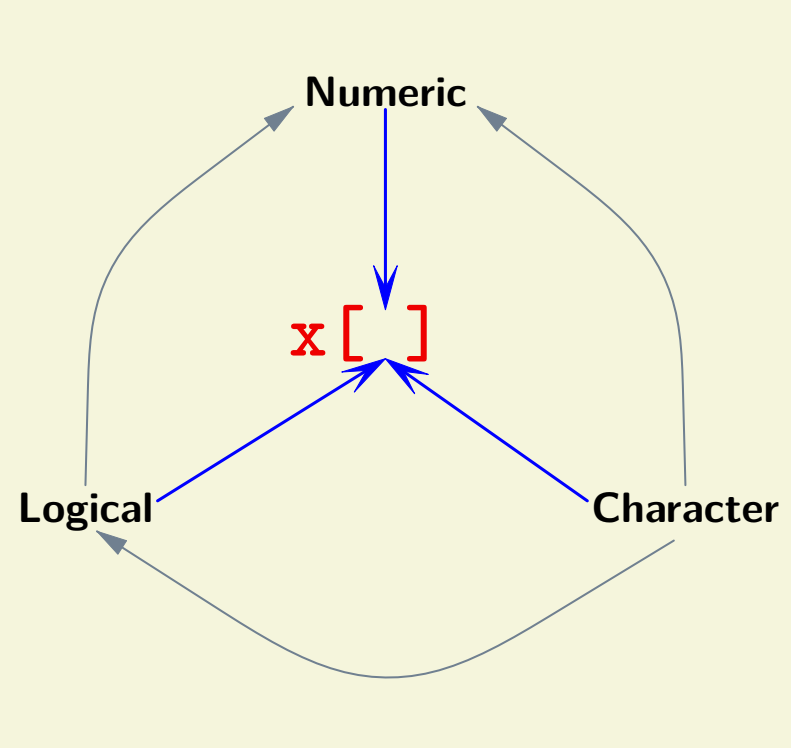
R Programming and Development From Basics to Advanced Topics

Emmanuel Paradis

R Programming and Development

From Basics to Advanced Topics

Emmanuel Paradis



R Programming and Development

From Basics to Advanced Topics

Emmanuel Paradis

ISEM, IRD, CNRS, Univ. Montpellier, France

This is publication ISEM 2022-262.

Contents

Preface

This book has its origins from several courses and workshops on R in which I participated as instructor. I firmly believe in the saying "*If you want to learn something, teach it.*" My background is in ecology and population biology, and as a student I had little interest in statistics or in computer science. Nevertheless, like other PhD candidates, I had to learn from these fields to analyse my field data. I cannot say that analysing data was an exciting experience at that time: quite often running a computer without it crashing after ten minutes was already a performance. After taking my position in Montpellier, R started to spread slowly but surely in the academic world, and it was clear to me that I had to learn it. So I taught it.

My early teaching experience convinced me to write *R for Beginners*, a document first published in French (as *R pour les débutants*, a 16-page document) in May 2000, then in a slightly longer version both in English and in French in October 2000 (31 pages), and in an extended version in August 2002 in these two languages again (58 pages). This third version was translated in Spanish and in Basque. An improved and again extended version (76 pages) version was published in September 2005, then translated in Chinese, in Romanian, and in Thai. Looking back at these documents helps to see the changes done by the R Core Team over the years. For example, one can read in the first version that an executable version of R was available for the m68k-based computer architecture running the NextStep operating system.

Undoubtedly, there has been a lot of things changed in the past 22 years. R has extended in many ways: scope, relevance, efficiency, versatility, popularity, and certainly others. In the meantime, I wrote two books on using R in two specialised fields of evolutionary biology (phylogenetics and population genomics). The present book can be seen as the continuation—or complement—of *R for Beginners* after exploring data analysis and statistical inference with R in my own discipline.

The aim of *R Programming and Development* is to explore the basics of R in order to improve both the understanding of these basics and the practice of using R. The issues addressed in these pages have been selected with the aim to be useful in a wide range of applications with R. The book is not intended to beginners who look for information on how and where to start with R: there is now a vast literature on this question, and it seems *R for Beginners* is still relevant for this. Therefore, the readers will not find here information on how

to install R or similar things. However, I believe that many novices with basic knowledge in R will be able to find their way relatively easily in the pages below because I tried to explain the concepts from their basics.

