

Chapter 1

The Basics of R

Don't think—use the computer.

— G. Dyke

1.1 Introduction

The R software package is a powerful and flexible tool for statistical analysis which is used by practitioners and researchers alike. A basic understanding of R allows applying a wide variety of statistical methods to actual data and presenting the results clearly and understandably. This chapter provides help in setting up the programme and gives a brief introduction to its basics.

R is open-source software with a list of available, add-on packages that provide additional functionalities. This chapter begins with detailed instructions on how to install it on the computer and explains all the procedures needed to customise it to the user's needs.

In the next step, it will guide you through the use of the basic commands and the structure of the R language. The goal is to give an idea of the syntax so as to be able to perform simple calculations as well as structure data and gain an understanding of the data types. Lastly, the chapter discusses methods of reading data and saving datasets and results.

1.2 R on Your Computer

1.2.1 History of the R Language

R is a complete programming language and software environment for statistical computing and graphical representation. R is closely related to S, the statistical program-

ming language of Bell Laboratories developed by Becker and Chamber in 1984. It is actually an implementation of S with lexical scoping semantics inspired by Scheme, which started in 1992 and with the first results published by the developers Ihaka and Gentleman (1996) of the University of Auckland, NZ, for teaching purposes. Its name, R, is taken from the first names of the authors.

As part of the GNU Project, the source code of R has been freely available under the GNU General Public License since 1995. This decision contributed to spreading the software within the community of statisticians using free-code operating systems (OS). It is now a multi-platform statistical package widely known by people from many scientific fields such as mathematics, medicine and biology.

R enables its users to handle and store data, perform calculations on many types of variables, statistically analyse information under different aspects, create graphics and execute programmes. Its functionalities can be expanded by importing packages and including code written in C, C++ or Fortran. It is freely available on the Internet using the CRAN mirrors (Comprehensive R Archive Network at <http://cran.r-project.org/>). Since this chapter deals with installation issues and the basics of the R language, the reader familiar with the basics may skip it.

There exist several books about R, discussing specific topics in statistics and econometrics (biostatistics, etc.) or comparing R with other software, for example Stata. Typical users of Stata may be interested in Muenchen and Hilbe (2010). If the research topic requires Bayesian econometrics and MCMC techniques, Albert (2009) might be helpful. Two additional books on R, by Gaetan and Guyon (2009) and Cowpertwait and Metcalfe (2009), may support the development of R skills, depending on the application.

1.2.2 Installing and Updating R

Installing

As mentioned before, R is a free software package, which can be downloaded legally from the Internet page <http://cran.r-project.org/bin>.

Since R is a cross-platform software package, installing R on different operating systems will be explained. A full installation guide for all systems is available at <http://cran.r-project.org/doc/manuals/R-admin.html>.

Precompiled binary distributions

There are several ways of setting up R on a computer. On the one hand, for many operating systems, precompiled binary files are available. And on the other hand, for those who use other operating systems, it is possible to compile the programme from the source code.

- **Installing R under Unix**

Precompiled binary files are available for the *Debian*, *RedHat*, *SuSe* and *Ubuntu* Unix distributions. They can be found on the CRAN website at <http://cran.r-project.org/bin/linux>.

- **Installing R under Windows**

The binary version of R for Windows is located at <http://cran.r-project.org/bin/windows>.

If an account with Administrator privileges is used, R can be installed in the *Program Files* path and all the optional registry entries are automatically set. Otherwise, there is only the possibility of installing R in the user files path. Recent versions of Windows ask for confirmation to proceed with installing a programme from an ‘unidentified publisher’. The installation can be customised, but the default is suitable for most users.

For further information, it is suggested to visit <http://cran.r-project.org/bin/windows/base/rw-FAQ.html>

- **Installing R under Mac**

The current version of R for Mac OS is located at <http://cran.r-project.org/bin/macosx/>.

The installation package corresponding to the specific version of the Mac OS must be chosen and downloaded. During the installation, the Installer will guide the user through the necessary steps. Note that this will require the password or login of an account with administrator privileges. The installation can be customised, but the default is suitable for most users. After the installation, R can be started from the application menu.

For further information, it is suggested to visit <http://cran.r-project.org/bin/macosx/RMacOSX-FAQ.html>.

Updating

The best way to upgrade R is to uninstall the previous version of R, then install the new version and copy the old installed packages to the library folder of the new installation. Command `update.packages(checkBuilt = TRUE, ask = FALSE)` will update the packages for the new installation. Afterwards, any remaining data from the old installation can be deleted. Old versions of the software may be kept due to the parallel structure of the folders of the different installations. In cases where the user has a personal library, the contents must be copied into an update folder before running the update of the packages.

1.2.3 Packages

A package is a file, which may be composed of R scripts (for example functions) or dynamic link libraries (DLL) written in other languages, such as C or

Fortran, that gives access to more functions or data sets for the current session. Some packages are ready for use after the basic installation, others have to be downloaded and then installed when needed. On all operating systems, the function `install.packages()` can be used to download and install a package automatically through an available internet connection. Command `install.packages` may require to decide whether to compile and install sources if they are newer, then binaries. When installing packages manually, there are slight differences between operating systems.

Unix

Gzipped tar packages can be installed using the UNIX console by

```
R CMD INSTALL /your_path/your_package.tar.gz
```

Windows

In the R GUI, one uses the menu *Packages*.

- With an available internet connection, new packages can be downloaded and installed directly by clicking the *Install Packages* button. In this case, it is proposed to choose the CRAN mirror nearest to the user's location, and select the package to be installed.
- If the `.zip` file is already available on the computer, the package can be installed through *Install Packages from Zip files*.

Mac OS

There is a recommended *Package Manager* in the R.APP GUI. It is possible to install packages from the shell, but we suggest having a look at the FAQ on the CRAN website first.

All systems

Once a package is installed, it should be loaded in a session when needed. This ensures that the software has all the additional functions and datasets from this package in memory. This can be done through the commands

```
> library(package) or > require(package)
```

If the requested package is not installed, the function `library()` gives an error, while `require()` is designed for use inside of other functions and only returns `FALSE` and gives a warning.

The package will also be loaded as the second item in the system search path. Packages can also be loaded automatically if the corresponding code line is included in the `.Rprofile` file.

To see the installed libraries, the functions `library()` or `require()` are used.

```
> library()
```

To detach or unload a loaded package one uses.

```
> detach("package:name", unload = TRUE)
```

The function `detach()` can also be used to remove any R object from the search path. This alternative usage will be shown later in this chapter.

1.3 First Steps

After this first impression of what R is and how it works, the next steps are to see how it is used and to get used to it. In general, users should be aware of the **case sensitivity** of the R language.

It is also convenient to know that previously executed commands can be selected by the ‘up arrow’ on the keyboard. This is particularly useful for correcting typos and mistakes in commands that caused an error, or to re-run commands with different parameters.

1.3.1 “Hello World !!!”

