|)
(Intercept) -0.51945 1.99979
                                -0.26
                                        0.796
            2.48846
                       0.03438
                                72.38
                                        <2e-16 ***
Signif. codes: 0 ,Äò***,Äô 0.001 ,Äò**,Äô 0.01 ,Äò*,Äô 0.05 ,Äò.,Äô 0.1 ,Äò ,Äô 1
Residual standard error: 9.924 on 98 degrees of freedom
                                Adjusted R-squared: 0.9815
Multiple R-squared: 0.9816,
F-statistic: 5239 on 1 and 98 DF, p-value: < 2.2e-16
```

Exercise 2.2	
	Analyse the output of $lm()$ shown in Example 2.3. Which of the terms can you interpret? Write down your interpretations. For which terms do you need more information?
	Generate a commented version of the output.

In Section 2.3 (page 79) we will present the theoretical background needed to interpret the remaining terms.

A warning needs to be added here. R reports a t-test value and an error probability for each of the components of the parameter vector β . However, the estimation of the components and hence the derived t-tests are not independent. So to be on the safe side, you have to do a **Bonferroni correction**, that is, if β has p components and you want to guarantee an error level of α , make sure that the nominal levels for your decision are at most α/p . This is a crude bound to keep you on the safe side. In special cases, it

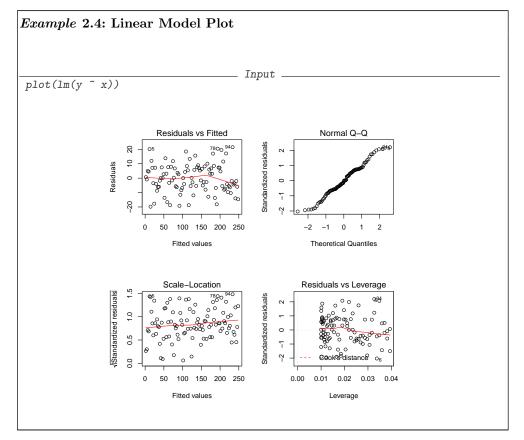
may be possible to have finer tools for simultaneous testing. Examples are in Section 2.4 (page 85).

2.2.5 Regression Diagnostics

Calling 1m() always returns a result if it is appropriate for the data, but it will also return a linear result if the linear model is not adequate. We need additional diagnostics to tell us whether the model is reliable and usable.

Exercise 2.3	
	Let
	yy <- 2.5*x +0.01 x^2 + err
	What are the results you get if you do a regression using the (incorrect) regression model $yy \sim x$? Do you get any hints that this model is not adequate?

The function 1m() not only gives an estimation for the linear model, but also provides a series of diagnostics that can help to judge whether the model assumptions are acceptable. A representation using plot() shows some aspects.



The top left plot shows the residuals against the fit. It gives a first survey.

The distribution of the fitted values depends on the design. Unless the design is homogeneous, you cannot expect the residual plot to be homogeneous.

The residuals should look approximately like a scatterplot of independent variables. The distribution of the residuals should not vary with the fit. If systematic structures show up in this plot, it is a warning that the model or the model assumptions may not be satisfied.

The previous discussion allows us to be more precise: the residuals should be linear combinations as in (2.7) of independent identically distributed variables. If the model assumptions are satisfied, the variance is given by (2.8).

In a one-dimensional situation, a plot of the residuals against the regressor would be sufficient. For p regressors, the graphical representation becomes difficult. The plot of the residuals against the fit, however, generalises to higher dimensions of the regressors.

Some caution is necessary. Even for independent identically distributed errors this plot is rarely a homogeneous plot. The variance of the residuals is in general not constant as seen from (2.8), and the visual spread will depend on the density of the fit values. But

it is good custom to start with a plot which is as near to the data as possible and leave adjustments for later steps.

If we start with distribution assumptions about the error terms, we can derive distribution properties of the estimator and of the residuals. The most powerful statements are possible if the error terms are independent identically distributed with a common normal distribution. In that case, the plot on the upper right should look approximately like a "normal probability plot" of normal random variates, where again "approximately" means: up to transformation with the matrix I - H. Using the empirical version of 2.8

$$Var(R_X(Y)) = Var((I - H)\varepsilon)$$

= $(I - H)\widehat{\Sigma}(I - H)^{\top}$ (2.14)

and inverting it would give *standardised residuals*. By convention, an approximation is taken, giving

$$R_i^{\text{(std)}} := \frac{R_i}{\sqrt{\widehat{\sigma^2}(1 - H_{ii})}}.$$
 (2.15)

This gives a unit variance and moves the residuals to a common scale. The dependence however is not removed. Standardised residuals in general are still dependent, even for independent errors.

After standardisation, residuals should be on a common scale. In particular their magnitude should not vary with the fit, to be inspected with the bottom-left plot.

The bottom-right plot is a scatterplot of the standardised residuals against leverage. Large standardised residuals are suspicious because they indicate a lack of fit. But small residuals may indicate a problem as well, in particular if they combine with a large leverage value. This hints at observations that may be outliers, possibly acting as leverage points with critical influence on the estimation. There is a rich literature on diagnostic plots which can be found using the keywords "residual analysis" or "regression diagnostics". As a concise textbook, see for example [22].

The plots included by default are a first step, and you will want to modify them or add your own selection. For example, if you are concerned about possible serial effects, you will add a plot of the (standardised) residuals against the case index, or add some indicators which are sensitive to loss of independence.

For diagnostic purposes, we go even further. We already have used a standardisation of the residuals in Equation (2.15) on page 20. If all assumptions are satisfied, this standardisation is sufficient. Standardisation transforms the residuals to a standard scale so that we have a notion of "large" and "small". Large standardised residuals indicate a poor fit and may indicate that a data point needs closer inspection.

If we have possible outliers that may work as leverage points and influence the regression critically, this influence on the estimation can lead to an over-fitting resulting in small residuals, effectively hiding the critical points. A large leverage value indicates a potential leverage influence, and large leverage values combined with small standardised residuals are particularly suspicious as this may hint to an effective leverage influence.

Least squares estimation, as used with linear models, is particular sensitive to leverage effects. From a decision theoretic point of view, this is using a square as a loss function. The loss is potentially unbounded, and a single far out point may destroy the estimation. Moreover, for square loss the slope is increasing. So far out points gain increasing influence.

As a first precaution, Equation (2.15) on page 20 is modified by *leave-one-out* deletion. At any data point, the regression is calculated and the variance is estimated on the data set, excluding that data point. This gives a variant of the residuals called *(externally) studentised residuals*. Standardised residuals are provided by *rstandard()*; externally studentised residuals are available as *rstudent()*.

Earlier versions of R used library MASS [21] which provides standardised residuals by stdres(); and externally studentised residuals as studres().

Coming back to the influence point of view, one can go beyond leverage diagnostics. The leverage gives the potential influence of a data point on the estimation. Taking partial derivatives of $\hat{\beta}$ or of \hat{Y} give an indication of the factual influence. The linear transformation which links the estimator $\hat{\beta}$ and the fit \hat{Y} may lead to different weights. Usually only leave-one-out versions of these diagnostics are considered, provided as dfbetas() and dffits(). Both are special cases of a general function influence.measures().

Leave-one-out diagnostics are but a first step in regression diagnostics. This approach is extensively discussed in [4].

Effects of several data points may interact. For example, a high and a low outlier may combine and be masked in leave-one-out diagnostics while still controlling the regression. Techniques for the analysis of multi-point effects are available. The computing effort however may soon become overwhelming. If there is but one outlier, there are just n candidates in n data points. If pairs are considered, there are $\binom{n}{2}$ possibilities, and complexity increases for more. With todays computing facilities, solutions are still feasible. For an example see the robust diagnostic regression analysis implemented in library forward [2][3].

Exercise 2.4	
	Use plot() to inspect the results of Exercise 2.3. Does it give you indications that the linear model is not appropriate? Which indications?

plot() provides additional diagnostic plots for linear models. These must be requested
explicitly using the parameter which.

help(lm)

lm

Fitting Linear Models

Description

1m is used to fit linear models. It can be used to carry out regression, single stratum analysis of variance and analysis of covariance (although aov may provide a more convenient interface for these).

Usage