A consequence of the main objective of *R Programming and Development* is that it focuses almost exclusively on the R recommended packages (such as parallel in the chapter about high-performance computing). Chapter 1 is an introduction with a brief overview on two general topics: the importance of open source software in data analysis, and user interfaces. Chapters 2–5 treat subjects likely to interest a wide range of users and programmers: data structure and manipulation, programming functions, and some specific data types such as character strings, dates, and times. Chapters 6 on debugging and 7 on code optimisation are for programmers and developers who want to improve their work. Chapter 8 tackles an important topic: interfacing R and C using the tools available in a default installation of R. Chapter 9 is an overview of using R with high-performance computing (HPC) hardware. Finally, two appendices look into the details of binary coding of numbers in computers and the problem of numerical precision when computing sums. Chapters 2–9 end with exercises that invite the reader to further explore the issues exposed there. The solutions will be published later on-line.

Most of the materials in these pages come from my own experience in developing packages for R as well as using it for my own research. I also benefited (and enjoyed) teaching R in several advanced courses, and I am grateful to the organisers of these. Special thanks to Perine Sanglier for starting the organisation of a permanent training on R in IRD, and to Soledad De Esteban-Trivigno for organising the course "Phylogenetic analysis using R" during ten years.

E.P.
November 2022

1

Introduction

1.1 Data Analysis, Open Source Software, and R

R was created in a period of renewed and intense software development: the rise of Internet during the 1990's combined with the increasing computing power of widespread and affordable hardware (e.g., 32-bit processors) stimulated new ideas, innovative projects, so that computers started to be used in many novel applications. At the end of that decade, R was the only free computer program for statistical analyses to be competitive in a field with many commercial programs.

Free software received increasing acceptance in the late twentieth and early twenty-first centuries. Linux, LaTeX, and OpenOffice/LibreOffice are a few examples of generalist, free software that have gained wide popularity in the last two decades with very significant and broad impacts. Successful free software projects share three features:

- open source;
- scalability;
- versatility.

These three features take a special importance when considering statistical methods and data analyses.

Open source leads to free software under the condition that they are properly distributed, and Internet has been critical for this. Open source is also fundamental in stimulating development and collaborations (Internet has also been critical for these). For scientific data analyses, open source has revolutionised practices, especially for publications. Many journals now require that software, scripts, and/or computer code be publicly released before acceptance, or publication, of a scientific article. Re-use of previously released or published code has stimulated a lot of new methodological developments.

Scalability is the ability of software to run efficiently on a wide range of hardware. Clearly, being able to run the same program on a laptop or on a supercomputer has tremendous advantage. Not many software have achieved this: Linux and some programming languages (e.g., C, C++, Fortran, Python) are remarkable examples.

Versatility is another critical feature that is likely to help some software to be popular: if a program can be used to do many different things, then it will be more likely to attract a large number of users and developers.

R checks all the three above features: it is free and open source; it can be used on very different machines running different operating systems (OSs); and a very wide range of statistical and computational methods have already been implemented in R. A lot has been said or written to explain the success of R: I think it simply arrived at the right moment to do the right things.

1.2 What Is (and What Is Not) in This Book

This book is for readers who wish to acquire a "less-than-superficial" knowledge and understanding of R. Prior knowledge or experience with R will obviously be useful but not a critical requirement since the basic concepts are detailed before going deeper. For a beginner's level introduction to R, the reader is referred to *R for Beginners* (see Preface) or browse the numerous resources on Internet. No knowledge on statistics or data analysis is required.

The following chapters explore most basic issues related to the use of R: data structure and manipulation, code optimisation, debugging, R–C interfaces, and high-performance computing. The main objective is to acquire a level of expertise sufficient to *develop* code and programs in R.

Almost all R resources used in this book come from the recommended R-packages, so that these are likely to be useful for a very wide range of readers, and no problem of installation should be encountered. On the other hand, the development of R code for the specific aim of writing, releasing, and maintaining a package is not treated—although the authors of R-packages will (I hope) find some useful information in these pages.

There are a few important topics that I chose to not treat because I wanted to keep this book within a reasonable length, and also because there are already very valuable resources on these topics:

- Reading and writing files. This is a central topic but a good understanding of data structures in R (see Chap. 2) is likely to help handling files in general. Besides, there are numerous specialised R-packages to help read a very diverse range of data file formats, and examining all of them would certainly require a full textbook.
- Graphics is another central topic with basic resources provided by several recommended packages (graphics, grDevices, and grid). The interested

Table 1.1. Some user interfaces that run on common OSs to use R interactively. (ESS stands for "Emacs Speaks Statistics".)

Name	Comments
ESS/Emacs	Emacs has many integrated tools for editing and software development
Jupyter	Appropriate for teaching
RKWard	User-friendly
RStudio	User-friendly; customer support
Visual Studio Code	Easy to use

reader is referred to the classic *R Graphics* by Paul Murrell (now in its third edition).