As a first example, we will write some code that gives the output ‘Hello World !!!’ and a plot, see Fig. 1.1. There is no need to understand all the lines of the code now.

```
> install.packages("rworldmap")
> require(rworldmap)
> data("countryExData", envir = environment())
> mapCountryData(joinCountryData2Map(countryExData),
+   nameColumnToPlot = "EPI_regions",
+   catMethod        = "categorical",
+   mapTitle         = "Hello World!!!",
+   colourPalette     = "rainbow",
+   missingCountryCol = "lightgrey",
+   addLegend         = FALSE)
```

1.3.2 Getting Help

Once R has been installed and/or updated, it is useful to have a way to get help. To open the primary interface to the help system, one uses

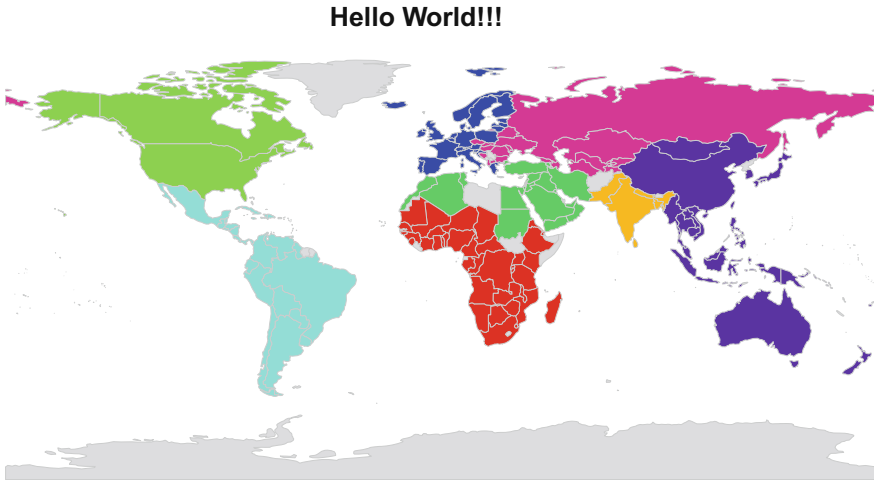



Fig. 1.1 “Hello World!!!” example in R.  [BCS_HelloWorld](#)

```
> help()
```

There are two ways of getting help for a particular function or command:

```
> help(function) and > ? function
```

To find help about packages, one uses

```
> help(package = package)
```

which returns the help file that comes with the specific package. If the different proposals of help were not satisfying, one can try

```
> help.search("function name") and > ?? "function name"
```

to see all help subjects containing “*function name*”. Finally, under Windows and Mac OS, under the *Help* menu are several PDF manuals which provide thorough and detailed information. The same help can be reached with the function

```
> help.start()
```

An HTML version without pictures can be found on the CRAN website. Examples for a particular topic can be found using

```
> example(topic)
```

1.3.3 Working Space

The current directory, where all pictures and tables are saved and from which all data is read by default, is known as the working directory. It can be found by `getwd()` and can be changed by `setwd()`.

```
> getwd()                                > setwd("your/own/path")
```

Each of the following functions returns a vector of character strings providing the names of the objects already defined in the current session.

```
> ls()                                    > objects()
```

To clear the R environment, objects can be deleted by

```
> rm(var1, var2)
```

The above line, for example, will remove the variables `var1` and `var2` from the working space. The next example will remove all objects defined in the current session and can be used to completely clean the whole working space.

```
> rm(list = ls())
```

The code below erases all variables, including the system ones beginning with a dot. Be cautious when using this command! Note that it has the same effect as the menu entry *Remove all objects* under Windows or *Clear Workspace* under Mac OS.

```
> rm(list = ls(all.names = TRUE))
```

However, we should always make sure that all previously defined variables are deleted or redefined when running new code, in order to be sure that there is no information left from the previous run of the programme which could affect the results. Therefore, it is recommended to have a line `rm(list = ls(all.names = TRUE))` in the beginning of each programme.

One saves the workspace as a *.Rdata* file in the specified working directory using the function `save.image()`, and saves the history of the commands in the *.R* format with `savehistory()`. Saving the workspace means keeping all defined variables in memory, so that the next time when R is in use, there is no need to define them again. If the history is saved, the variables will NOT be saved, whereas the commands defining them will be. So once the history is loaded, everything that was in the console should be compiled, but this can take a while for time-consuming calculations. The previously saved workspace and the history can be loaded with

```
> load(".Rdata")                        > loadhistory()
```

The `apropos('word')` returns the vector of functions, variables, etc., containing the argument *word*, as does `find('word')`, but with a different user interface. Without

going into details, the best way to set one's own search parameters is to consult the help concerning these functions.

Furthermore, a recommended and very convenient way of writing programmes is to split them into different modules, which might contain a list of definitions or functions, in order not to mess up the main file. They are executed by the function

```
> source("my_module.r")
```

To write the output in a separate `.txt` file instead of the screen, one uses `sink()`. This file appears in the working directory and shows the full output of the session.

```
> sink("my_output.txt")
```

To place that output back on the screen, one uses

```
> sink()
```

The simplest ways to quit R are

```
> quit() or > q()
```

1.4 Basics of the R Language

This section contains information on how R can be a useful software for all basic mathematical and programming needs.

1.4.1 *R as a Calculator*

R may be seen as a powerful calculator which allows dealing with a lot of mathematical functions. Classical fundamental operations as presented in Tables 1.1, 1.2 and 1.3 are, of course, available in R.

In contrast to the classical calculator, R allows assigning one or more values to a variable.

```
> a = pi + 0.5; a # create variable a; print a
[1] 3.641593
```


Table 1.1 Fundamental operations

Function name	Example	Result
Addition	<code>1 + 2</code>	3
Subtraction	<code>1 - 2</code>	-1
Multiplication	<code>1 * 2</code>	2
Division	<code>1 / 2</code>	0.5
Raising to a power	<code>3^2</code>	9
Integer division	<code>5 %/% 2</code>	2
Modulo division	<code>5 %% 2</code>	1

Table 1.2 Basic functions

Function name	Example	Result
Square root	<code>sqrt(2)</code>	1.414214
Sine	<code>sin(pi)</code>	1.224606e-16
Cosine	<code>cos(pi)</code>	-1
Tangent	<code>tan(pi/4)</code>	1
Arcsine	<code>asin(pi/4)</code>	0.903339
Arccosine	<code>acos(0)</code>	1.570796
Arctangent	<code>atan(1)</code>	0.785398
Arctan(y/x)	<code>atan2(1, 2)</code>	0.463647
Hyperbolic sine	<code>sinh(1)</code>	1.175201
Hyperbolic cosine	<code>cosh(0)</code>	1
Hyperbolic tangent	<code>tanh(pi)</code>	0.9962721
Exponential	<code>exp(1)</code>	2.718282
Logarithm	<code>log(1)</code>	0

Table 1.3 Comparison relations

Meaning	Example	Result
Smaller	<code>5 < 5</code>	FALSE
Smaller or equal	<code>3 <= 4</code>	TRUE
Bigger	<code>7 > 2</code>	TRUE
Bigger or equal	<code>5 >= 5</code>	TRUE
Unequal	<code>2 != 1</code>	TRUE
Logical equal	<code>pi == acos(-1)</code>	TRUE

Transformations of numbers are implemented by the following functions.

```
> floor(a)
[1] 3
> floor(-a)
[1] -4
> ceiling(a)
[1] 4
> ceiling(-a)
[1] -3
> round(a, digits = 2)
[1] 3.64

> trunc(a)
[1] 3
> trunc(-a)
[1] -3
> round(a)
[1] 4
> round(-a)
[1] -4
> factorial(a)
[1] 14.19451
```

`floor()` (`ceiling()`) returns the largest (smallest) integer that is smaller (larger) than the value of the given variable `a`, `trunc()` truncates the decimal part of a real-valued variable to obtain an integer variable. The function `round()` rounds a real-valued variable scientifically to an integer unless the argument `digits` is applied specifying the number of decimal places, in which case it scientifically rounds the given real number to that many decimal places. Scientific rounding of a real number rounds it to the closest integer, except for the case there the number after the predefined decimal place is exactly 5. For this case the closest even integer is returned. The function `factorial()`, which for an integer a returns $f(a) = a! = 1 \cdot 2 \cdot \dots \cdot a$, works with real-valued arguments as well, by using the Gamma function

$$\Gamma(x) = \int_0^{\infty} t^{x-1} \exp(-t) dt,$$

implemented by `gamma(x+1)` in R.