```
lm(formula, data, subset, weights, na.action,
  method = "qr", model = TRUE, x = FALSE, y = FALSE, qr = TRUE,
  singular.ok = TRUE, contrasts = NULL, offset, ...)
```

Arguments

formula an object of class "formula" (or one that can be coerced to that

class): a symbolic description of the model to be fitted. The details

of model specification are given under 'Details'.

data an optional data frame, list or environment (or object coercible

by as.data.frame to a data frame) containing the variables in the model. If not found in data, the variables are taken from environment(formula), typically the environment from which lm

is called.

subset an optional vector specifying a subset of observations to be used

in the fitting process.

weights an optional vector of weights to be used in the fitting process.

Should be NULL or a numeric vector. If non-NULL, weighted least squares is used with weights weights (that is, minimizing sum(w*e^2));

otherwise ordinary least squares is used. See also 'Details',

na.action a function which indicates what should happen when the data con-

tain NAs. The default is set by the na.action setting of options, and is na.fail if that is unset. The 'factory-fresh' default is na.omit. Another possible value is NULL, no action. Value na.exclude can

be useful.

method the method to be used; for fitting, currently only method = "qr"

is supported; method = "model.frame" returns the model frame

(the same as with model = TRUE, see below).

model, x, y, qr

logicals. If TRUE the corresponding components of the fit (the model frame, the model matrix, the response, the QR decomposition) are returned.

singular.ok logical. If FALSE (the default in S but not in R) a singular fit is an

error.

contrasts an optional list. See the contrasts.arg of model.matrix.default.

offset this can be used to specify an a priori known component to be

included in the linear predictor during fitting. This should be NULL or a numeric vector of length equal to the number of cases. One or more offset terms can be included in the formula instead or as well, and if more than one are specified their sum is used. See

model.offset.

... additional arguments to be passed to the low level regression fitting

functions (see below).

Details

Models for 1m are specified symbolically. A typical model has the form response "terms where response is the (numeric) response vector and terms is a series of terms which specifies a linear predictor for response. A terms specification of the form first + second indicates all the terms in first together with all the terms in second with duplicates removed. A specification of the form first:second indicates the set of terms obtained by taking the interactions of all terms in first with all terms in second. The specification first*second indicates the cross of first and second. This is the same as first + second + first:second.

If the formula includes an offset, this is evaluated and subtracted from the response.

If response is a matrix a linear model is fitted separately by least-squares to each column of the matrix.

See model.matrix for some further details. The terms in the formula will be reordered so that main effects come first, followed by the interactions, all second-order, all third-order and so on: to avoid this pass a terms object as the formula (see aov and demo(glm.vr) for an example).

A formula has an implied intercept term. To remove this use either $y \sim x - 1$ or $y \sim 0 + x$. See formula for more details of allowed formulae.

Non-NULL weights can be used to indicate that different observations have different variances (with the values in weights being inversely proportional to the variances); or equivalently, when the elements of weights are positive integers w_i , that each response y_i is the mean of w_i unit-weight observations (including the case that there are w_i observations equal to y_i and the data have been summarized).

 ${\tt lm}$ calls the lower level functions ${\tt lm.fit}$, etc, see below, for the actual numerical computations. For programming only, you may consider doing likewise.

All of weights, subset and offset are evaluated in the same way as variables in formula, that is first in data and then in the environment of formula.

Value

lm returns an object of class "lm" or for multiple responses of class c("mlm",
"lm").

The functions summary and anova are used to obtain and print a summary and analysis of variance table of the results. The generic accessor functions coefficients, effects, fitted.values and residuals extract various useful features of the value returned by lm.

An object of class "lm" is a list containing at least the following components:

coefficients a named vector of coefficients

residuals the residuals, that is response minus fitted values.

fitted.values the fitted mean values.

rank the numeric rank of the fitted linear model.
weights (only for weighted fits) the specified weights.

df.residual the residual degrees of freedom.

the matched call.

terms the terms object used.

contrasts (only where relevant) the contrasts used.

xlevels (only where relevant) a record of the levels of the factors used in

fitting.

offset the offset used (missing if none were used).

y if requested, the response used.
x if requested, the model matrix used.

model if requested (the default), the model frame used.

na.action (where relevant) information returned by model.frame on the spe-

cial handling of NAs.

In addition, non-null fits will have components assign, effects and (unless not requested) qr relating to the linear fit, for use by extractor functions such as summary and effects.

Using time series

Considerable care is needed when using lm with time series.

Unless na.action = NULL, the time series attributes are stripped from the variables before the regression is done. (This is necessary as omitting NAs would invalidate the time series attributes, and if NAs are omitted in the middle of the series the result would no longer be a regular time series.)

Even if the time series attributes are retained, they are not used to line up series, so that the time shift of a lagged or differenced regressor would be ignored. It is good practice to prepare a data argument by ts.intersect(..., dframe = TRUE), then apply a suitable na.action to that data frame and call lm with na.action = NULL so that residuals and fitted values are time series.

Note

Offsets specified by offset will not be included in predictions by predict.lm, whereas those specified by an offset term in the formula will be.

Author(s)

The design was inspired by the S function of the same name described in Chambers (1992). The implementation of model formula by Ross Ihaka was based on Wilkinson & Rogers (1973).

References

Chambers, J. M. (1992) Linear models. Chapter 4 of Statistical Models in S eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

Wilkinson, G. N. and Rogers, C. E. (1973) Symbolic descriptions of factorial models for analysis of variance. *Applied Statistics*, **22**, **392–9**.

See Also

summary.1m for summaries and anova.1m for the ANOVA table; aov for a different interface.

The generic functions coef, effects, residuals, fitted, vcov.

predict.lm (via predict) for prediction, including confidence and prediction intervals; confint for confidence intervals of *parameters*.

lm.influence for regression diagnostics, and glm for generalized linear models. The underlying low level functions, lm.fit for plain, and lm.wfit for weighted regression fitting.

More lm() examples are available e.g., in anscombe, attitude, freeny, LifeCycleSavings, longley, stackloss, swiss.

biglm in package biglm for an alternative way to fit linear models to large datasets (especially those with many cases).

Examples

```
require(graphics)
```

```
## Annette Dobson (1990) "An Introduction to Generalized Linear Models".
## Page 9: Plant Weight Data.
ctl <- c(4.17,5.58,5.18,6.11,4.50,4.61,5.17,4.53,5.33,5.14)
trt <- c(4.81,4.17,4.41,3.59,5.87,3.83,6.03,4.89,4.32,4.69)
group <- gl(2,10,20, labels=c("Ctl","Trt"))
weight <- c(ctl, trt)
lm.D9 <- lm(weight ~ group)
lm.D90 <- lm(weight ~ group - 1) # omitting intercept
anova(lm.D9)
summary(lm.D90)</pre>
```

```
opar <- par(mfrow = c(2,2), oma = c(0, 0, 1.1, 0))
plot(lm.D9, las = 1)  # Residuals, Fitted, ...
par(opar)
### less simple examples in "See Also" above</pre>
```

To be added to the help information: in the formula notation, with two terms or lists of terms first and second, first-second includes the variables indicated by the first term, but excludes those indicated by the second. For more information on the formula notation, see help(formula). A summary is given in Appendix A.18 (page Suppl.A-70).

The hat matrix is a particularity of linear models. Fit and residuals, however, are general concepts and can be applied for all kind of estimations. Clients are often satisfied seeing a fit (or the estimation). For serious clients, and for statisticians, the residuals often contain more valuable information. They indicate what is not yet covered by the model or the estimation.

2.2.6 Gauss-Markov Estimator

Let us take a closer look at the Gauss-Markov estimator. Knowledge from linear algebra, considerable thought or other sources tell us:

Remark 2.2

- (1) The design matrix X defines a mapping $\mathbb{R}^p \to \mathbb{R}^n$ with $\beta \mapsto X\beta$. Let \mathcal{M}_X , $\mathcal{M}_X \subset \mathbb{R}^n$ be the image space of this mapping. \mathcal{M}_X is the vector space generated by the column vectors from X.
- (2) If the model assumptions are satisfied, $E(Y) \in \mathcal{M}_X$.
- (3) $\widehat{Y} = \pi_{\mathscr{M}_X}(Y)$, where $\pi_{\mathscr{M}_X} : \mathbb{R}^n \to \mathscr{M}_X$ is the (Euclidean) orthogonal projection.
- (4) In the full rank case, $\hat{\beta} = \arg\min_{\beta} |Y \hat{Y}_{\beta}|^2$ where $\hat{Y}_{\beta} = X\beta$.

The characterisation (3) of the Gauss-Markov estimator as an orthogonal projection often helps understanding. The fit is the orthogonal projection of the observation vector on the space of expected values of the model (which hence minimises the quadratic distance). This is the space spanned by the columns of the design matrix. The vector of residuals is the orthogonal complement.

In statistics, the estimator is analysed systematically, and the characterisation given above is just one starting point. Some properties of the estimator can be easily derived using knowledge from probability theory, such as the following lemma:

Theorem 2.3 Let Z be a random variable with values in \mathbb{R}^n , with $N(0, \sigma^2 I_{n \times n})$ distribution, and let $\mathbb{R}^n = L_0 \oplus \ldots \oplus L_r$ be an orthogonal decomposition. Let $\pi_i = \pi L_i$ be the orthogonal projection onto L_i , $i = 0, \ldots, r$. Then the following holds:

REGRESSION 27

(i) $\pi_0(Z), \ldots, \pi_r(Z)$ are independent random variables with normal distributions.

(ii)
$$\frac{|\pi_i(Z)|^2}{\sigma^2} \sim \chi^2(\dim L_i)$$
 for $i = 0, \dots, r$.

Proof. \rightarrow probability theory. See, for example, [11], 2.5 Theorem 3.

Using $\varepsilon = Y - X\beta$ allows us to derive the theoretical distributions for the estimator $\widehat{\beta}$ and the residuals $Y - \widehat{Y}$.

In particular, for simple linear models, the residual variance can be used to calculate the variance (resp. standard deviation) for each component $\widehat{\beta}_k$. The corresponding t statistics and the p-value for the test of the hypothesis $\widehat{\beta}_k = 0$ are given in the output of summary().

Exercise 2.5	
	What is the distribution of $ R_X(Y) ^2 = Y - \widehat{Y} ^2$, if ε has a $N(0, \sigma^2 I)$ distribution?

At first glance $|R_X(Y)|^2 = |Y - \widehat{Y}|^2$ seems an appropriate gauge to judge the quality of a model: small values indicate a good fit, large values indicate a poor fit. However, this has to be taken with caution. On the one hand, this value depends on linear scale factors. On the other hand, the dimension of the spaces involved has to be taken into account.

What happens if additional regressors are taken into the model? We have already seen that "linear" includes the possibility of modelling non-linear relations, for example, by taking transformed variables into the design matrix. The characterisation (3) in Remark 2.2 tells us that effectively only the vector space spanned by the design matrix is relevant. Here we can see limits for the Gauss-Markov estimator in linear models: if many transformed variables are taken into the model, or generally if the image space determined by the design matrix becomes too large, an over-fitting will result. In the extreme case we may get $\widehat{Y} = Y$. So all residuals are zero, but the estimation is not useful.

We use $|R_X(Y)|^2/\dim(L_X)$, where L_X is the orthogonal complement of \mathcal{M}_X in \mathbb{R}^n (so $\dim(L_X) = n - \dim(\mathcal{M}_X)$) to compensate for the number of dimensions.

Exercise 2.6	
	Modify the output of plot.lm() for the linear model so that instead of the Tukey-Anscombe plot the studentised residuals are plotted against the fit.
	$(\text{cont.}) \rightarrow$

28 REGRESSION

Exercise 2.6	(cont.)
*	Enhance the QQ -Plot by Monte Carlo bands for independent nor-
	mal errors.
	Hint: You cannot generate the bands directly from a normal dis-
	tribution — you need the distribution of the residuals, not the
	distribution of the errors.

Exercise 2.7	
	Write a procedure that calculates the Gauss-Markov estimator for the simple linear regression
	$y_i = a + bx_i + \varepsilon_i$ with $x_i \in \mathbb{R}, a, b \in \mathbb{R}$ and shows four plots:
	• response against regressor, with estimated straight line
	• studentised residuals against fit
	ullet distribution function of the studentised residuals in a QQ plot with confidence bands
	• histogram of the studentised residuals

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

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

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

R COMPLEMENTS 29

linear function. This results in a generalised linear model

$$Y=m(X)+arepsilon$$
 Y with values in \mathbb{R}^n
$$X\in\mathbb{R}^{n\times p}$$

$$E(arepsilon)=0$$

$$m(X)=\overline{m}(\eta) \text{ with } \eta=X\beta,\beta\in\mathbb{R}^p.$$

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 generalised linear models is [7], and an extensive classical survey is [13].

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

Polymorphism 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 inherit the structure of their 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. <code>setClass()</code> allows to define the structure of a class. <code>new()</code> is available to create an instance of a class. For details, see chapter 5 of [20].



CHAPTER 3

Comparisons

Exercise 3.1	Click Timing
	Define a function $click(runs)$ that repeats $click1()$ a chosen number $runs$ plus one times and returns the result as a $data.frame$. The additional first timing should be considered as a "warming up" and is not included in the following evaluations.
	Select a number runs. Give reasons for your choice of runs. Execute click(runs) and store the result in a file using write.table().
	Display the distribution of the component tclick with the methods from Chapter 1 (distribution function, histogram, box-and-whisker plot).

3.1 Shift/Scale Families, and Stochastic Order

Exercise 3.1	Click Comparison
	Perform Exercise 3.1 using the right hand and then again using the left hand. Compare the empirical distributions of the timing data returned by tclick() for the right and left hand.
	The recorded data also contain information about the positions. Define a distance measure <code>dist</code> for the deviation. Give reasons for your definition. Perform a right/left comparison for <code>dist</code> .
	For later analysis, store the results for the right hand and for the left hand in files. named "clickright-xxxx" and " "clickright-xxxx", where xxxx is an identification of you choice. For example, use your initials, the date and some sequential number, such as in "clickright-cs20050416-1".

32 COMPARISONS

3.3 Tests for Shift Alternatives

. . .

To apply the Wilcoxon test, on the one hand the test statistic has to be calculated. On the other hand, to determine the critical values, the distribution function has to be evaluated. If all observations are distinct, this function depends only on n_1 and n_2 , and fairly simple algorithms are available. These are provided in the R base and used by wilcox.test(). However, if there are ties in the data, that is, there are values occurring more than once, the distribution depends on the special pattern of these ties and the calculation is laborious. wilcox.test() returns to approximations in this case. For an exact evaluation (in contrast to approximative), the necessary algorithms are available as well. To use them, you need library(coin). The exact variant of the Wilcoxon tests, for example, is implemented as one option in $wilcox_test()$. Besides providing the exact Wilcoxon test, this function allows to calculate approximative Monte Carlo solutions and solutions based on asymptotic approximations.

For the exact test, you have to combine the data, for example code as in