- The development, release, and maintenance of a package written in R has become an important output of many scientific projects. The manual *Writing R Extensions* is the central resource for package developers and should be consulted regularly since it is regularly updated by the R Core Team.[1]
- Hadley Wickham and his collaborators have released a number of popular packages that are presented in his books and additional resources can be found on his web site.[2]
- Rcpp would have been introduced in Chapter 8 if Dirk Eddelbuettel would have not written an extensive documentation with this package, several open-access papers, and several books.[3]

1.3 User Interfaces

There are two basic ways to use R: either interactively, or by running a program written with R code. In an interactive session, the user starts R and then executes commands to perform their analyses (by typing them or using the menus of a graphical user-interface). In this situation, the user is likely to choose the analyses depending on the progress of the session (as typical in an interactive work). On the other hand, with an R program the commands are executed successively without intervention from the user. The separation between these two ways is not clear-cut, however; for instance, an R program can be run during an interactive session (e.g., with the function `source`).

During an interactive session, an additional software can be used to interact more easily with R. Table 1.1 lists a selection of these interfaces: using one or another is mainly a question of personal preference (there are probably others

[1] `https://cran.r-project.org/manuals.html`
[2] `https://hadley.nz/`
[3] `https://dirk.eddelbuettel.com/`

that can be found on the Internet). However, a few points might be considered when choosing one such program:

- The interface must handle (open, edit, and save) R code in simple text in order to be compatible with other interfaces. Indeed, if you want to be able to choose your interface, this should not be at the expense of compatibility with your colleagues.
- The interface is itself a program and must run on the computer in addition to R, so it must be as economic as possible in terms of resource use for your machine.

Additionally to the programs listed in Table 1.1, there are several OS-specific user-interfaces delivered with R for Windows and for MacOS X: they include several basic features such as drop-down menus to access some common commands (e.g., change the working directory).

There are a number of specific commands which are always run non-interactively that we will see in the some chapters of this book. They are run from the command line of the computer with `R CMD <command>`, where the possible commands are listed in Table 1.2. Other uses of R non-interactively are detailed below when discussing high performance computing (Chap. 9).

1.4 Conventions

In the text, R input commands and output results are printed inside boxes with the standard R prompt symbol (>) before the commands:

```
> 1 + 2
[1] 3
```

Occasionally, commands are simply printed without the prompt symbol:

```
ls()
```

C code (always in a file) are printed on a light blue background:

```
#include <R.h>
```

Commands to be typed on the system (e.g., in a command shell) are printed on a light grey background:

```
R CMD BATCH script.R
```

Contents of file other than C source code are also shown on a light grey background.

R objects are written in monospace font (e.g., `x`, `DF`), package names in sans serif font (e.g., parallel), and file names are written within single quotes (e.g., 'data.txt').

Table 1.2. Commands when R is used non-interactively with `R CMD <command>`.

`<command>`	Effect
`BATCH`	Run R in batch mode
`COMPILE`	Compile files for use with R[a]
`SHLIB`	Build shared library for dynamic loading[a]
`INSTALL`	Install packages
`REMOVE`	Remove packages
`build`	Build packages
`check`	Check packages
`LINK`	Front-end for creating executable programs[a]
`Rprof`	Post-process R profiling files
`Rdconv`	Convert Rd format to various other formats
`Rd2pdf`	Convert Rd format to PDF
`Rd2txt`	Convert Rd format to pretty text
`Stangle`	Extract S/R code from Sweave documentation
`Sweave`	Process Sweave documentation
`Rdiff`	Diff R output ignoring headers
`config`	Obtain configuration information about R
`javareconf`	Update the Java configuration variables
`rtags`	Create Emacs-style tag files from C, R, and Rd files

[a]See Chap. 8.

Three kinds of quotation marks (often simply called "quotes") are used in the R syntax: straight double or single quote, and backtick (Table 1.3). When printing R code, the straight single quotes are typeset as right (closing) single quotation marks, whereas backticks are typeset as left (opening) single quotation marks. To avoid confusion, the context is explicit about the use of these single quotes.

Table 1.3. The three types of quote used in R code. The second column shows the aspect of these symbols on most keyboards (their locations vary a lot with the keyboard layout).

Type of quote	On keyboards	R code in this book
Straight double quote	" or “	"
Straight single quote	' or ‘	’
Backtick	`	‘

2

Data Structures in R

2.1 General Considerations

Data are everywhere in our modern world. Phones, computers, sensors, cameras, and other devices, generate enormous quantities of data. Yet, when it comes to analyse data, whatever their origins, a very general rule is to arrange them in a rectangular table with rows representing the *observations* and columns representing the *variables*.[1] The observations may be persons, animal or plant species, atoms, planets, and so on; the variables are measures on these observations, for instance, body mass, energy, temperature, genome size, diameter, and so on. Clearly, such a data table is asymmetric: it makes no sense to consider a variable as an observation, or an observation as a variable.[2]

Another general rule about data structure is that a variable can be considered as either *quantitative* or *qualitative*. In practice, this dichotomy is not strict: a quantitative variable can be considered as discrete (e.g., number of individuals in a group) or continuous (e.g., body length). Qualitative variables can be also of many different subtypes: the number of categories (or classes, or levels) may be fixed or not; the categories may be ordered (e.g., the choices in a customer satisfaction survey) or not. A quantitative variable may be transformed into a qualitative one by defining intervals, for instance, age classes, or concentration classes of a chemical in blood.