1.4.2 Variables

Assigning variables

There are different ways to assign variables to symbols.

```
> a = pi + 0.5; a      # assign (pi + 0.5) to a
[1] 3.641593
> b = a; b             # assign the value of a to b
[1] 3.641593
> d = e = 2^(1 / 2); d # assign 2^(1 / 2) to e
>                       # and the value of e to d
[1] 1.414214
> e
[1] 1.414214
> f <- d; f            # assign the value of d to f
[1] 1.414214
> d -> g; g            # assign the value of d to g
[1] 1.414214
```

Be careful with using ‘=’ for assigning, because the known argument, which defines the other, must be placed on the right side of the equals sign. The arrow assignment allows the following kind of constructions:

```
> h <- 4 -> j # assign the value 4 to the variables h and j
```

These constructions should not be used extensively due to their evident lack of clarity. Note that variable names are case sensitive and must not begin with a digit or a period followed by a digit. Furthermore, names should not begin with a dot as this is common only for system variables. It is often convenient to choose names that contain the type of the specific variable, e.g. for the variable `iNumber`, the ‘i’ at the beginning indicates that the variable is of the type integer.

```
> iNumber = c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

It is also useful to add the prefix ‘L’ to all local variables. For example, despite the fact that `pi` is a constant, one can reassign a different value to it. In order to avoid confusion, it would be convenient in this case to call the reassigned variable `Lpi`. We are not always following this suggestion in order to keep listings’ lengths as short as possible.

It is also possible to define functions in a similar fashion.

```
> Stirling = function(n){sqrt(2 * pi * n) * (n / exp(1))^n}
```

This is the mathematical function known as *Stirling’s formula* or *Stirling’s approximation* for the factorial, which has the property

$$\frac{n!}{\sqrt{2\pi n} \cdot (n/e)^n} \rightarrow 1, \quad \text{as } n \rightarrow \infty.$$

For multi-output functions, see Sect. 1.4.6.

Working with variables

There are different *types* of variables in R. A brief summary is given in Table 1.4. Obviously each variable requires its own storage space, so during computationally intensive calculations one should pay attention to the choice of the variables. The types *numeric* and *double* are identical. Both are vectors of a specific length and store real valued elements. A variable of the type *logical* contains only the values TRUE and FALSE.

The command returning the type, i.e. the storage mode, of an object is `typeof()`, the possible values are listed in the structure `TypeTable`. Alternatively, the function `class()` can be used, which in turn returns the class of the object and is often used in the object-oriented style of programming.

```
> typeof(object name)
```

```
> class(object name)
```

A *character* variable consists of elements within quotation marks.

Table 1.4 Variable types

Variable type	Example of memory needed	Result
Numeric	<code>object.size(numeric(2))</code>	56 bytes
Logical	<code>object.size(logical(2))</code>	48 bytes
Character	<code>object.size(character(2))</code>	104 bytes
Integer	<code>object.size(integer(2))</code>	48 bytes
Double	<code>object.size(double(2))</code>	56 bytes
Matrix	<code>object.size(matrix(0, ncol = 2))</code>	216 bytes
List	<code>object.size(list(2))</code>	96 bytes

```
> a = character(length = 2); a
[1] " " " "
> a = c(exp(1), "exp(1)")
> class(a)
[1] "character"
> a
[1] "2.71828182845905" "exp(1)"
```

A description of the variable types *matrix* and *list* is given in Sects. 1.4.3 and 1.4.5 in more detail. To show, or *print*, the content of `a`, one uses the function `print()`.

```
> print(a)
```

‘Unknown values’ as a result of missing values is a common problem in statistics. To handle this situation, R uses the value `NA`. Every operation with `NA` will give an `NA` result as well. Note that `NA` is different from the value `NaN`, which is the abbreviation for ‘not a number’. If R returns `NaN`, the underlying operation is not valid, which obviously has to be distinguished from the case in which the data is not available. A further output, that the reader should worry about, is `Inf` denoting infinity. It is difficult to work with such results, but R provides tools which modify the class of results, like `NaN`, and there are functions to help in transforming variables from one type into another. The functions `as.character`, `as.double`, `as.integer`, `as.list`, `as.matrix`, `as.data.frame`, etc., coerce their arguments to the function specific type. It is therefore possible to transform results like `NaN` or `Inf` to a preferable type, which is often done in programming with R.

In the example below, the function `paste()` automatically converts the arguments to strings, by using the function `as.character()`, and concatenates them into a single string. The option `sep` specifies the string character that is used to separate them.

```
> paste(NA, 1, "hop", sep = "@") # concatenate objects, separator @
[1] "NA@1@hop"
> typeof(paste(NA, 1, "hop", sep = "@"))
[1] "character"
```

Furthermore, one can check if an R object is finite, infinite, unknown, or of any other type. The function `is.finite(argument)` returns a Boolean object (a vector or matrix, if the input is a vector or a matrix) indicating whether the values are finite or not. This test is also available to test types, such as `is.integer(x)`, to test whether x is an integer or not, etc.

```
> x = c(Inf, NaN, 4)
> is.finite(x)           # check if finite
[1] FALSE FALSE  TRUE
> is.nan(x)             # check if NaN (operation not valid)
[1] FALSE TRUE  FALSE
> is.double(x)          # check if type double
[1] TRUE
> is.character(x)       # check if type character
[1] FALSE
```

1.4.3 Arrays

A vector is a one-dimensional array of fixed length, which can contain objects of one type only. There are only three categories of vectors: *numerical*, *character* and *logical*.

Basic manipulations

Joining the values 1 , π , $\sqrt{2}$ in a vector is done easily by the concatenate function, `c`.

```
> v = c(1, pi, sqrt(2)); v      # concatenate
[1] 1.000000 3.141593 1.414214
```

The i th element of the vector can be addressed using `v[i]`.

```
> v = c(1.000000, 3.141593, 1.414214)
> v[2]                      # 2nd element of v
[1] 3.141593
```

The indexing of a vector starts from 1. If an element is addressed that does not exist, e.g. `v[0]`, the error `NA` or `numeric(0)` is returned. A numerical vector may be *integer* if it contains only integers, *numeric* if it contains only real numbers, and *complex* if it contains complex numbers. The length of a vector object `v` is found through

```
> v = c(1.000000, 3.141593, 1.414214)
> length(v)                 # length of vector v
[1] 3
```

Be careful with this function, keeping in mind that it *always returns one value, even for multi-dimensional arrays*, so one should know the nature of the objects one is dealing with.

One easily applies the same transformation to all elements of a vector. One can calculate, for example, the *elementwise inverse* with the command `^(-1)`. This is still the case for other objects, such as arrays.

```
> v = c(1.000000, 3.141593, 1.414214)
> d = v + 3; d
[1] 4.000000 6.141593 4.414214
> v^(-1)
[1] 1.0000000 0.3183099 0.7071068
> v * v^(-1)
[1] 1 1 1
```

There are a lot of other ways to construct vectors. The function `array(x, y)` creates an array of dimension `y` filled with the value `x` only. The function `seq(x, y, by = z)` gives a sequence of numbers from `x` to `y` in steps of `z`. Alternatively, the required length can be specified by option `length.out`.