```
clicktimes <- stack(list(left=left$tclick, right= right$tclick))
names(clicktimes) <- c("time", "side")
wilcox_test(time~side, data=clicktimes, distribution="exact")</pre>
```

Exercise 3.1	
	Use the Wilcoxon test to compare the results of the right/left click experiment.
	Use both variants, the approximative test wilcox.test() and the exact Wilcoxon test wilcox_test().

CHAPTER 4

Dimensions 1, 2, 3, ..., ∞

4.1 R Complements

In this chapter we begin with complements on R in order to concentrate on statistical questions for the remainder without disrupting the discussion with programming details. We take a look at the graphical possibilities that are at our disposal.

The basic graphics model of R is oriented to possibilities historically provided by a plotter as an output device. The graphics follow the possibilities available when drawing with a pen. Besides the one- and two-dimensional possibilities which we have seen so far, there are possibilities to display a real valued function that is defined over a grid. Basically, three R functions are available for this.

3d Basic Graphics	
image()	gives the values of a variable z in grey levels or colour coding.
contour()	gives the contours of a variable z .
filled.contour()	gives the contours of a variable z , with the areas between the contours filled in solid colour
persp()	gives a perspective plot of a variable z .

The basic graphic system does not provide a generalisation of plot() for three dimensions. The classic solution is to use persp() to set up a 3d coordinate system. The translation matrix for this coordinate system to 2d is returned as a hidden result an can be used to project 3d points to 2d in this perspective using trans3d(). points() is then used to add the points. See the examples section in help(persp).

The basic graphics system in R is easy to use, and it is available in all R implementations. But it is limited in possibilities. A newer graphics system, the grid and lattice graphics [19], conceptually works with objects and a viewport model. The graphic objects can be combined and post-processed. The display takes part in a separate step. Simple 2d graphs can be post-processed. For a 3d display, distance, the point of view and the focal length can be chosen as we would do when using a camera. The object-oriented

graphics system consists of a library <code>grid</code> with the elementary operations required, and a higher level library <code>lattice</code> that gives new implementation of the displays known from the basic graphics and adds additional displays. While the basic graphic system has the plotter as underlying technical device, you should think of a postscript printer as a technical model for <code>grid/lattice</code>.

image() and contour() can also be used to give an overlay on other plots.

Example 4.1: 3d Surface Displays Using Base Graphics Input -# 10 meter spacing (S to N) x <- 10*(1:nrow(volcano))</pre> y <- 10*(1:ncol(volcano))</pre> # 10 meter spacing (E to W) image(x, y, volcano, col = terrain.colors(100), axes = FALSE, xlab = "Meters North", ylab = "Meters West") axis(1, at = seq(100, 800, by = 100))axis(2, at = seq(100, 600, by = 100))title(main = "Maunga Whau Volcano", font.main = 4) contour(x, y, volcano, levels = seq(90, 200, by = 5), col = "peru", main = "Maunga Whau Volcano", font.main = 4, xlab = "Meters North", ylab = "Meters West") $z \leftarrow 2 * volcano$ # Exaggerate the relief ## Don't draw the grid lines : border = NA persp(x, y, z, theta = 135, phi = 30, col = "green3", scale = FALSE, ltheta = -120, shade = 0.75, border = NA, box = FALSE) 8 200 image() contour() persp() See Colour Figure ??.

3d Lattice Graphics	
cloud()	generic lattice function to draw 3d scatterplots.

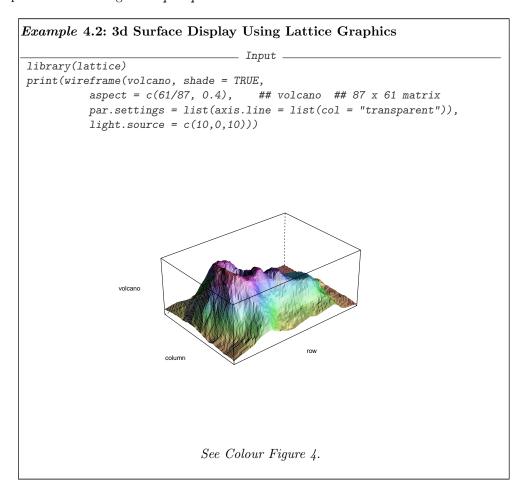
 $(cont.) \rightarrow$

R COMPLEMENTS 35

3d Lattice Graphics	
(cont.)	
wireframe()	generic lattice function to draw 3d surfaces.

In the basic graphics system, the functions usually provide a graphical output, and the internal information must be accessed explicitly. In the lattice system, the functions usually return lattice objects. Graphical output must be requested explicitly. For the output of lattice objects the function print() is used.

As far as parametrising the graphics system is concerned, grid/lattice encourages a different programming style than base graphics. In base graphics, you would use par() to define the graphics set up, or pass individual parameters via high level function to par(). With grid/lattice, the preferred way is to collect the parameters as a list and pass this list as argument par.parameters.



The basic graphics system and lattice graphics are separate graphics systems. Unfortunately, they use different notations for comparable functions, and comparable displays have different representations. A small translation aid is given in Table 4.5. Some convenience functions to combine both graphics systems are provided in library gridBase. An extensive introduction to both graphics systems is [14].

Basic Graphics		Lattice
barplot()	bar chart	barchart()
boxplot()	box-and-whisker plot	bwplot()
	three-dimensional scatterplot	cloud()
contour	contour plot	contourplot()
coplot	conditional scatterplots	xyplot()
<pre>plot(density())</pre>	density estimator	densityplot()
dotchart()	dot plot	dotplot()
hist()	histogram	histogram()
image()	colour map plots	splom()
	parallel coordinate plots	parallel()
pairs()	scatterplot matrices	wireframe()
persp()	three-dimensional surface	wireframe()
plot()	scatterplot	xyplot()
qqnorm()	theoretical QQ plot	qqmath()
qqplot()	empirical QQ plot	qq()
stripchart()	one-dimensional scatterplot	stripplot()

Table 4.5 Basic graphics and lattice graphics

If you know that you are displaying 3d scenes, you might consider <code>library(rgl)</code> [1] as an alternative. If implemented on your system, <code>rgl</code> provides real-time 3d rendering with interactive facilities. This code snippet will allow you to turn the vulcano upside down:

```
library("rgl")
example(surface3d)
```

In a wide range of scientific visualisations, OpenGL is used as a common standard. OpenGL functions are accessible in R using the library rg1. There are, however, certain differences between common requirements for graphics, and the specific requirements of statistical graphics. As far as the representation of functions is concerned, statistical graphics is comparable with the requirements usual in analysis. The small difference is that functions in statistics are often piece-wise constant or only piece-wise continuous,

R COMPLEMENTS 37

while, for example, in analysis continuous or even differentiable functions are the rule rather than the exception. When it comes to displaying data, the situation changes drastically. Usually, statistical data are discrete. Smoothness properties that simplify display of analytical data are not available for statistical data. So visualisations adapted to the needs of statistics are required.

Library misc3d [6] provides a more convenient plotting procedure like contour3d() and image3d() which can be used either with the basic graphic system, or grid/lattice, or rgl.



R as a Programming Language and Environment

 ${\sf R}$ is an interpreted expression language. Expressions are composed of objects and operators.

A.1 Help and Information

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

R <i>Help</i>		
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.	

 $(\text{cont.}) \rightarrow$

Suppl.A-40 R AS A PROGRAMMING LANGUAGE AND ENVIRONMENT

R <i>Help</i>	
(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.
	<pre>vignette(all = TRUE) lists vignettes from all installed pack- ages.</pre>
	Example: vignette("grid") shows a vignette for the grid graphics.

See also Appendix A.6 "Object Inspection" (page Suppl.A-47) and Appendix A.7 "System Inspection" (page Suppl.A-48).

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 <code>search()</code>.

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

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()
	4

 $(cont.) \rightarrow$

Suppl.A-42 R AS A PROGRAMMING LANGUAGE AND ENVIRONMENT

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(\langle object list \rangle)

R offers a series of possibilities to configure the system so that certain commands are executed upon start or termination. When starting, the files <code>.Rprofile</code> and <code>.RData</code> 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-79).
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-48).

Suppl.A-44 $\,$ $\,$ R AS A PROGRAMMING LANGUAGE AND ENVIRONMENT

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. $\langle \texttt{type} \rangle$ () tests for a type, as. $\langle \texttt{type} \rangle$ () converts to a type.

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

$Special \\ stants$	Con-	
TRUE		alternative: T. Type: logical.
FALSE		alternative: F. Type: logical.

 $(cont.) \rightarrow$

$Special \ stants$	Con-	
(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.

Suppl.A-46 R AS A PROGRAMMING LANGUAGE AND ENVIRONMENT

A.5 Output for Objects

Revised from Appendix A.5 (page Suppl.A-46): 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	
<pre>print()</pre>	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) [5] is available.

Output of objects to files is discussed in Appendix A.15 "Input and Output to Data Streams" (page Suppl.A-64).

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(\langle object \rangle)
structure()	shows the internal structure of an object. Attributes for the display can be passed as parameters.
	Example: structure(1:6, dim = 2:3)
	$Syntax:$ $structure(\langle object \rangle,)$
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($\langle obj \rangle$) gives the names attribute of $\langle obj \rangle$. names($\langle obj \rangle$)<- $\langle charvec \rangle$ sets the names attribute.
	Example: $x < -values$ $names(x) < -\langle charvec \rangle$

Suppl.A-48 R AS A PROGRAMMING LANGUAGE AND ENVIRONMENT

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:	
	$Syntax: \begin{array}{ll} \textit{methods}(\langle \texttt{fun} \rangle) \\ & \text{shows specialised functions for } \langle \texttt{fun} \rangle, \\ & \textit{methods}(\textit{class} = \langle \texttt{c} \rangle) \text{the class-specific functions for } \\ & \text{class} \ \langle \texttt{c} \rangle. \end{array}$	
	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 2.6.5 (page 104)).

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-64).

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 dat	a types.
Matrices	latrices vectors with two-dimensional layout.	
	See also	Appendix A.10 "Data Manipulations" (page Suppl.A-54).
Arrays vectors with higher-dimensional layout.		h 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-54).
Factors	special case	e for categorical data.
	factor()	converts a numeric vector into a factor.
	See also	Section 2.2.1.
	ordered()	converts a vector into a factor with ordered levels. This is a shortcut for $factor(x,, ordered = TRUE)$.

 $(\text{cont.}) \rightarrow$

Suppl.A-50 R AS A PROGRAMMING LANGUAGE AND ENVIRONMENT

Compound Data Types (cont.)		
	levels()	returns the levels of a factor.
	Example:	<pre>x <- c("a", "b", "a", "c", "a") xf <- factor(x) levels(xf) results in [1] "a" "b" "c"</pre>
	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:	$list(\langle {\tt components} \rangle)$
	[[]]	access to components of a list by index.
	\langle list\$compo	onent) access by names.
	Example:	<pre>1 <- list(name = "xyz", age = 22, fak = "math") > 1[[2]] 22 >1\$age 22</pre>
Data Frames		nes analogous to arrays resp. lists, with column-wise pe and uniform column length.
	data.fram	e()
		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.

$Accessing \ Components$	
$x[\langle indices \rangle]$	indicated components of x . Example: $x[1:3]$
$x[-\langle indices \rangle]$	x omitting indicated components. Example: $x[-3]$ x omitting the 3. component.
$x[\langle condition \rangle]$	components of x , for which the $\langle condition \rangle$ 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, by-row = FALSE, dimnames = NULL)	
	See also Example 1.11 (page 23)	

 $(cont.) \rightarrow$

Suppl.A-52 R AS A PROGRAMMING LANGUAGE AND ENVIRONMENT

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

 $NCOL\left(\right)$ and $NROW\left(\right)$ are variants treating a vector as a one-column resp. as a one-row matrix.

R Iterators			
apply()	applies a f	unction 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 f	unction 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:	<pre>sapply(X, FUN,, simplify = TRUE, USE.NAMES = TRUE)</pre>	
mapply()	applies a function to multiple list or vector arguments.		
	Syntax:	<pre>mapply(FUN,, MoreArgs = NULL, simplify = TRUE, USE.NAMES = TRUE)</pre>	
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:	<pre>Vectorize(FUN, vectorize.args = arg.names, simplify = TRUE, USE.NAMES = TRUE)</pre>	
tapply()	applies a function to components of an object depending on a list of controlling factors.		
by()	object-orie	ented variant of tapply.	
	Syntax:	by(data, INDICES, FUN,)	
aggregate()	calculates	statistics for subsets.	
	Syntax:	aggregate(x,)	

 $(\mathrm{cont.}) {\rightarrow}$

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,)

Suppl.A-54 R AS A PROGRAMMING LANGUAGE AND ENVIRONMENT ${\bf A.10~Data~Manipulation}$

combines by columns.	
combines by rows.	
splits a vector or matrix into the groups defined by a factor. Syntax: split(x, f, drop = FALSE,)	
combines components to a vector or matrix, i.e., reverses $split()$.	
generates a table of counts.	
expresses table entries as fraction of marginal table, i.e., gives relative counts.	
transposes rows and columns. Syntax: t(x)	
Transpose an array, with subscripts permuted as indicated by perm. 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.

 $(\mathrm{cont.}) {\rightarrow}$

Character String Transformations (cont.)			
,			
toupper()	translates to uppercase.		
chartr()	translates characters in a character vector.		
substr()	extracts or replaces substrings in a character vector.		
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.	

$egin{array}{c} Vector \\ Manipulation \end{array}$			
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,)		

 $(\mathrm{cont.}) {\rightarrow}$

Suppl.A-56 R AS A PROGRAMMING LANGUAGE AND ENVIRONMENT

Vector Manipulation (cont.)		
unstack()	splits a vector by an indicator variable, i.e., reverses the operation of stack().	
	Syntax: unstack(x,)	
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: $cut(x,)$	

OPERATORS Suppl.A-57

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 $\%\langle \mathtt{name}\rangle \%$ 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 $\{\ldots\}$.

Suppl.A-58 $\,$ R AS A PROGRAMMING LANGUAGE AND ENVIRONMENT

A.12 Functions

Functions are special objects. Functions can return objects as results. $\,$

R Function				
Declarations				
Declarations				
	Example: fak <- function(n) prod(1:n)			
Formal argument	\langle argument name \rangle = \langle default value \rangle			
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.			
	$\langle \mathtt{value} \rangle$ 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: \langle Variable \rangle << - \langle Value \rangle$			

R Function Call	
Function call	$\langle name \rangle (\langle Supplied (actual) argument list \rangle)$ $Example: fak(3)$
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: \langle \text{list of values} \langle \text{\argument name} = \langle \text{values} \rangle Example: \text{rnorm(10, sd = 2)}

FUNCTIONS Suppl.A-59

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 <code>environment()</code> allows direct access to variables. Techniques to use this are described in [8].

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

In ${\sf R}$ assignment functions the value argument ${\sf must}$ be called "value".

Suppl.A-60 R AS A PROGRAMMING LANGUAGE AND ENVIRONMENT

A.13 Debugging and Profiling

Revised from Appendix A.13 (page A-215): changes in "Profiling Support".

R provides a collection of tools for identification of errors. These are particularly helpful in connection with functions. <code>browser()</code> 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 <code>debug()</code>, the browser mode is activated automatically upon entry to a function. The browser mode is marked by a special prompt <code>Browse[xx]></code>.

⟨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 function is directed to call browser() automatically in case of an error.		
debug()	marks a function for debugger control. On subsequent calls to th function, the debugger is activated and switches to browser mode.		
	$Syntax:$ debug($\langle function \rangle$)		
undebug()	cancels debugger control for a function.		
	$Syntax:$ undebug($\langle function \rangle$)		
marks a function for trace control. On subsequent calls to the the call is signalled together with its arguments.			
	$Syntax: trace(\langle function \rangle)$		
untrace()	cancels trace control for a function.		
	$Syntax:$ $untrace(\langle function \rangle)$		
	(

 $(\text{cont.}) \rightarrow$

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.	
	$Syntax:$ $traceback(\langle expression \rangle)$	

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 <code>capabilities()</code>. See Appendix A.7 "System Inspection" (page Suppl.A-48).

Profiling Support		
system.time()		execution time of an expression. This function is available ementations.
	Syntax:	$system.time(\langle expr angle, \langle gcFirst angle)$
R has been compiled for "pro With memory.profiling = 1		ry.profiling = TRUE, in addition to the timing the mems recorded periodically. This option is only available if R
	Syntax:	<pre>Rprof(filename = "Rprof.out", append = FALSE, interval = 0.02, memory.profiling = FALSE)</pre>
		Use Rprof(NULL) to switch off profiling.
Rprofmem()		mory requirements on demand. on is only available if R has been compiled for "memory
	Syntax:	<pre>Rprofmem(filename = "Rprofmem.out", append = FALSE, threshold = 0)</pre>
		Use Rprofmem(NULL) to switch off profiling.
summaryRprof()	summarises the output of Rprof() and reports the timing by function.	
	Syntax:	<pre>summaryRprof(filename = "Rprof.out", chunksize = 5000, memory = c("none", "both", "tseries", "stats"), index = 2, diff = TRUE, exclude = NULL)</pre>
		As an alternative, the Perl script R CMD Rprof can be used. See R CMD Rprof -help for usage information.

Suppl.A-62 R AS A PROGRAMMING LANGUAGE AND ENVIRONMENT

A.14 Control Structures

R Control Structures				
if	conditional	conditional execution.		
	Syntax:	if ($\langle \log. expression1 \rangle$) $\langle expression2 \rangle$ The logical expression1 may return only one logical value. For vector-oriented access use ifelse.		
	Syntax:	$\begin{array}{ll} \mbox{if} & (\langle \log.~ \mbox{expression1} \rangle) & \langle \mbox{expression2} \rangle & \mbox{else} \\ \langle \mbox{expression3} \rangle & \end{array}$		
ifelse	element wis	se conditional execution.		
	Syntax:	<pre>ifelse(\langle log. expression1\rangle, \langle expression2\rangle, \langle expression3\rangle) evaluates the logical expression1 element wise on a vec- tor, and returns expression2 if the evaluation gives true, else expression3.</pre>		
	Example:	trimmedX <- ifelse (abs(X)<2, x, $sign(X)*2$)		
switch	evaluates a sult.	n expression and executes an instruction based on the re-		
	Syntax:	$switch(\langle expression1 \rangle, \ldots)$ expression1 must return a numeric value or a character string is an explicit list of alternative actions.		
	Example:	<pre>centre <- function (x , type) { switch(type, mean = mean(x), median = median(x), trimmed = mean(x, trim = .1)}</pre>		
for iteration (loop).		oop).		
	Syntax:	for $(\langle \mathtt{name} \rangle \ \texttt{in} \ \langle \mathtt{expression1} \rangle) \ \langle \mathtt{expression2} \rangle$		
repeat	iteration. N	iteration. Must be terminated explicitly, for example with break.		
	Syntax:	$ extit{repeat } \langle exttt{expression} angle$		
	Example:	<pre>pars<-init repeat { res<- get.resid (data, pars) if (converged(res)) break pars<-new.fit (data, pars)}</pre>		
		$({\rm cont.}) {\rightarrow}$		

R Control Structures (cont.)		
while	conditional repetitions.	
	$Syntax:$ while ($\langle log. expression \rangle$) $\langle expression \rangle$	
	<pre>Example: pars<-init; res <- get.resid (data, pars) while (!converged(res)) { pars<- new.fit(data, pars) res<- get.resid}</pre>	
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 [12]).

Suppl.A-64 R AS A PROGRAMMING LANGUAGE AND ENVIRONMENT

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("\(file name \)", \(\ldots \)	
sink()	redirects output in the file specified.	
	$Syntax:$ $sink("\langle file name \rangle")$	
	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().	
	$Syntax: \qquad \texttt{dump(list, file = "}\langle \texttt{dumpdata.R}\rangle", append = FALSE)$	

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 <code>help(connections)</code>.

To edit or enter data, R provides <code>edit()</code>. This is a polymorphic function For the special case of matrix-like data, <code>data.entry()</code> 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 <code>library(foreign)</code> for Stata, SAS, Minitab and SPSS, <code>library(RODBC)</code> for SQL. For more information, see the manual "Data Import/Export" [15].

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-67).

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/Out- put for R		
save()	stores data in an external file.	
	$Syntax: \qquad \text{save}(\langle \texttt{names of the objects to be stored} \rangle, \ \textit{file = } \\ \langle \texttt{file name} \rangle, \ \ldots)$	
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 = $\langle file name \rangle$,)	
data()	loads data. data() can handle various file formats, if the access paths and filenames follow the R conventions.	
	Syntax: data(, list = character(0),	
	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:

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

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

Input and Output of Data for Exchange		
read.table()	reads data t	ables. read.table(file, header = FALSE, sep = "\t",)
	Examples:	<pre>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,</pre>

 $(cont.) \rightarrow$

Input and Output of Data for Exchange (cont.)		
write.table()	writes data table.	
	Syntax: write.table(file, header = FALSE, sep = $'\t'$,)	
	Examples: write.table(\langle data frame \rangle, \langle file name \rangle, header = TRUE, sep = '\t') headers in row 1, row labels in column 1 write.table(\langle data frame \rangle, \langle file name \rangle, 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 [9]:

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

Package Utilities		
install.packages()	installs add-on package in $\langle \mathtt{lib} \rangle$, downloading it from the archive CRAN or from specified archives.	
	Syntax:	<pre>install.packages(pkgs, lib, CRAN = getOp- tion("CRAN"),)</pre>
	Example:	<pre>install.packages("mypackage.tgz", repos=NULL installs package from a local file.</pre>
library()	loads an inst	talled add-on package into the current workspace.
	Syntax:	library(package,)
	See also	Section 1.5.2 "Packages" (page 54).
require()	tries to load	an add-on package; gives warning on error.
	Syntax:	require(package,)
detach()	releases an a	add-on package and removes it from the search path.
	Syntax:	$detach(\langle name \rangle)$
package.manager()	if implement	ed, interface for management of installed packages.
	Syntax:	<pre>package.manager()</pre>
package.skeleton()	creates a skeleton for a new package.	
	Syntax:	$\label{eq:package} $
citation()	gives bibliog	raphic information for citing a package.
	Syntax:	${\it citation}(\langle {\tt package name} \rangle, \; {\it lib.loc} \; = {\it NULL})$

Suppl.A-68 R AS A PROGRAMMING LANGUAGE AND ENVIRONMENT

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 ${\sf R}$ packages is in "Writing ${\sf R}$ Extensions" ([17]).

A.17 Mathematical Functions; Linear Algebra

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

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 <code>smooth</code> to find more fitting methods.

Suppl.A-70 R AS A PROGRAMMING LANGUAGE AND ENVIRONMENT

A.18 Model Descriptions and Diagnostics

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

$$\mathbf{Y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon},$$

where the matrix ${\tt X}$ 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
~	$Y \sim M$	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^n	M with all "interactions" up to level n	
<i>I</i> ()	I(M)	interpret M ; terms in M retain their original meaning; the result determines the model)

 ${\it Table A.39 \ Wilkinson-Rogers \ Notation \ for \ Linear \ Models}$

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

Examples:

$$\begin{array}{lll} y \sim \ 1 \ + \ x & \text{corresponds to} \ y_i = (1 \ x_i)(\beta_1 \ \beta_2)^\top + \varepsilon \\ y \sim \ x & \text{short for} \ y \sim \ 1 \ + \ x \\ & \text{(a constant term is assumed implicitly)} \end{array}$$

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

Example:

$$\text{lm}(y \sim \text{poly(x, 4), data = experiment)}$$

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

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.

$egin{aligned} Model \ Administration \end{aligned}$	
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.

$Standard \ Analysis$		
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.	

Suppl.A-72 R AS A PROGRAMMING LANGUAGE AND ENVIRONMENT

For (generalised) linear models, various influence measures are provided. See help(influence). Some classical indicators are

$Influence\\ Diagnostics$		
rstandard()	standardised leave-one-out residuals.	
rstudent()	(externally) studentised residuals.	
dffits()	DFFITS influence on fit.	
dfbeta()	DFBETA influence on parameter estimation.	
dfbetas()	DFBETA influence on parameter estimation, standardised by a leave- one-out estimate of the coefficient standard error.	
covratio()	influence on variance of parameter estimation.	
cooks.distance()	Cook's distance (scaled as F-values).	
hatvalues()	leverage.	

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:

 $\hbox{``high level'' functions.} \quad \hbox{These define a new output.}$

"low level" functions. These modify an existing output.

parametrisations. These modify the settings of the graphics system.

Graphic devices can be opened explicitly. For example a call to pdf() will open a pdf device. Subsequent graphic output is written as pdf information to a file. The file must be closed by a balancing call to dev.off(). If no device is open, using a high-level graphics function will cause a default device to be opened. Usually this will direct graphic output to the screen. See help(Devices) for more information on graphic devices.

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.	

 $(cont.) \rightarrow$

Suppl.A-74 R AS A PROGRAMMING LANGUAGE AND ENVIRONMENT

High-Level Graphics (cont.)	
· ·	mosaic displays to visualise (standardised) residuals of a log-linear model for the table.
_	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 146).

$A.19.2\ Low-Level\ Graphics$

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 = "1")		
Sys.sleep()	suspends execution for a time interval.		
	$Syntax:$ $Sys.sleep(\langle seconds \rangle)$		
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:

- $\bullet\,$ rgl implements OpenGL for real-time 3d rendering,
- $\bullet\,$ 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,)
	$(\text{cont.}) \!\! \to \!\!$

Suppl.A-76 R AS A PROGRAMMING LANGUAGE AND ENVIRONMENT

Low Level Annotation		
(cont.)		
legend()	adds a legend block.	
	Syntax: legend(x, y = NULL, text,)	
	<pre>Example: plot(runif(100)); legend(locator(1), legend = "You clicked here")</pre>	
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 <code>abbreviate()</code>.

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 <code>expression()</code> and evaluated with <code>eval()</code> or <code>bquote()</code>.

Example:

```
 \begin{array}{lll} text(x, y, & expression(paste(bquote("(", atop(n, x), ")"), \\ & .(p) \\ & .(q) \\ & \end{array} ), \\ & \begin{array}{lll} (q) \\ & \\ & \end{array} ), \\ & \begin{array}{lll} (q) \\ & \end{array} ), \\ \\ & \begin{array}{lll} (q) \\ & \end{array} ), \\ \\ & \begin{array}{lll} (q) \\ & \end{array} ), \\ \\ \end{array} ), \\ \\ \end{array} ), \\ \end{array} ), \\ \begin{array}{lll} (q) \\ & \end{array} ), \\ \\ \end{array} ), \\ \\ \end{array} ), \\ \\ \end{array} ), \\ \\ \end{array} ), \\ \end{array} ), \\ \\ \}, \\ \\ \end{array} ), \\
```

 ${\tt demo(plotmath)} \ \ {\rm gives} \ \ {\rm several} \ \ {\rm examples} \ \ {\rm for} \ \ {\rm mathematical} \ \ {\rm typesetting} \ \ {\rm in} \ \ {\rm plots}.$

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)	
		$(\text{cont.}) \rightarrow$	

Parametrisations			
(cont.)			
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 [01]. 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.		

Suppl.A-78 R AS A PROGRAMMING LANGUAGE AND ENVIRONMENT 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 <code>Random</code>. 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 <code>help(.Random.seed))</code>. 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.

$\begin{array}{c} {\sf R} \; Random \\ Numbers \end{array}$			
.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)		

Suppl.A-80 R AS A PROGRAMMING LANGUAGE AND ENVIRONMENT

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()</pre>
```