These two principles, asymmetric data table and quantitative *vs.* qualitative variables, are so general that almost all data analysis computer programs implement them in one way or another; and R is no exception as we will see in this chapter. It should be pointed out here that the vocabulary used for

[1] See Fig. 2.3 (p. 14) for a view of such a data table.

[2] There are some exceptions: in ecology, a table with counts of species by sites is symmetric and can be transposed: we may consider species as observations which are characterized by their distributions over sites, or the sites as observations characterized by their species compositions.

these concepts vary a lot with the software: in the case of R, the basic type of a variable is called the *mode*, and the data structures are called *objects*. In practice, things are slightly more complicated and we will need to understand both modes and objects to have a general understanding of data structure in R.

Before detailing the R specifics on data structure, it is interesting to note that the idea of a *variable* is so important in scientific research that this word is used with four distinct meanings:

- A variable is a mathematical entity which can take several values, typically in a mathematical function or an equation. For instance, in the linear function $f(x) = 2x + 3$, x is a variable which can take values between $-\infty$ and $+\infty$; however, there is here no implication on the distribution of x.
- A variable is a quantity (or quality) measured or observed which can take different values. For instance, the temperature on the surface of our planet is a variable. On the other hand, the temperature at which water boils under atmospheric pressure is a *constant*.
- A *random* variable is a quantity following a specific distribution. Random variables may be real (e.g., the number of heads from several coin tosses) or abstract (e.g., a binomial variable).
- A variable is a part of the memory of a computer used to store some information which may change during the execution of a program. For instance, the temperatures measured every week during one year constitute a set of 52 variables stored on a computer.

2.2 Modes

All computer languages, systems, or applications have their own ways to code basic data. In R, this is called the *mode*. Every data in R are characterised by their mode which is the basic type of the data stored. The concept of mode in R is actually broader and applies also to objects which are not data. The next section details how modes are defined for data in R, and the following one gives an overview of other uses of mode in R.

2.2.1 Data Modes

The mode is usually defined implicitly when some data are input. For instance, if we create two vectors named `x` and `z`:

```
> x <- 1:5
> z <- c("Homo", "Pan", "Gorilla")
```

Their respective mode can be printed with the function `mode`:

```
> mode(x)
[1] "numeric"
> mode(z)
[1] "character"
```

In simple objects such as `x` or `y`, all elements are of the same mode. At this point it is useful to introduce another characteristic of data in R: the length which is printed with the function `length`:

```
> length(x)
[1] 5
> length(z)
[1] 3
```

The length is the number of elements in an object. Together with the mode, these two characteristics represent the *intrinsic attributes* of the data in R: they are mandatory and therefore always set (Fig. 2.1). We will see more about attributes in Section 2.3.2.

Before going further about modes, it may useful to note a remarkable feature about R's operations: they return R data which have, obviously, a mode and a length:

```
> mode(mode(x))
[1] "character"
> length(mode(x))
[1] 1
> mode(length(x))
[1] "numeric"
> length(length(x))
[1] 1
```

We will see more about objects returned by R's functions in the next chapter.

There are five main modes for data in R: numeric, character, logical, complex, and raw (Table 2.1). In practice, it is mainly the first one which is used to store data. Indeed, the mode numeric is used to store both quantitative variables (as numeric vectors) and qualitative ones (as factors). We will see these details in Section 2.3; for the moment we only need to know that all sorts of numbers (reals, integers, complex) are stored as mode numeric in R.

The mode character is somewhat special in R: each element of a vector of this mode is a character string, not a single character (Fig. 2.1). The reason for this is because, in most applications, these vectors are used as labels or identifiers of data stored with the mode numeric. Of course, there are exceptions: researchers working in linguistics undoubtedly manipulate data as character strings. But the fact that vectors of mode character store character strings instead of single characters makes data manipulation particularly easy