```
> c(1, 2, 3)
[1] 1 2 3
> 1:3
[1] 1 2 3
> array(1:3, 6)
[1] 1 2 3 1 2 3
> seq(1, 3)
[1] 1 2 3
> seq(1, 3, by = 2)
[1] 1 3
> seq(1, 4, length.out = 5)
[1] 1.00 1.75 2.50 3.25 4.00
```

One can also use the `rep()` function to create a vector in which some values are repeated.

```
> v = c(1.000000, 3.141593, 1.414214)
> rep(v, 2) # the vector twice
[1] 1.00 3.14 1.41 1.00 3.14 1.41
> rep(v, c(2, 0, 1)) # 1st value twice, no 2nd value
# 3rd value once
[1] 1.00 1.00 1.41
> rep(v, each = 2) # each value twice
[1] 1.00 1.00 3.14 3.14 1.41 1.41
```

With the second command of the above code, R creates a vector in which the first value of `v` should appear two times, the second zero times, and the third only once. Note that if the second argument is not an integer, R takes the rounded value. In the last call, each element is repeated twice, proceeding element per element.

The names of the months, their abbreviations and all letters of the alphabet are stored in predefined vectors. The months can be addressed in the vector `month.name[]`. For their abbreviations, use `month.abb[]`. Letters are stored in `letters[]` and capital letters in `LETTERS[]`.

```
> s = c(2, month.abb[2], FALSE, LETTERS[6]); s
[1] "2"      "Feb"     "FALSE"   "F"
> class(s)
[1] "character"
```

Note that if one element in a vector is of type `character`, then all elements in the vector are converted to `character`, since a vector can only contain objects of one type.

To keep only some specific values of a vector, one can use different methods of conditional selection. The first is to use logical operators for vectors in R: “!” is the logical NOT, “&” is the logical AND and “|” is the logical OR. Using these commands, it is possible to perform a conditional selection of vector elements. The elements for which the conditions are `TRUE` can then, for example, be saved in another vector.

```
> v = c(1.000000, 3.141593, 1.414214)
> v > 0                                # element greater 0
[1] TRUE TRUE TRUE
> (v != 1) & (v > 0)                  # element not equal to 1 and greater 0
[1] FALSE TRUE TRUE
```

In the last example, the first value is bigger than zero, but equal to one, so `FALSE` is returned. This method may be a little bit confusing for beginners, but it is very useful for working with multi-dimensional arrays.

Multiple selection of elements of a vector may be done using another vector of indices as arguments in the square brackets.

```
> v = c(1.000000, 3.141593, 1.414214)
> v[c(1, 3)]                          # 1st and 3rd element
[1] 1.000000 1.414214
> w = v[(v != 1) & (v > 0)]; w         # save the specified elements in w
[1] 3.141593 1.414214
```

To eliminate specific elements in a vector, the same procedure is used as for selection, but a minus sign indicates the elements which should be removed.

<pre>> v = c(1.0000, 3.1416, 1.4142) > v[-1] # exclude first [1] 3.1416 1.4142</pre>	<pre>> > v[-c(1, 3)] # excl. 1st and 3rd [1] 3.141593</pre>
---	---

For a one-dimensional vector function, which returns the index or indices of specific elements.

```
> v = c(1.000000, 3.141593, 1.414214)
> which(v == pi)      # indices of elements that fulfill the condition
[1] 2
```

There are different functions for working with vectors. Extremal values are found through the functions `min` and `max`, which return the minimal and maximal values

of the vector, respectively.

```
> v = c(1.0000, 3.1416, 1.4142)
> min(v)
[1] 1.000000
>
> max(v)
[1] 3.141593
```

However, this can be done simultaneously by the function `range`, which returns a vector consisting of the two extreme values.

```
> v = c(1.000000, 3.141593, 1.414214)
> range(v)
[1] 1.000000 3.141593
# min and max value
```

Joining the function `which()` with `min` or `max`, one gets the function `which.min` or `which.max` that returns the index of the smallest or largest element of the vector, respectively, and is equivalent to `which(x == max(x))` and `which(x == min(x))`.

Quite often, the elements of a vector have to be sorted before one can proceed with further transformations. The simplest function for this purpose is `sort()`.

```
> x = c(4, 2, 5, 7, 1, 9, 0, 3)
> sort(x)
[1] 0 1 2 3 4 5 7 9
# values in increasing order
```

Being a function, it does not modify the original vector `x`. To get the coordinates of the elements that are in the sorted vector, we use the function `rank()`.

```
> x = c(4, 2, 5, 7, 1, 9, 0, 3)
> rank(x)
[1] 5 3 6 7 2 8 1 4
# rank of elements in increasing order
```

In this example, the first value of the result is '5'. This means that the first element in the original vector `x[1] = 4` is in the fifth place in the ordered vector. The inverse function to `rank()` is `order()`, which states the position of the element of the sorted vector in the original vector, e.g. the smallest element in `x` is the seventh, the second smallest is the fifth, etc.

```
> x = c(4, 2, 5, 7, 1, 9, 0, 3)
> order(x)
[1] 7 5 2 8 1 3 4 6
# positions of sorted elements in the original vector
```

Replacing specific values in a vector is done with the function `replace()`. This function replaces the elements of `x` that are specified by the second argument by the values given in the third argument.

Table 1.5 Cumulative functions

Meaning	Implementation	Result
Sum	<code>cumsum(1:10)</code>	1 3 6 10 15 21 28 36 45 55
Product	<code>cumprod(1:5)</code>	1 2 6 24 120
Minimum	<code>cummin(c(3:1,2:0,4:2))</code>	3 2 1 1 1 0 0 0 0
Maximum	<code>cummax(c(3:1,2:0,4:2))</code>	3 3 3 3 3 3 4 4 4

```

> v = 1:10; v
[1] 1 2 3 4 5 6 7 8 9 10
> replace(v, v < 3, 12)      # replace all els. smaller than 3 by 12
[1] 12 12 3 4 5 6 7 8 9 10
> replace(v, 6, 12)         # replace the 6th element by 12
[1] 1 2 3 4 5 12 7 8 9 10

```

The second argument is a vector of indices for the elements to be replaced by the values. In the second line, all numbers smaller than 3 are to be replaced by 12, while in the last line, the element with index 6 is replaced by 12. Note again that functions do not change the original vectors, so that the last output does not show 1 and 2 replaced by 12 after the second command.

There are also a few more functions for vectors which are of further interest. The function `rev()` returns the elements in reversed order, and `sum()` gives the sum of all the elements in the vector.

```

> x = c(4, 2, 5, 7, 1, 9, 0, 3)
> rev(x)                      # reverse the order of x
[1] 3 0 9 1 7 5 2 4
> sum(x)                      # sum all elements of x
[1] 31

```

More sophisticated ways of cumulative summation of the elements of vectors are given in Table 1.5.

Vectors can also be considered as sets, and for this purpose there exist binary set operators, such as `a %in% b`, which gives the elements of `a` that are also in `b`. More advanced functions for working with sets are discussed in Chap. 3.