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:

 $x \leftarrow runif(100)$ generates 100 random variables with U(0,1) distribution. qf(0.95, 10, 2) calculates the 95% quantile of the 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

 $(cont.) \rightarrow$

DISTRIBUTIONS, RANDOM NUMBERS, DENSITIES...

Suppl.A-81

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 <code>library(distr)</code> [18].

Suppl.A-82 R AS A PROGRAMMING LANGUAGE AND ENVIRONMENT

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" [16] gives details for computing on the language. See also Section 2.1.5, "Function objects" of [16].

Conversions		
parse()	converts input into a list of R expressions. <i>parse</i> 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))	
substitute()	See also 1.3: mathematical typesetting in plot annotations 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] Daniel Adler and Duncan Murdoch. rgl: 3D Visualization Device System (OpenGL), 2008. R package version 0.81.
- Anthony C. Atkinson and Marco Riani. Robust Diagnostic Regression Analysis. Springer, 2000.
- [3] Anthony C. Atkinson, Marco Riani, and Andrea Cerioli. Exploring Multivariate Data with the Forward Search. Springer, 2004.
- [4] David A. Belsley, Edwin Kuh, and Roy E. Welsch. Regression diagnostics: identifying influential data and sources of collinearity. John Wiley & Sons, New York-Chichester-Brisbane, 1980.
- [5] David B. Dahl and contributions from many others. xtable: Export Tables to LaTeX or HTML, 2008. R package version 1.5-3.
- [6] Dai Feng and Luke Tierney. Computing and displaying isosurfaces in R. Journal of Statistical Software, 28(1), 2008.
- [7] 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.
- [8] Robert Gentleman and Ross Ihaka. Lexical scope and statistical computing. *Journal of Computational and Graphical Statistics*, 9:491–508, 2000.
- [9] Gabor Grothendieck and Thomas Petzoldt. R help desk: Date and time classes in R. R News, 4(1):29-32, June 2004.
- [10] 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.
- [11] Bent Jørgensen. The Theory of Linear Models. Chapman & Hall, New York, 1993.
- [12] 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.
- [13] P. McCullagh and J.A. Nelder. *Generalized linear models*. Number 37 in Monographs on statistics and applied probability. London: Chapman & Hall, 2nd edition, 1989.
- [14] Paul Murrell. R Graphics. Chapman & Hall/CRC, Boca Raton, Fla., 2006.
- $[15]\ \ {\rm R}\ {\rm Development}\ {\rm Core}\ {\rm Team}.\ {\it R}\ {\it Data}\ {\it Import/Export},\, 2008.$
- [16] R Development Core Team. The R language definition, 2008.
- [17] R Development Core Team. Writing R Extensions, 2008.
- [18] Peter Ruckdeschel, Matthias Kohl, Thomas Stabla, and Florian Camphausen. S4 classes for distributions. R News, 6(2):2-6, May 2006.
- [19] Deepayan Sarkar. lattice: Lattice Graphics, 2008. R package version 0.17-15.

84 REFERENCES

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

- [21] William N. Venables and Brian D. Ripley. *Modern Applied Statistics with S.* Springer, Heidelberg, fourth edition, 2002.
- [22] Sanford Weisberg. Applied Linear Regression. Wiley Series in Probability and Statistics. Wiley, New York, third edition, 2005.

Functions and Variables by Topic

Topic $\mathbf{N}\mathbf{A}$	rug, Suppl.A-74
is. (type), Suppl.A-44	screen, Suppl.A-77
is.na, Suppl.A-45	segments, Suppl.A-74
Topic algebra	split.screen, Suppl.A-77
%*% , Suppl.A-69	symbols, Suppl.A-74
approx, Suppl.A-69	text, Suppl.A-75
beta, Suppl.A-69	title, Suppl.A-75
choose, Suppl.A-69	Topic arith
colMeans, Suppl.A-69	cummax, Suppl.A-78
colsum, Suppl.A-69	cummin, Suppl.A-78
diag, Suppl.A-69	cumprod, Suppl.A-78
eigen, Suppl.A-69	cumsum, Suppl.A-78
factorial, Suppl.A-69	max, Suppl.A-78
glm, Suppl.A-69	min, Suppl.A-78
lm, Suppl.A-69	prod, Suppl.A-78
matrix, Suppl.A-51	range, Suppl.A-78
nlm, Suppl.A-69	sort, Suppl.A-78
nls, Suppl.A-69	sum, Suppl.A-78
optim, Suppl.A-69	Topic array
qr, Suppl.A-69	%*% , Suppl.A-69
rowMeans, Suppl.A-69	aggregate, Suppl.A-52
rowsum, Suppl.A-69	aperm, Suppl.A-54
solve, Suppl.A-69	apply, Suppl.A-52
spline, Suppl.A-69	array, Suppl.A-52
svd, Suppl.A-69	cbind, Suppl.A-54
t, Suppl.A-69	${\tt colMeans}, {\tt Suppl.A-69}$
Topic aplot	colsum, Suppl.A-69
abline, 12, 13, Suppl.A-74	determinant, Suppl.A-69
arrows, Suppl.A-74	diag, Suppl.A-69
axis, Suppl.A-74	dim, Suppl.A-49
contour, 33, 34, Suppl.A-73	dimnames, $Suppl.A-51$
coplot, Suppl.A-73	eigen, Suppl.A-69
filled.contour, 33	expand.grid, Suppl.A-55
image, 33, 34, 36, Suppl.A-73	gl, Suppl.A-55
legend, Suppl.A-76	matrix, Suppl.A-51
lines, Suppl.A-74	merge, Suppl.A-55
mtext, Suppl.A-76	NCOL, Suppl.A-52
plot, 33	ncol, Suppl.A-51
points, Suppl.A-74	NROW, Suppl.A-52
polygon, Suppl.A-74	nrow, Suppl.A-51
rect, Suppl.A-74	outer, Suppl.A- 53

prop.table, Suppl.A-54 qr, Suppl.A-69 rbind, Suppl.A-54 rowMeans, Suppl.A-69 rowsum, Suppl.A-69 subset, Suppl.A-51 svd, Suppl.A-69 t, Suppl.A-54 var, Suppl.A-78 which, Suppl.A-51 Topic attribute attributes, Suppl.A-47 length, Suppl.A-47 mode, Suppl.A-44 names, Suppl.A-47 storage.mode, Suppl.A-47 str, Suppl.A-46 structure, Suppl.A-46 typeof, Suppl.A-44 Topic category by, Suppl.A-52 cut, Suppl.A-56 factor, Suppl.A-49 levels, Suppl.A-50 ordered, Suppl.A-49 prop.table, Suppl.A-54 split, Suppl.A-56 table, Suppl.A-55 tapply, Suppl.A-52 unsplit, Suppl.A-54 Topic character abbreviate, Suppl.A-76 casefold, Suppl.A-54 chartr, Suppl.A-55 grep, Suppl.A-55 gsub, Suppl.A-55 paste, Suppl.A-55 pmatch, Suppl.A-54 strsplit, Suppl.A-55 substr, Suppl.A-55 substring, Suppl.A-55 tolower, Suppl.A-54 toupper, Suppl.A-55 Topic chron chron, Suppl.A-66 Topic classes class, Suppl.A-47 data.frame, Suppl.A-50 is.vector, 1 methods, Suppl.A-46