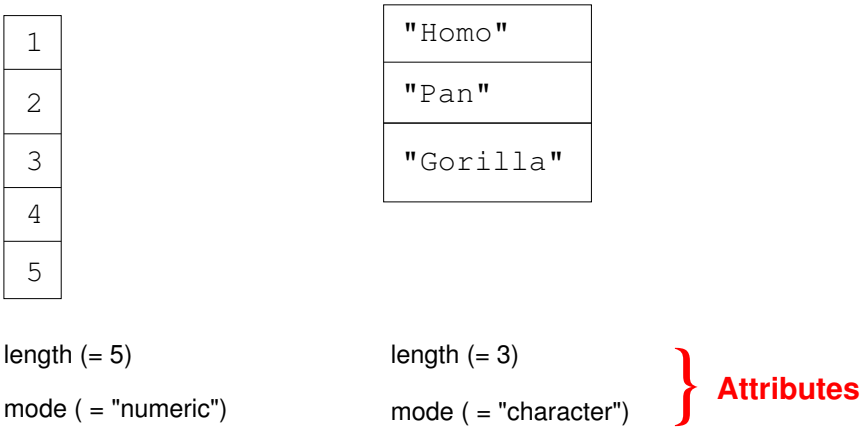


Fig. 2.1. A representation of two vectors in R: a numeric vector (left) and a character vector (right). Their respective intrinsic attributes are shown below them.

and powerful (see Chap. 4). Furthermore, text and characters have some specificities that will be examined later (Sect. 4.3).

The mode logical codes values which are either `TRUE` or `FALSE`. They may be used to store some data (which are known as binary or Boolean variables), but their usefulness resides in the fact that are powerful tools to manipulate data (see Sect. 4.2).

The mode complex can hardly code for (real) data, but they are very important in computing where operations potentially output complex numbers (e.g., Fourier transform, eigendecomposition).

Finally, the mode raw stores single bytes (e.g., integer values between 0 and 255; see p. 135). Like for the mode complex, it is rarely used to store some data, but in some specific applications this can be useful.

To summarise, among the five main data modes in R (Table 2.1), the mode numeric is the most frequently used one to store data. The mode character is used to store identifiers and labels, whereas the mode logical is used for data manipulation. The modes complex and raw are used in some specific applications.

Table 2.1. The five main (data) modes in R.

Mode	Data	Comments
numeric	numbers	Can be integers or real numbers
character	text strings	Strings may be of different lengths
logical	Boolean	Stored as integers
complex	complex numbers	Stored as two numeric values
raw	bytes	Missing value are not allowed

Table 2.2. Other modes in R.

Mode	Description
function	The basic engines in R[a]
expression	R command(s) after parsing text[b]
environment	A subset of memory containing R objects with distinct symbol names[a]
formula	Relations (models) among variables[c]
list	A list of object(s)

[a]See Chap. 3.
[b]See Sect. 5.1.
[c]See Sect. 5.2.

2.2.2 Other Modes

Every object in R has a mode. For instance, a function has mode `"function"`; an environment has mode `"environment"`:

```
> mode(environment)
[1] "function"
> mode(environment())
[1] "environment"
```

Some of these objects are used internally by R and knowing their details is rarely useful to most users, even if they develop packages. Nevertheless, there are a few modes outside of data objects which are good to know because they can be manipulated in the usual way in R (Table 2.2). For instance, a series of formulas can be stored in a list to fit a series of models. This emphasizes the need to have a good understanding of what is a list since this type of object can store any objects as we will see in the next section.

2.2.3 `NULL`

The object `NULL` deserves a special mention because it is often used in practice, for example to delete an element in a data frame or a list. However, `NULL` has a double usage in R. First, it can be used to create an object with its name but no other feature:

```
> y <- NULL
> mode(y)
[1] "NULL"
> length(y)
[1] 0
```

Even though `y` has length zero and mode `NULL`, it can be combined with an existing object, which is useful in operations within a loop where different

values have to be combined successively:

```
> x <- 1
> y <- c(y, x)
> y
[1] 1
> mode(y)
[1] "numeric"
> length(y)
[1] 1
```

The second usage of `NULL` is to delete elements in a list: this is often used to delete columns in data frames. However, it is also sometimes useful to have an empty element in a list: in that case `NULL` should be included in a list before being assigned into the list:

```
> z <- list(a = 1:2, b = 3)
> z
$a
[1] 1 2

$b
[1] 3

> z[1] <- list(NULL)
> z
$a
NULL

$b
[1] 3
```

Note that the name (`"a"`) is not deleted. If instead `NULL` is not included in `list()`, the element is deleted from the list:

```
> z[1] <- NULL
> z
$b
[1] 3
```

The commands `z[[1]] <- NULL` and `z$a <- NULL` have the same effect than the last one.

2.3 Data Structures

We now have all the ingredients to see how R builds complicated data structures from the simple objects. We first examine the five main data structures

Table 2.3. The five main data structures in R.

R object	Data stored
vector	quantitative variable (or text)
factor	qualitative variable
matrix	matrix
data frame	table of vector(s) and/or factor(s)
list	vector of objects

in R, then see more details about the importance of attributes.