```

> a = 1:3                      # 1 2 3
> b = 2:6                      # 2 3 4 5 6
> a %in% b                     # FALSE TRUE TRUE
> b %in% a                     # TRUE TRUE FALSE FALSE FALSE
> a = c("A", "B")             # "A" "B"
> b = LETTERS[2:6]            # "B" "C" "D" "E" "F"
> a %in% b                     # FALSE TRUE
> b %in% a                     # TRUE FALSE FALSE FALSE FALSE

```

In algebra and statistics, matrices are fundamental objects, which allow summarising a large amount of data in a simple format. In R, matrices are only allowed to have one data type for their entries, which is their main difference from data frames, see Sect. 1.4.4.

Creating a matrix

There are many possible ways to create a matrix, as shown in the example below. The function `matrix()` constructs matrices with specified dimensions.

```
> matrix(0, 2, 5) # zeros, 2x5
      [,1] [,2] [,3] [,4] [,5]
[1,]    0    0    0    0    0
[2,]    0    0    0    0    0

> matrix(1:12, nrow = 3)
      [,1] [,2] [,3] [,4]
[1,]     1     4     7    10
[2,]     2     5     8    11
[3,]     3     6     9    12

> matrix(1:6, nrow = 2, byrow = TRUE)
      [,1] [,2] [,3]
[1,]     1     2     3
[2,]     4     5     6
```

In the third matrix, in the above example, the argument `byrow = TRUE` indicates that the filling must be done by rows, which is not the case in the second matrix, where the matrix was filled by columns (*column-major storage*), the function `as.vector(matrix)` converts a matrix into a vector. If the matrix has more than one row or column, the function concatenates the columns into a vector. One can also construct diagonal matrices using `diag()`, see Sect. 2.1.1.

Another way to transform a given vector into a matrix with specified dimensions is the function `dim()`. The function `t()` is used to transpose matrices.

```
> m = 1:6
> dim(m) = c(2, 3); m
      [,1] [,2] [,3]
[1,]     1     3     5
[2,]     2     4     6

> t(m) # transpose m
      [,1] [,2]
[1,]     1     2
[2,]     3     4
[3,]     5     6
```

Coupling vectors using the functions `cbind()` (column bind) and `rbind()` (row bind) joins vectors column-wise or row-wise into a matrix.

```
> x = 1:6
> y = LETTERS[1:6]
> rbind(x, y) # bind vectors row-wise
 [,1] [,2] [,3] [,4] [,5] [,6]
x  "1"  "2"  "3"  "4"  "5"  "6"
y  "A"  "B"  "C"  "D"  "E"  "F"
```

The functions `col` and `row` return the column and row indices of all elements of the argument, respectively.

```
> m = matrix(1:6, ncol = 3)
> col(m)      # column-indices
      [,1] [,2] [,3]
[1,]    1    2    3
[2,]    1    2    3

> m = matrix(1:6, ncol = 3)
> row(m)      # row-indices
      [,1] [,2] [,3]
[1,]    1    1    1
[2,]    2    2    2
```

The procedure to extract an element or submatrix uses a syntax similar to the syntax for vectors. In order to extract a particular element, one uses `m[row index, column index]`. As a reminder, in the example below, 10 is the second element of the fifth column, in accordance with the standard mathematical convention.

```
> k = matrix(1:10, 2, 5); k      # create a matrix
      [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10

> k[2, 5]                        # select element in row 2, column 5
[1] 10
```

One can combine this with `row()` and `col()` to construct a useful tool for the conditional selection of matrix elements. For example, extracting the diagonal of a matrix can be done with the following code.

```
> m = matrix(1:6, ncol = 3)
> m[row(m) == col(m)]           # select elements [1, 1]; [2, 2]; etc.
[1] 1 4
```

The same result is obtained by using the function `diag(m)`. To better understand the process, note that the command `row(m) == col(m)` creates just the Boolean matrix below and all elements with value `TRUE` are subsequently selected.

```
> row(m) == col(m)              # condition (row index = column index)
      [,1] [,2] [,3]
[1,]  TRUE FALSE FALSE
[2,] FALSE  TRUE FALSE
```

This syntax can also be used to select whole rows or columns.

```
> y = matrix(1:16, ncol = 4, nrow = 4); y
      [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16
> y[2, ]      # second row
[1]  2  6 10 14
> y[, 2]      # second column
[1] 5 6 7 8
> y[2]        # second element (column-wise)
[1] 2
> y[1:2, 3:4] # several rows and columns
      [,1] [,2]
[1,]    9   13
[2,]   10   14
```

The first command selects the second row of `y`. The second command selects the 2nd column. The third line considers the matrix as a succession of column vectors and gives, according to this construction, the second element. The last call selects a range of rows and columns.

Many functions can take matrices as an argument, such as `sum()` or `product()`, which will calculate the sum or product of all elements in the matrix, respectively. The functions `colSums()` and `rowSums()` can be used to calculate the column-wise or row-wise sums. All classical binary operators are implemented element-by-element. This means that, for example, `x * y` returns the Kronecker product, not the classical matrix product discussed in Sect. 2.1 on matrix algebra.

One can assign names to the rows and columns of a matrix using the function `dimnames()`. Alternatively, the column and row names can be assigned separately by `colnames()` and `rownames()`, respectively.

```
> A = matrix(1:20, ncol = 5, nrow = 4)
> dimnames(A) = list(letters[4:7], letters[5:9]) # name dimensions
> A
  e f g h i
d 1 5 9 13 17
e 2 6 10 14 18
f 3 7 11 15 19
g 4 8 12 16 20
> A[2, 2]
[1] 6
> A["b", "b"]
[1] 6
```

This leads directly to another very useful format in R: the data frame.

1.4.4 Data Frames

A data frame is a very useful object, because of the possibility of collecting data of different types (numeric, logical, factor, character, etc.). Note, however, that all elements must have the same length. The function `data.frame()` creates a new data frame object. It contains several arguments, as the column names can be directly specified with `data.frame(..., row.names = c(), ...)`. A further possibility for creating a data frame is to convert it from a matrix with the `as.data.frame(matrix name)` function.

Basic manipulations

Consider the following example, which constructs a data frame.

```
> cities = c("Berlin", "New York", "Paris", "Tokyo")
> area = c(892, 1214, 105, 2188)
> population = c(3.4, 8.1, 2.1, 12.9)
> continent = factor(c("Europe", "North America", "Europe", "Asia"))
> myframe = data.frame(cities, area, population, continent)
> is.data.frame(myframe) # check if object is a dataframe
[1] TRUE
> rownames(myframe) = c("Berlin", "New York", "Paris", "Tokyo")
```

```
> colnames(myframe) = c("City", "Area", "Pop.", "Continent")
> myframe
```

	City	Area	Pop.	Continent
Berlin	Berlin	892	3.4	Europe
New York	New York	1214	8.1	North America
Paris	Paris	105	2.1	Europe
Tokyo	Tokyo	2188	12.9	Asia

Note that if we defined the above data frame as a matrix, then all elements would be converted to type `character`, since matrices can only store one data type.

`data.frame()` automatically calls the function `factor()` to convert all character vectors to factors, as it does for the `Continent` column above, because these variables are assumed to be indicators for a subdivision of the data set. To perform data analysis (e.g. principal component analysis or cluster analysis, see Chap. 8), numerical expressions of character variables are needed. It is therefore often useful to assign ordered numeric values to character variables, in order to perform statistical modelling, set the correct number of degrees of freedom, and customise graphics. These variables are treated in R as factors. As an example, a new variable is constructed, which will be added to the data frame “*myframe*”. Three position categories are set, according to the proximity of each city to the sea: Coastal (‘0’), Middle (‘1’) and Inland (‘2’). These categories follow a certain order, with Middle being in between the others, which needs to be conveyed to R.