Topic connection deparse, Suppl.A-82 dump, Suppl.A-64 read.csv, Suppl.A-66 read.fwf, Suppl.A-66 read.table, Suppl.A-65 scan, Suppl.A-66 sink, Suppl.A-64 source, Suppl.A-64 write, Suppl.A-64 write.csv, Suppl.A-66 Topic data apropos, Suppl.A-39 attach, 5, Suppl.A-50 bquote, Suppl.A-82 data, Suppl.A-64 deparse, Suppl.A-82 detach, 5, Suppl.A-50 environment, Suppl.A-41 eval, Suppl.A-82 find, Suppl.A-41 library, Suppl.A-67 require, Suppl.A-67 search, Suppl.A-48 searchpaths, Suppl.A-41 substitute, Suppl.A-82 sys.parent, Suppl.A-42 Topic debugging browser, Suppl.A-60 debug, Suppl.A-60 recover, Suppl.A-60 trace, Suppl.A-60 traceback, Suppl.A-61 untrace, Suppl.A-60 Topic device dev.off, Suppl.A-73 pdf, Suppl.A-73 split.screen, Suppl.A-77 Topic distribution .Random.seed, Suppl.A-79 hist, 36, Suppl.A-73 qqnorm, 36, Suppl.A-73 qqplot, Suppl.A-73 RNGkind, Suppl.A-79 sample, Suppl.A-79 set.seed, Suppl.A-79 Topic documentation apropos, Suppl.A-41 args, Suppl.A-39

data, Suppl.A-48

demo, Suppl.A-39	data.entry, Suppl.A-64
example, Suppl.A-39	dir, Suppl.A-48
find, Suppl.A-41	dump, Suppl.A-64
help, Suppl.A-39	load, Suppl.A-65
help.search, Suppl.A-39	package.skeleton, Suppl.A-67
help.start, Suppl.A-39	read.csv, Suppl.A-66
library, Suppl.A-40	read.csv2, Suppl.A-66
package.manager, Suppl.A-67	read.fwf, Suppl.A-66
RSiteSearch, Suppl.A-39	read.table, Suppl.A-65
str, Suppl.A-47	save, Suppl.A-64
vignette, Suppl.A-40	save.image, Suppl.A-65
Topic dplot	scan, Suppl.A-66
cloud, 34	sink, Suppl.A-64
densityplot, 36	source, Suppl.A-64
expression, Suppl.A-76	Sweave, Suppl.A-64
hist, 36, Suppl.A-73	system, Suppl.A-48
lattice.options, Suppl.A-76	write, Suppl.A-64
matplot, Suppl.A-73	write.csv, Suppl.A-66
mosaicplot, Suppl.A-74	write.csv2, Suppl.A-66
par, Suppl.A-76	write.table, Suppl.A-66
parallel, 36	Topic $\mathbf{grDevices}$
persp, 33, 34, 36, Suppl.A-73	devAskNewPage, Suppl.A-75
qq, 36	plotmath, Suppl.A-76
split.screen, Suppl.A-77	trans3d, 33
termplot, Suppl.A-74	Topic $\mathbf{graphics}$
trellis.par.set, Suppl.A-48	par, 35
wireframe, 35	points, 33
Topic dynamic	Topic \mathbf{hplot}
rggobi, Suppl.A-75	barchart, 36
rgl, 36, Suppl.A-75	barplot, 36
Topic environment	boxplot, 36, Suppl.A-73
apropos, Suppl.A-41	bwplot, 36
browser, Suppl.A-60	cloud, 34, 36
debug, Suppl.A-60	contour3d, 37
find, Suppl.A-41	contourplot, 36
lattice.options, Suppl.A-76	curve, 1, Suppl.A-73
ls, Suppl.A-48	dev.off, Suppl.A-73
objects, Suppl.A-43	dotchart, 36, Suppl.A-73
options, Suppl.A-48	dotplot, 36
par, Suppl.A-48	hist, 36, Suppl.A-73
rm, Suppl.A-43	histogram, 36
search, 5	image3d, 37
searchpaths, 5	matplot, Suppl.A-73
undebug, Suppl.A-60	mosaicplot, Suppl.A-74
Topic error	pairs, 36, Suppl.A-73
debug, Suppl.A-60	pdf, Suppl.A-73
options, Suppl.A-48	persp, 33, 34, 36, Suppl.A-73
trace, Suppl.A-60	plot, 36, Suppl.A-73
try, Suppl.A-61	qqmath, 36
Горіс file	qqnorm, 36, Suppl.A-73
-opio -ii-o	44.01.m, 50, Duppini 10

qqplot, 36, Suppl.A-73	dimnames, Suppl.A-51
splom, 36	${\tt duplicated}, {\tt Suppl.A-54}$
$\mathtt{stripchart},36$	$is.\langle type \rangle$, Suppl.A-44
$\mathtt{stripplot},36$	list, Suppl.A-50
termplot, Suppl.A-74	mapply, $Suppl.A-52$
wireframe, 35 , 36	match, Suppl.A-54
xyplot, 36	merge, Suppl.A-55
Topic htest	order, Suppl.A-78
wilcox.test, 32	rbind, Suppl.A-54
$wilcox_test, 32$	reshape, Suppl.A-55
Topic iplot	rev, Suppl.A-78
devAskNewPage, Suppl.A-75	seq, Suppl.A-55
getGraphicsEvent, Suppl.A-75	sort, Suppl.A-78
lattice.options, Suppl.A-76	split, Suppl.A-56
locator, Suppl.A-75	stack, Suppl.A-55
par, Suppl.A-48	str, Suppl.A-46
rggobi, Suppl.A-75	structure, Suppl.A-46
rgl, 36, Suppl.A-75	unique, Suppl.A-54
Sys.sleep, Suppl.A-75	unsplit, Suppl.A-56
Topic iteration	unstack, Suppl.A-56
apply, Suppl.A-52	Vectorize, Suppl.A-52
by, Suppl.A-52	which.max, Suppl.A-78
lapply, Suppl.A-52	which.min, Suppl.A-78
mapply, Suppl.A-52	Topic \mathbf{math}
replicate, Suppl.A-53	integrate, 1
sapply, Suppl.A-52	is.inf, Suppl.A-45
tapply, Suppl.A-50	is.nan, Suppl.A-45
Vectorize, Suppl.A-52	na.fail, Suppl.A-45
Topic list	na.omit, Suppl.A-45
lapply, Suppl.A-52	Topic $\mathbf{methods}$
list, Suppl.A-50	class, Suppl.A-47
replicate, Suppl.A-53	data.frame, Suppl.A-50
sapply, Suppl.A-52	methods, Suppl.A-48
Vectorize, Suppl.A-52	new, 29
Topic logic	setClass, 29
duplicated, Suppl.A-54	summary, Suppl.A-46
ifelse, 2	Topic misc
is. (type), Suppl.A-44	forward, 21
is.inf, Suppl.A-45	MASS, 21
is.na, Suppl.A-45	Topic \mathbf{models}
is.nan, Suppl.A-45	anova, Suppl.A-71
match, Suppl.A-54	contrasts, Suppl.A-71
na.fails, Suppl.A-45	dfbetas, 21
na.omit, Suppl.A-45	dffits, 21
unique, Suppl.A-54	expand.grid, Suppl.A-55
Topic manip	formula, Suppl.A-71
as. (type), Suppl.A-44	gl, Suppl.A-55
cbind, Suppl.A-54	glm, Suppl.A-71
cut, Suppl.A-56	influence.measures, 21
deparse, Suppl.A-82	model.matrix, Suppl.A-71
deparse, puppi.A-02	moder.matrix, Suppl.A-11

nls, Suppl.A-67	covratio, Suppl.A-72
rstandard, 21	dfbeta, Suppl.A-72
rstudent, 21	dfbetas, 21, Suppl.A-72
stdres, 21	dffits, 21, Suppl.A-72
studres, 21	formula, Suppl.A-71
terms, Suppl.A-71	glm, Suppl.A-71
update, Suppl.A-71	hatvalues, Suppl.A-72
Topic multivariate	influence.measures, 21
var, Suppl.A-78	lm, 11, 22, Suppl.A-71
Topic non-linear	nls, Suppl.A-71
nlm, Suppl.A-71	rstandard, 21, Suppl.A-72
nls, Suppl.A-71	rstudent, 21, Suppl.A-72
Topic optimize	stdres, 21
nlm, Suppl.A-71	studres, 21
optim, Suppl.A-69	Topic $\mathbf{sysdata}$
Topic print	.Random.seed, Suppl.A-79
cat, Suppl.A-46	RNGkind, Suppl.A-79
format, Suppl.A-46	set.seed, Suppl.A-79
ls.str, Suppl.A-41	Topic univar
options, Suppl.A-48	max, Suppl.A-78
print, 35, Suppl.A-46	mean, Suppl.A-78
str, Suppl.A-47	median, Suppl.A-78
write.table, Suppl.A-65	min, Suppl.A-78
Topic programming	order, Suppl.A-78
bquote, Suppl.A-76	quantile, Suppl.A-78
browser, Suppl.A-60	range, Suppl.A-78
debug, Suppl.A-60	rank, Suppl.A-78
deparse, Suppl.A-82	sort, Suppl.A-78
environment, Suppl.A-59	var, Suppl.A-78
eval, Suppl.A-76	which.max, Suppl.A-78
expression, Suppl.A-76	which.min, Suppl.A-78
missing, Suppl.A-58	Topic utilities
parse, Suppl.A-82	capabilities, Suppl.A-48
recover, Suppl.A-60	citation, Suppl.A-67
source, Suppl.A-64	data, Suppl.A-40
substitute, Suppl.A-82	data.entry, Suppl.A-64
Sweave, Suppl.A-64	demo, Suppl.A-39
sys.calls, 8	edit, Suppl.A-64
sys.frame, 8	example, Suppl.A-39
sys.parent, Suppl.A-42	getwd, Suppl.A-48
trace, Suppl.A-60	install.packages, Suppl.A-67
traceback, Suppl.A-61	library, Suppl.A-40
try, Suppl.A-61	ls.str, Suppl.A-41
undebug, Suppl.A-60	mapply, Suppl.A-52
untrace, Suppl.A-60	package.skeleton, Suppl.A-67
Vectorize, 3, 4, 8	Rprof, Suppl.A-61
Topic regression	Rprofmem, Suppl.A-61
anova, Suppl.A-71	setwd, Suppl.A-48
contrasts, Suppl.A-71	str, Suppl.A-47
cooks.distance, Suppl.A-72	summaryRprof, Suppl.A-61
	Zamaz J. Proz., Supplier Of

Sweave, Suppl.A-64 system, Suppl.A-48 system.time, Suppl.A-61 Vectorize, Suppl.A-52 vignette, Suppl.A-40