2.3.1 The Five Main Data Structures in R

Numeric vectors are our starting point: they are used to code quantitative variables (Table 2.3). A numeric vector can be viewed as similar to a column in a data matrix which stores a quantitative variable.

The second type of data is the *factor* and is used to code qualitative variables. A factor is actually a numeric vector with only integer values, and with each distinct value (1, 2, ...) coding for a specific category. In order to not confuse a factor with a quantitative variable, R needs an additional information which is provided by the attribute "levels", a vector of mode character giving names (or labels) to each category (Fig. 2.2).

Now that we know how to code both quantitative and qualitative variables, the next step is to combine them to build a data table. Before doing that, we have to see the object named *matrix*. Contrary to what is suggested by its name, a matrix in R is not a combination of different variables arranged along its columns, but a vector arranged as a rectangle. So, all elements of a matrix are of the same mode. The matrix is actually a special case of another data structure in R: the *array*; the difference is that a matrix has two dimensions (its rows and columns), whereas an array can have any number of dimensions. Below is an example with three objects: `x1` is a matrix with four rows and two columns, `x2` is a matrix with two rows and four columns, and `x3` is an array with three dimensions; however, all three object have the same data. We first create these objects as three identical vectors with the values 1, ... 8:

```
> x1 <- x2 <- x3 <- 1:8
```

We then give them dimensions as explained above:

```
> dim(x1) <- c(4, 2)
> dim(x2) <- c(2, 4)
> dim(x3) <- c(2, 2, 2)
> x1
     [,1] [,2]
[1,]    1    5
```

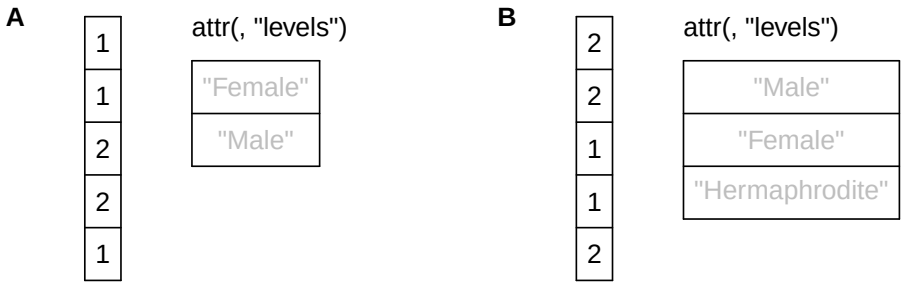


Fig. 2.2. Two examples of the same data, a sample of 5 individuals with 3 ♀ and 2 ♂, coded in two different factors. (A) A factor with two levels where the first level is 'Female' and the second one is 'Male'. (B) A factor with three levels: 'Male', 'Female", and 'Hermaphrodite' (the last one is not observed in the data).

```
[2,]    2    6
[3,]    3    7
[4,]    4    8
> x2
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
> x3
, , 1

     [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8
```

We can check that the data stored by these three objects are still the same:

```
> identical(as.vector(x1), as.vector(x2))
[1] TRUE
> identical(as.vector(x1), as.vector(x3))
[1] TRUE
```

We note that for all arrays (including matrices), the following always returns `TRUE`:

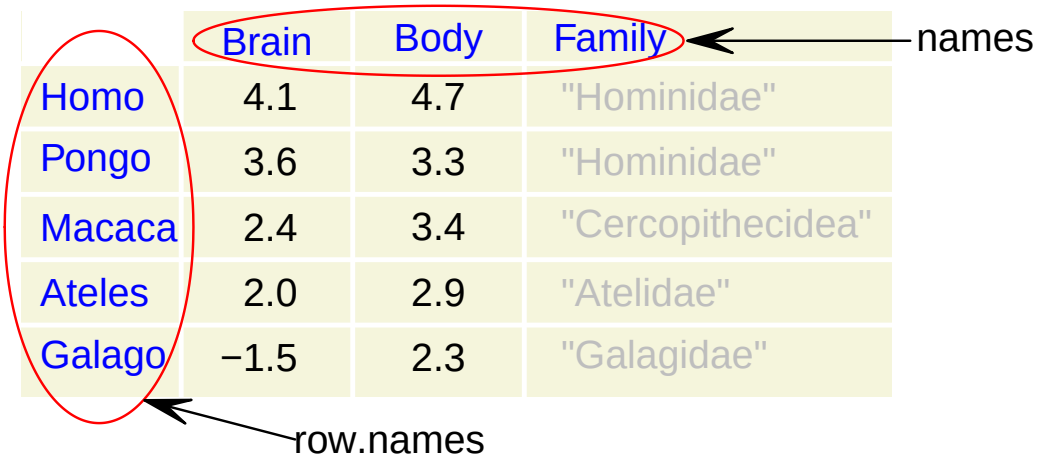


Fig. 2.3. A data frame with five rows and three variables.

```
prod(dim(x)) == length(x)
```