```
> e = c(2, 0, 2, 0) # code info. in e
> f = factor(e, level = 0:2) # create factor f
> levels(f) = c("Coastal", "Middle", "Inland"); f # with 3 levels
[1] Inland Coastal Inland Coastal
Levels: Coastal Middle Inland
> class(f)
[1] "factor"
> as.numeric(f)
[1] 3 1 3 1
```

The variable `f` is now a factor, and levels are defined by the function `levels()` in the 3rd line in decreasing order of the proximity to the sea. When sorting the variable, R will now follow the order of the levels. If the position values were simply coded as *string*, i.e. Coastal, Middle and Inland, any sorting would be done *alphabetically*. The first level would be Coastal, but the second Inland, which does not follow the inherited order of the category.

The function `as.numeric()` extracts the numerical coding of the levels and the indexation begins now with 1.

```
> myframe = data.frame(myframe, f)
> colnames(myframe)[5] = "Prox.Sea" # name 5th column
> myframe
```

	City	Area	Pop.	Continent	Prox.Sea
Berlin	Berlin	892	3.4	Europe	Inland
New York	New York	1214	8.1	North America	Coastal
Paris	Paris	105	2.1	Europe	Inland
Tokyo	Tokyo	2188	12.9	Asia	Coastal

The column names for columns 1 to 4 are the ones that were assigned before, since `myframe` is used in the call of `data.frame()`. Note that one should not use names with spaces, e.g. `Sea.Env.` instead of `Sea. Env.` To add columns or rows to a data frame, one can use the same functions as for matrices, or the procedure described below.

```
> myframe = cbind(myframe, "Language.Spoken" =
+   c("German", "English", "French", "Japanese"))
> myframe
```

	City	Area	Pop.	Continent	Prox.Sea	Language.Spoken
Berlin	Berlin	892	3.4	Europe	Inland	German
New York	New York	1214	8.1	North America	Coastal	English
Paris	Paris	105	2.1	Europe	Inland	French
Tokyo	Tokyo	2188	12.9	Asia	Coastal	Japanese

There are several ways of addressing one particular column by its name: `myframe$Pop.`, `myframe[, 3]`, `myframe[, "Pop."]`, `myframe["Pop."]`. All these commands except the last return a *numeric vector*. The last command returns a *data frame*.

```
> myframe$Pop.      # select only population column
[1] 3.4 8.1 2.1 12.9
> myframe["Pop."]  # population column as dataframe
  Pop.
Berlin 3.4
New York 8.1
Paris 2.1
Tokyo 12.9
> myframe[3] == myframe["Pop."]
  Pop.
Berlin TRUE
New York TRUE
Paris TRUE
Tokyo TRUE
```

The output of the above code is a data frame and, therefore, can not be indexed like a vector. One uses `$` notation similar to addressing fields of objects in the C++ programming language.

```
> myframe[2, 3]     # select 3rd entry of 2nd row
[1] 8.1
> myframe[2, ]      # select 2nd row
      City Area Pop.      Continent Prox.Sea Language.Spoken
New York New York 1214 8.1 North America Coastal      English
```

Long names for data frames and the contained variables should be avoided, because the source code becomes very messy if several of them are called. This can be solved by the function `attach()`. Attached data frames will be set to the search path and the included variables can be called directly. Any R object can be attached. To remove it from the search path, one uses the function `detach()`.

```
> rm(area)          # remove var. "area" to avoid confusion
> attach(myframe)    # attach dataframe "myframe"
> Area              # specify column Area in attached frame
[1] 892 1214 105 2188
> detach(myframe)
```

If two-word names are used, it is advised to label the data frame or variable with a block name, so that the two words in the name are connected with a dot or an underline, e.g. `Language.Spoken`. This avoids having to put names in quotes.

One of the easiest ways to edit a data frame or a matrix is through interactive tables, called by the `edit` function. Note that the `edit()` function does not allow changing the original data frame.

```
> edit(myframe)
```

If the modifications are to be saved, the function `fix()` is employed. It opens a table like `edit()`, but the changes in the data are stored.

```
> fix(myframe)
```

A data frame as a database

Furthermore, R provides the possibility of selecting subsets of a data frame by using the logical operators discussed in Sects. 1.4.1 and 1.4.3: `<`, `>`, `=<`, `>=`, `==`, `!=`, `&`, `|` and `!`.

```
> myframe[(myframe$Language.Spoken == "French") |
+ (myframe$Pop. > 10), -1]
  Area Pop. Continent Prox.Sea Language.Spoken
Paris  105   2.1    Europe   Inland         French
Tokyo 2188 12.9     Asia    Coastal        Japanese
> myframe[, -c(1, 2, 3, 5)] # select all except specified columns
  Continent Language.Spoken
Berlin      Europe         German
New York North America      English
Paris       Europe         French
Tokyo       Asia          Japanese
```

The first command of the last listing selects both the cities in which French is spoken or the cities with more than 10 million inhabitants. The second command selects only the first, fourth and sixth variables for display. As explained above, the individual data, as well as rows and columns, can be addressed using the square brackets. If no variable is selected, i.e. `[,]`, all information about the observations is kept.

The following functions are also helpful for conditional selections from data frames. The function `subset()`, which performs conditional selection from a data frame, is frequently used when only a subset of the data is used for the analysis.

```
> subset(myframe, Area > 1000)
  City Area Pop. Continent Prox.Sea Language.Spoken
New York New York 1214   8.1 North America Coastal      English
Tokyo      Tokyo  2188 12.9         Asia    Coastal        Japanese
```

A conditional transformation of the data frames, by adding a new variable which is a function of others, is done by using the function `transform()`. As an example, a new variable `Density` is added to our data frame.

```
> transform(myframe[, -c(1, 4, 5)], Density = Pop. * 10^6 / Area)
      Area Pop. Language.Spoken   Density
Berlin   892  3.4         German 3811.659
New York 1214  8.1         English 6672.158
Paris    105  2.1         French 20000.000
Tokyo    2188 12.9        Japanese 5895.795
```

Another way to extract data according to the values is based on addressing specific variables. In the next example, the interest is in the cumulative area of cities that are not inland.

```
> Area.Seasiders = myframe$Area[myframe$Prox.Sea == "Middle"
+ | myframe$Prox.Sea == "Coastal"]
> Area.Seasiders
[1] 1214 2188
> sum(Area.Seasiders)
[1] 3402
```

The important technique of sorting the data frame is illustrated below. Remember that `order()` sorts the elements and returns their ranks in the original vector. The optional argument `partial` specifies the columns for subsequent ordering, if necessary. It is used to order groups of data according to one column and order the values in each group according to another column.

```
> myframe[order(myframe$Pop., partial = myframe$Area), ]
      City Area Pop.   Continent Prox.Sea Language.Spoken
Paris   Paris  105  2.1     Europe   Inland         French
Berlin  Berlin  892  3.4     Europe   Inland         German
New York New York 1214  8.1 North America Coastal       English
Tokyo   Tokyo  2188 12.9     Asia     Coastal       Japanese
```

1.4.5 Lists

Lists are very flexible objects which, unlike matrices and data frames, may contain variables of different types and lengths.

The simplest way to construct a list is by using the function `list()`. In the following example, a string, a vector and a function are joined into one variable.