Function and Variable Index

 ${\tt abbreviate}, \, Suppl.A\text{-}55$ abline, 12, 13, Suppl.A-74 aggregate, Suppl.A-52anova, 24, Suppl.A-71 anova.lm, 25anscombe, 25 aov, 22, 23, 25 aperm, Suppl.A-54apply, Suppl.A-52approx, Suppl.A-69 apropos, Suppl.A-39 ${\tt args}, \, Suppl.A \text{-} 39$ array, Suppl.A-52 arrows, Suppl.A-74 as. $\langle type \rangle$, Suppl.A-44as.data.frame, 22 attach, 5, Suppl.A-50 attitude, 25 attributes, Suppl.A-47 $\mathtt{axis}, \, Suppl.A\text{-}74$

barchart, 36 barplot, 36 beta, Suppl.A-69 boxplot, 36, Suppl.A-73 bquote, Suppl.A-76 browser, Suppl.A-60 bwplot, 36 by, Suppl.A-52

capabilities, Suppl.A-48
casefold, Suppl.A-54
cat, Suppl.A-46
cbind, Suppl.A-55
chartr, Suppl.A-55
choose, Suppl.A-69
citation, Suppl.A-67
class, 24, Suppl.A-47
cloud, 34, 36
coef, 25

 ${\tt colMeans}, \, Suppl.A {\tt -} 69$ colsum, Suppl.A-69 confint, 25 contour, 33, 34, Suppl.A-73 contour3d, 37 contourplot, 36 contrasts, Suppl.A-71 cooks.distance, Suppl.A-72coplot, Suppl.A-73 covratio, Suppl.A-72 cummax, Suppl.A-78 cummin, Suppl.A-78 cumprod, Suppl.A-78 cumsum, Suppl.A-78 curve, 1, Suppl.A-73 cut, Suppl.A-56

data, Suppl.A-48 data.entry, Suppl.A-64 data.frame, Suppl.A-50 debug, Suppl.A-60 demo, Suppl.A-39 densityplot, 36deparse, Suppl.A-82 detach, 5, Suppl.A-50 determinant, Suppl.A-69 dev.off, Suppl.A-73 ${\tt devAskNewPage}, \, Suppl. A \text{-} 75$ ${\tt dfbeta}, \, Suppl.A \text{-} \textit{72}$ dfbetas, 21, Suppl.A-72 dffits, 21, Suppl.A-72diag, Suppl.A-69 dim, Suppl.A-51 dimnames, Suppl.A-51dir, Suppl.A-48 dotchart, 36, Suppl.A-73 dotplot, 36 dump, Suppl.A-64 duplicated, Suppl.A-54

edit, Suppl.A-64 effects, 24, 25 eigen, Suppl.A-69 environment, Suppl.A-41 eval, Suppl.A-82 example, Suppl.A-39 expand.grid, Suppl.A-55 expression, Suppl.A-82

factor, Suppl.A-49 factorial, Suppl.A-69 filled.contour, 33 find, Suppl.A-41 fitted, 25 format, Suppl.A-46 formula, 22, 23, Suppl.A-71 freeny, 25

getGraphicsEvent, Suppl.A-75 getwd, Suppl.A-48 gl, Suppl.A-55 glm, 25, Suppl.A-71 grep, Suppl.A-55 gsub, Suppl.A-55

hatvalues, Suppl.A-72 help, Suppl.A-39 help.search, Suppl.A-39 help.start, Suppl.A-39 hist, 36, Suppl.A-73 histogram, 36

ifelse, 2 image, 33, 34, 36, Suppl.A-73 image3d, 37 influence.measures, 21 install.packages, Suppl.A-67 integrate, 1 is. \langle type \rangle , Suppl.A-44 is.inf, Suppl.A-45 is.na, Suppl.A-45 is.nan, Suppl.A-45 is.nan, Suppl.A-45 is.vector, 1

lapply, Suppl.A-52 lattice.options, Suppl.A-48 legend, Suppl.A-76 length, Suppl.A-47 levels, Suppl.A-50 library, Suppl.A-48 LifeCycleSavings, 25 lines, Suppl.A-74 list, Suppl.A-50 lm, 11, 22, Suppl.A-71 lm.fit, 23, 25 lm.influence, 25 lm.wfit, 25 load, Suppl.A-65 locator, Suppl.A-75 longley, 25 ls, Suppl.A-41 ls.str, Suppl.A-41

mapply, Suppl.A-52 match, Suppl.A-54 matplot, Suppl.A-73 matrix, Suppl.A-51 max, Suppl.A-78 mean, Suppl.A-78 median, Suppl.A-78merge, Suppl.A-55 methods, Suppl.A-48 min, Suppl.A-78 missing, Suppl.A-58 mode, Suppl.A-47 model.frame, 24 model.matrix, 23, Suppl.A-71 model.matrix.default, 23 model.offset, 23 mosaicplot, Suppl.A-74 mtext, Suppl.A-76

na.exclude, 22na.fail, 22, Suppl.A-45na.omit, 22, Suppl.A-45names, Suppl.A-52ncol, Suppl.A-51new, 29nlm, Suppl.A-71nls, Suppl.A-71NROW, Suppl.A-52nrow, Suppl.A-51

objects, Suppl.A-41 offset, 23 optim, Suppl.A-69

Function and Variable Index

options, 22, Suppl.A-48 order, Suppl.A-78 ordered, Suppl.A-49 outer, Suppl.A-53

package.manager, Suppl.A-67 package.skeleton, Suppl.A-67 pairs, 36, Suppl.A-73 par, 35, Suppl.A-48 ${\tt parallel}, \, 36$ parse, Suppl.A-82 paste, Suppl.A-55 pdf, Suppl.A-73 persp, 33, 34, 36, Suppl.A-73 plot, 33, 36, Suppl.A-73 pmatch, Suppl.A-54 points, 33, Suppl.A-74 polygon, Suppl.A-74 $\mathtt{predict},\ 25$ predict.lm, 25 print, 35, Suppl.A-46 print.lm (lm), 22 prod, Suppl.A-78 programming (ifelse), 2 prop.table, Suppl.A-54

qq, 36 qqline (qqnorm), Suppl.A-73 qqmath, 36 qqnorm, 36, Suppl.A-73 qqplot, 36, Suppl.A-73 qr, Suppl.A-69 quantile, Suppl.A-78

range, Suppl.A-78
rank, Suppl.A-78
rbind, Suppl.A-54
read.csv, Suppl.A-66
read.fwf, Suppl.A-66
read.table, Suppl.A-65
recover, Suppl.A-60
rect, Suppl.A-74
replicate, Suppl.A-53
require, Suppl.A-67
reshape, Suppl.A-55
residuals, 25
rev, Suppl.A-78
rm, Suppl.A-43

RNGkind, Suppl.A-79
rowMeans, Suppl.A-69
rowsum, Suppl.A-69
Rprof, Suppl.A-61
Rprofmem, Suppl.A-61
RSiteSearch, Suppl.A-39
rstandard, 21, Suppl.A-72
rstudent, 21, Suppl.A-72
rug, Suppl.A-74

sample, Suppl.A-79 sapply, Suppl.A-52 save, Suppl.A-64 save.image, Suppl.A-65 scan, Suppl.A-66 screen, Suppl.A-77 search, 5, Suppl.A-41 searchpaths, 5, Suppl.A-41 segments, Suppl.A-74 seq, Suppl.A-55 set.seed, Suppl.A-79 setClass, 29 setwd, Suppl.A-48 $\mathtt{sink}, \, Suppl.A\text{-}64$ solve, Suppl.A-69 sort, Suppl.A-78 source, Suppl.A-64 spline, Suppl.A-69split, Suppl.A-56 split.screen, Suppl.A-77 splom, 36stack, Suppl.A-55 stackloss, 25 stdres, 21 storage.mode, Suppl.A-47 str, Suppl.A-46 stripchart, 36stripplot, 36 ${\tt strsplit}, \, Suppl.A-55$ structure, Suppl.A-47 studres, 21 subset, Suppl.A-51 substitute, Suppl.A-82 substr, Suppl.A-55 substring, Suppl.A-55 sum, Suppl.A-78 summary, Suppl.A-46 summary.lm, 25 summaryRprof, Suppl.A-61 svd, Suppl.A-69

Sweave, Suppl.A-64 swiss, 25 symbols, Suppl.A-74 sys.calls, 8 sys.frame, 8 sys.parent, Suppl.A-42 Sys.sleep, Suppl.A-75 system, Suppl.A-48 system.time, Suppl.A-61

t, Suppl.A-54 ${\tt table}, \, Suppl.A\text{-}55$ $\mathtt{tapply}, \ Suppl.A\text{-}50$ ${\tt termplot}, \, Suppl. A \text{-} \, \textit{74}$ $\mathtt{terms},\ 24,\ Suppl.A-71$ $\mathtt{text}, \ Suppl.A \text{-} \textit{74}$ $\mathtt{title}, \, Suppl.A\text{-}75$ ${\tt tolower}, \, Suppl.A\text{-}54$ ${\tt toupper}, \ Suppl.A\text{-}55$ ${\tt trace}, \, Suppl.A\text{-}60$ ${\tt traceback}, \, Suppl. A\text{-}61$ trans3d, 33 trellis.par.set, Suppl.A-48 try, Suppl.A-61 ts.intersect, 24 typeof, Suppl.A-44

undebug, Suppl.A-60 unique, Suppl.A-54 unsplit, Suppl.A-56 unstack, Suppl.A-56 untrace, Suppl.A-60 update, Suppl.A-71 utilities (integrate), 1

 $\begin{array}{l} \texttt{var}, \ Suppl.A\text{-}78 \\ \texttt{vcov}, \ 25 \\ \\ \texttt{Vectorize}, \ 3, \ 4, \ 8, \ Suppl.A\text{-}52 \\ \texttt{vignette}, \ Suppl.A\text{-}40 \end{array}$

which, Suppl.A-51 which.max, Suppl.A-78 which.min, Suppl.A-78 wilcox.test, 32 wilcox_test, 32 wireframe, 35, 36 write, Suppl.A-64 write.csv, Suppl.A-66

write.csv2, Suppl.A-66 write.table, Suppl.A-65

xyplot, 36