We have seen that vectors and factors code for quantitative and qualitative variables, respectively. We can now combine them to create *data frames*, the R data structure coding for a data table. To fix ideas, let us take three vectors with five values each (Fig. 2.3). The data may be read from a file in a specific format; to simplify this step, they are simply input at the keyboard:

```
> x <- c(4.1, 3.6, 2.4, 2, -1.5)
> y <- c(4.7, 3.3, 3.4, 2.9, 2.3)
> z <- c("Hominidae", "Hominidae", "Cercopithecidea", "Atelidae"
    , "Galagidae")
> names(x) <- c("Homo", "Pongo", "Macaca", "Ateles", "Galago")
> data.primates <- data.frame(Brain = x, Body = y, Family = z)
```

The three columns do not have all the same mode:

```
> sapply(data.primates, mode)
     Brain       Body     Family
 "numeric"   "numeric" "character"
```

Note that the labels associated with the rows and the columns are stored in their respective attributes:

```
> row.names(data.primates)
[1] "Homo"   "Pongo"  "Macaca" "Ateles" "Galago"
> names(data.primates)
[1] "Brain"  "Body"   "Family"
```

We notice that these two attributes are vectors of mode character.

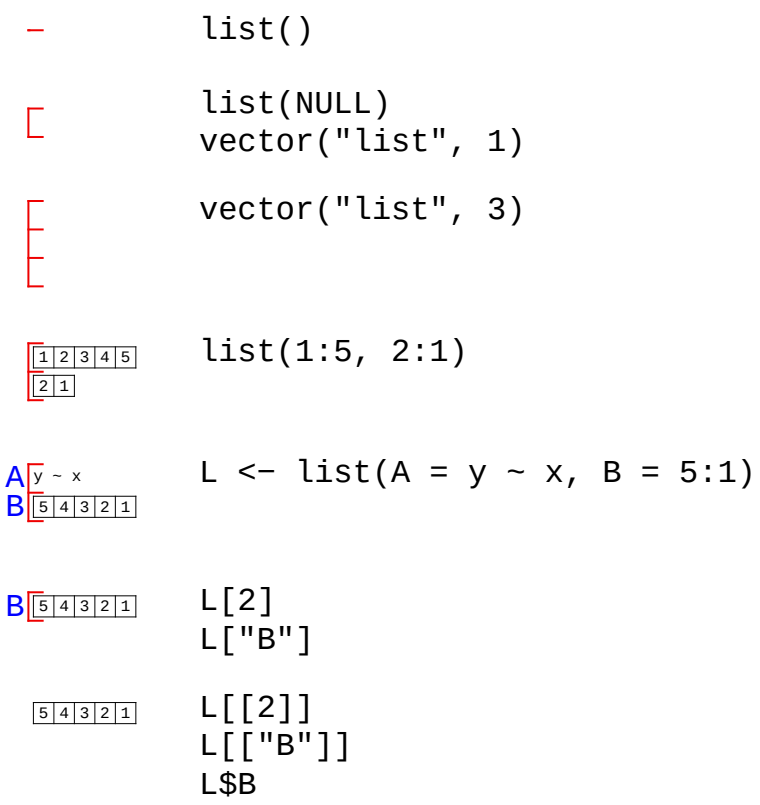


Fig. 2.4. Several examples of lists with the R code to create them (see text for the actual outputs). The list 'skeleton' is in red; names are in blue (see Fig. 2.3).

The final step to build data structures is to relax the constraint of equal length and make possible to combine all types of objects: this is possible with a *list*. In fact, a data frame is stored as a list:

```
> mode(data.primates)
[1] "list"
> is.list(data.primates)
[1] TRUE
```

Figure 2.4 represents schematically several lists to illustrate the basic features of this data structure; the R inputs and outputs to produce these examples are:

```
> list()
list()
> list(NULL)
[[1]]
NULL
```

```
> vector("list", 1)
[[1]]
NULL

> vector("list", 3)
[[1]]
NULL

[[2]]
NULL

[[3]]
NULL

> list(1:5, 2:1)
[[1]]
[1] 1 2 3 4 5

[[2]]
[1] 2 1

> L <- list(A = y ~ x, B = 5:1)
> L[2]
$B
[1] 5 4 3 2 1

> L["B"]
$B
[1] 5 4 3 2 1

> L[[2]]
[1] 5 4 3 2 1
> L[["B"]]
[1] 5 4 3 2 1
> L$B
[1] 5 4 3 2 1
```

These small examples show a few things about lists that are useful to keep in mind:

- Lists behave like vectors: they have a length, can be indexed with the `[` operator, and can have (optional) names.
- The two operators `[` and `[[` behave differently: the first one indexes the list and thus returns a list; the second one *extracts* an element of the

Table 2.4. Common attributes. Mandatory attributes are marked with *; the others are optional.