```
> a = c(2, 7)
> b = "Hello"
> d = list(example = Stirling, a, end = b)
> d
$example
function(x){
  sqrt(2 * pi * x) * (x / exp(1))^x
}

[[2]]
[1] 2 7

$end
[1] "Hello"
```


Another way to join these into a list is through a *vector* construction.

```
> z = c(Stirling, a)
> typeof(z)
[1] "list"
```

To address the elements of a list object, one again uses '\$', the same syntax as for a data frame.

```
> d$end
[1] "Hello"
```

A list can be transformed into a 1-element list, i.e. a list of length 1, using `unlist`. In this example, the element `[[2]]` of list `d` is split into two elements, each of length 1.

```
> unlist(d) # transform to list with elements of length 1
$example
function(x){
  sqrt(2 * pi * x) * (x / exp(1))^x
}

[[2]]
[1] 2

[[3]]
[1] 7

$end
[1] "Hello"
```

One of the possible ways of converting objects is to use the function `split()`. This returns a list of the split objects with separations according to the defined criteria.

```
> split(myframe, myframe$Continent)
$Asia
  City Area Pop. Continent Prox.Sea Language.Spoken
Tokyo Tokyo 2188 12.9      Asia Coastal           Japanese

$Europe
  City Area Pop. Continent Prox.Sea Language.Spoken
Berlin Berlin 892 3.4      Europe Inland           German
Paris Paris 105 2.1       Europe Inland           French

$'North America'
  City Area Pop. Continent Prox.Sea Language.Spoken
New York New York 1214 8.1 North America Coastal           English
```

In the above example, the data frame `myframe` is split into elements according to its column `Continent` and transformed into a list.

1.4.6 Programming in R

Functions

R has many programming capabilities, and allows creating powerful routines with functions, loops, conditions, packages and objects. As in the Stirling example, `args()` is used to receive a list of possible arguments for a specific function.

```
> args(data.frame) # list possible arguments and default values
function(..., row.names = NULL, check.rows = FALSE, check.names
         = TRUE, stringsAsFactors = default.stringsAsFactors())
NULL
```

This command provides a list of all arguments that can be used in the function, including the default settings for the optional ones, which have the form *optional argument = setting value*.

Below a simple function is presented, which returns the list $\{a \cdot \sin(x), a \cdot \cos(x)\}$. The arguments `a` and `x` are defined in round brackets. We can define functions with optional arguments that have default values, in this example, `a = 1`.

```
> myfun = function(x, a = 1){ # define function
+   r1 = a * sin(x)
+   r2 = a * cos(x)
+   list(r1, r2)
+ }
> myfun(pi / 2) # apply to pi / 2, a = default
[[1]]
[1] 1

[[2]]
[1] 6.123234e-17
```

Note that if no `return(result)` operator is given at the end of the function body, then the last created object will be returned.

Loops and conditions

The family of these operators is a powerful and useful tool. However, in order to perform well, they should be used wisely. Let us start with the ‘*if*’ condition.

```
> x = 1
> if(x == 2){print("x == 2")}
> if(x == 2){print("x == 2")}else{print("x != 2")}
[1] "x != 2"
```

The first programme is only an `if` condition, whereas the second is extended by the `else` command, which provides an alternative command in case the condition is not realised. More advanced, but more unusual in syntax, is the function `ifelse(boolean check, if-case, else-case)`. It is used mainly in advanced frameworks, to simplify code in which several `ifelse` constructions are embedded within each other.

Furthermore, `for` and `while` are very useful functions for creating loops, but are best avoided in case of large sample sizes and extensive computations, since they work very slowly. The difference between the functions is that `for` applies the computation for a defined range of integers and `while` carries out the computation until a certain condition is fulfilled. One may also use `repeat`, which will repeat the specified code until it reaches the command `break`. One must be careful to include a break rule or the loop will repeat infinitely.

```
> x = numeric(1)
> # for i from 1 to 10, the i-th element of x takes value i
> for(i in 1:10) x[i] = i
> x
[1] 1 2 3 4 5 6 7 8 9 10
> # as long as i < 21, set i-th element equal to i and increase i by 1
> i = 1
> while(i < 21){
+   x[i] = i
+   i = i + 1
+ }
> x
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
> # remove the first element of x, stop when x has length 1
> repeat{
+   x = x[-1]
+   if(length(x) == 1) break
+ }
> x
[1] 20
```

As an alternative to loops, R provides the functions `apply()`, `sapply()` (for multivariate case `mapply()`) and `lapply()`. In many cases, using these functions will improve the computational time significantly compared to the above loop functions. The function `apply()` applies a deterministic function to the rows or to the columns of a matrix. The first argument specifies the matrix, the second argument determines whether the third argument is applied to rows or columns, and the third argument specifies the function.

```
> A = matrix(1:24, 12, 2, byrow = TRUE)
> apply(A, 2, mean) # apply mean to columns separately
[1] 12 13
> apply(A, 2, sum) # column-wise sum
[1] 144 156
> apply(A, 1, mean) # row-wise mean
[1] 1.5 3.5 5.5 7.5 9.5 11.5 13.5 15.5 17.5 19.5 21.5 23.5
> apply(A, 1, sum) # row-wise sum
[1] 3 7 11 15 19 23 27 31 35 39 43 47
```

The functions `lapply()` and `sapply()` are of a more general form, since the function is applied to each element of the list object and returns a list; and `sapply` returns a numeric vector or a matrix, if appropriate. So if the class of the object is *matrix* or *numeric*, the `sapply` function is preferred, but this function takes longer,

as it applies `lapply` and converts the result afterwards. If a more general object, e.g. a list of objects, is used, the `lapply` function is more appropriate.

```
> A = matrix(1:24, 12, 2, byrow = TRUE)
> # apply function sin() to every element, return numeric vector
> sapply(A[1, ], sin)
[1] 0.8414710 0.9092974
> class(sapply(A[1:4, ], sin))
[1] "numeric"

> # apply function sin() to every element, return list
> lapply(A[1, ], sin)
[[1]]
[1] 0.841471

[[2]]
[1] 0.9092974
> class(lapply(A[1:4, ], sin))
[1] "list"
```

There is one more useful function, called `tapply()`, which applies a defined function to each cell of a ragged array. The latter is made from non-empty groups of values given by a unique combination of the levels of certain factors. Simply speaking, `tapply()` is used to break an array or vector into different subgroups before applying the function to each subgroup. In the example below, a matrix `A` is examined, which could contain the observations 1–12 of individuals from group 1, 2 or 3. Our intention is to calculate the mean for each group separately.

```
> g = c(rep(1, 4), rep(2, 4), rep(3, 4)) # create vector "group ID"
> A = cbind(1:12, g) # observations and group ID
> tapply(A[, 1], A[, 2], mean) # apply function per group
1 2 3
2.5 6.5 10.5
```

Finally, the `switch()` function may be seen as the highlight of R's built-in programming functions. The function `switch(i, expression1, expression2,...)` chooses the *i*-th expression in the given expression arguments. It works with numbers, but also with character chains to specify the expressions. This can be used to simplify code, e.g. by defining a function that can be called to perform different computations.

```
> rootsquare = function(x, type){ # define function for ^2 or ^(0.5)
+   switch (type, square = x * x, root = sqrt(x))
+ }
> rootsquare(10, "square") # apply "square" to argument 10
[1] 100
> rootsquare(10, 1) # first is equivalent to "square"
[1] 100
> rootsquare(10, "root") # apply "root" to argument 10
[1] 3.162278
> rootsquare(10, 2)
[1] 3.162278
> rootsquare(10, "ROOT") # apply "ROOT" (not defined)
[1] NULL
```

It is sometimes useful to compare the efficiency of two different commands in terms of the time they need to be computed, which can be done by the function `system.time()`. This function returns three values: *user* is the CPU time charged for the execution of the user instructions of R, i.e. the processing of the functions. The *system time* is the CPU time used by the system on behalf of the calling process. In sum, they give the total amount of CPU time. *Elapsed time* is the total real time passed for the user. Since CPU processes can be run simultaneously, there is no clear relation between total CPU time and elapsed time.

```
> x = c(1:500000)
> system.time(for(i in 1:200000) {x[i] = rnorm(1)}))
user system elapsed
0.892 0.033 0.925
> system.time(x <- rnorm(200000))
user system elapsed
0.017 0.000 0.017
```

Here the function `rnorm(x)` is used, which simulates from the normal distribution, see Sect. 4.3. Note that the hardwired `rnorm()` is faster than the `for` loop.

1.4.7 Date Types

R provides full access to current date and time values through the functions

```
> Sys.time() > date()
```

The function `as.Date()` is used to format data from another source to dates that R can work with. When reading in dates, it is important to specify the date format, i.e. the order and delimiter. A list of date formats that can be converted by R via the appropriate conversion specifications can be found in the help for `strptime`. One can also change the format of the dates in R.

```
> dates = c("23.05.1984", "2001/01/01", "May 3, 1256")
> # read dates specifying correct format
> dates1 = as.Date(dates[1], "%d.%m.%Y"); dates1
[1] "1984-05-23"
> dates2 = as.Date(dates[2], "%Y/%m/%d"); dates2
[1] "2001-01-01"
> dates3 = as.Date(dates[3], "%B %d,%Y"); dates3
[1] "1256-05-03"
> dates.a = c(dates1, dates2, dates3)
> format(dates.a, "%m.%Y") # delimiter "." and month/year only
[1] "05.1984" "01.2001" "05.1256"
```

Note that the function `as.Date` is not only applicable to *character strings*, *factors* and *logical NA*, but also to objects of types *POSIXlt* and *POSIXct*. The last two objects represent calendar dates and times, where *POSIXct* denotes the UTC timezone as a numeric vector and *POSIXlt* gives a list of vectors including seconds, minutes, hours, etc.

The difference between two dates is calculated by `difftime()`.

```
> dates.a = as.Date(c("1984/05/23", "2001/01/01", "1256/05/03"))
> difftime(Sys.time(), dates.a)
Time differences in days
[1] 11033.473 5332.473 276949.473
```

The functions `months()`, `weekdays()` and `quarters()` give the month, week-day and quarter of the specified date, respectively.

```
> dates.a = as.Date(c("1984/05/23", "2001/01/01", "1256/05/03"))
> months(dates.a)
[1] "May" "January" "May"
> weekdays(dates.a)
[1] "Wednesday" "Saturday" "Wednesday"
> quarters(dates.a)
[1] "Q2" "Q1" "Q2"
```

1.4.8 Reading and Writing Data from and to Files

For statisticians, software must be able to easily handle data without restrictions on its format, whether it is ‘human readable’ (such as `.csv`, `.txt`), in binary format (SPSS, STATA, Minitab, S-PLUS, SAS (export libs)) or from relational databases.

Writing data

There are some useful functions for writing data, e.g. the standard `write.table()`. Its often used options include `col.names` and `row.names`, which specify whether row or column names are written to the data file, as well as `sep`, which specifies the separator to be used between values.

```
> write.table(myframe, "mydata.txt")
> write.table(Orange, "example.txt",
+   col.names = FALSE, row.names = FALSE)
> write.table(Orange, "example2.txt", sep="\t")
```

The first command creates the file *mydata.txt* in the working directory of the data frame *myframe* from Sect. 1.4.4, the second specifies that the names for columns and rows are not defined, and the last one asks for *tab separation* between cells.

The functions `write.csv()` and `write.csv2()` are both used to create Excel-compatible files. They differ from `write.table()` only in the default definition of the decimal separator, where `write.csv()` uses ‘.’ as a decimal separator and ‘;’ as the separator between columns in the data. Function `write.csv2()` uses ‘,’ as decimal separator and ‘;’ as column separator.

```
> data = write.csv("file name") # decimal ".", column separator ","
> data = write.csv2("file name") # decimal ",", column separator ";"
```

Reading data

R supplies many built-in data sets. They can be found through the function `data()`, or, less efficiently, through `objects(package:datasets)`. In any case, we

can load the pre-built data sets library using `library("datasets")`, where the quotation marks are optional. Many packages bring their own data, so for many examples in this book, packages will be loaded in order to work with their included data sets. To check whether a data set is in a package, `data(package = "package name")` is used.

Moreover, we can import `.txt` files, with or without header.

```
> data = read.table("mydata.txt", header = TRUE)
```

The default option is `header = FALSE`, indicating that there is no description in the first row of each column. Information about the data frame is requested by the functions `names()`, which states the column names, and `str()`, which displays the structure of the data.

```
> names(data)
[1] "City"          "Area"          "Pop."          "Continent"
[5] "Prox.Sea"      "Language.Spoken"
> str(data)
'data.frame':  4 obs. of  6 variables:
 $ City      : Factor w/ 4 levels "Berlin","New York",...:1 2 3 4
 $ Area      : int  892 1214 105 2188
 $ Pop.      : num  3.4 8.1 2.1 12.9
 $ Continent : Factor w/ 3 levels "Asia","Europe",...:2 3 2 1
 $ Prox.Sea  : Factor w/ 2 levels "Coastal","Inland":2 1 2 1
 $ Language.Spoken: Factor w/ 4 levels "English","French",...:3 1 2 4
```

The function `head()` returns the first few rows of an object, which can be used to check whether there is a header. To do this, it is important to know that the first row is generally a header when it has one column less than the second row.

In some cases, the separation between the columns will not follow any of the standard formats. We can then use option `sep` to manually specify the column separator.

```
> data = read.table("file name", sep = "\t")
```

Here we specified manually that there is a tab character between the variables. Without a correctly specified separator, R may read all the lines as a single expression.

Missing values are represented by `NA` in R, but different programmes and authors use other symbols, which can be defined in the function. Suppose, for example, that the `NA` values were denoted by 'missing' by the creator of the dataset.

```
> data = read.table("file name", na.strings = "missing")
```

To import data in `.csv` (comma-separated list) format, e.g. from Microsoft Excel, the functions `read.csv()` or `read.csv2()` are used. They differ from each other in the same way as the functions `write.csv()` or `write.csv2()` discussed above.

To import or even write data in the formats of statistic software packages such as STATA or SPSS, the package `foreign` provides a number of additional functions. These functions are named `read.` plus the data file extension, e.g. `read.dta()` for STATA data.

To read data in the most general way from any file, the function `scan("file name")` is used. This function is more universal than `read.table()`, but not as simple to handle. It can be used to read columnar data or read data into a list.

It is possible to have the user choose interactively between several options by using the function `menu()`. This function shows a list of options from which the user can choose by entering the value or its index number, and gives as output the list rank. With the option `graphics = TRUE`, the list is shown in a separate window.

```
> menu(c("abc", "def"), title = "Enter value")
Enter value

1: abc
2: def

Selection: def
[1] 2
> menu(c("abc", "def"), graphics = TRUE, title = "Enter value")
```