Object	Attribute(s)
vector	names
factor	names, levels*
matrix, array	dim*, dimnames
data frame	row.names*, names*
list	names

list and can return any type of object. Thus, `[` can be given a vector with several values whereas `[[` accepts only a single value (i.e., `L[[1:2]]` gives an error).

- Since a data frame is stored as a list, its columns are the elements of the list, so the labels of the variables are given by the names (see above).

Note the difference between the `[[` and `$` operators when extracting with a variable name—which both have the same side effect—since the former can accept a vector of mode character storing the name of the list element:

```
> v <- "B"
> L[[v]] # same output than with L$B
```

2.3.2 Attributes

We can now offer a (almost) complete description of an R data object [2]:

1. a character string naming the object (a symbol),
2. a vector or a list,
3. a list of attributes.

The second element of this description can be empty (`NULL`). Consequently, the third element, which is itself a list, can also be `NULL`. To be complete, we should add a short internal description of the object which is used only by R.[3] Table 2.4 lists the attributes commonly used in practice.

To summarise, with R all data are in vectors, either standard vectors (also called "atomic vectors" in R), or lists ("generic vectors"). Attributes, which are themselves vectors, give additional information so that R knows how to treat the data in the appropriate way (Fig. 2.5). With the exception of the length and the mode, these attributes are optional.

[3] See the *R Internals* manual delivered with R or on CRAN's web site.

- **vector**
  - → If vector of integers + attributes 'class' (1) and 'levels' (2) ⇒ **factor**
  - → If attribute 'dim' ⇒ **matrix** (3)
- **list** (vector of objects)
  - → If list of vectors and/or of factors all of same length + attributes 'class' (4), 'names' (2) and 'row.names' (2) ⇒ **data frame**

(1) class = `"factor"`
(2) vector of mode character
(3) if `length(dim)` > 2 ⇒ **array**
(4) class = `"data.frame"`

Fig. 2.5. Overview of R data objects.

2.4 Exercises

1. Give examples, relevant to your field, of quantitative and qualitative variables. Search the answer to this question to some of your colleagues who work in different fields and compare their answers with yours.

2. Create a vector with the values 1, 2, and 3. Transform this vector as a factor and store the result in a different object. Compare the characteristics of these two objects and explain the usefulness of the additional attributes attached to the factor.

3. What are the mode and length of the results returned by the functions `mode` and `length`?

4. Look at the example at the beginning of Section 2.2.2: explain why the parentheses changed the result returned by the function `mode`. (Hint: you may have a look at the next chapter for help.)

5. What is the difference between data read from files and data input from the keyboard?

6. Look at the help page of the function `attributes`. Try this function on the objects created in the small examples above. Compare this function with the function `attr`.

7. Explain, as simply as possible, the difference between these two operators in R: `[[` and `$`.

8. Explain, using your knowledge (not only about R), why single characters are not commonly used as data in R.

9. Create a matrix, say `X`, with three rows, three columns, and nine values of your choice so that they are all distinct. Execute the command `X[9]` and explain its result. What other command could give the same result?

10. Create a list with the command `L <- list(a = NULL)`. Compare the outputs from the commands `L$a` and `L$b`. Do the same with the commands `L[["a"]]` and `L[["b"]]`.

3

Programming R Functions

Functions are the working engines of R: all commands and operations are done with functions. Even the simple operators (`+`, `-`, `*`, `/`, ...) are functions; for example, here are two ways to use the `+` operator like a usual R function, either with the function `get` or with the backtick operator:

```
> get("+")(1, 2)
[1] 3
> `+`(1, 2)
[1] 3
```

In this chapter, we will see some details about using and programming R functions. Many of these details are not really useful for a daily use of R, but knowing them is very useful for advanced users and developers. We will first see how R manages objects in memory before examining how function arguments are defined and used. The subsequent sections look at some important aspects of function programming: return values, recursive functions, and generic functions.

3.1 Environments

This section is concerned with some aspects of how R manages objects in memory. Although this is an important point when using or programming functions, it is also important for other aspects of how R works. Thus, some of the discussions here are general and go beyond the topic of this chapter.

An environment can be seen as a portion of the memory of the computer where objects with distinct names (the symbols) are stored. A crucial feature of environments in R is that they have a *parent*: an environment where R will look for objects if they are not found in the current environment.

Because assignment with the `<-` operator is done in the current environment, we have to use the function `assign` to create an object inside another

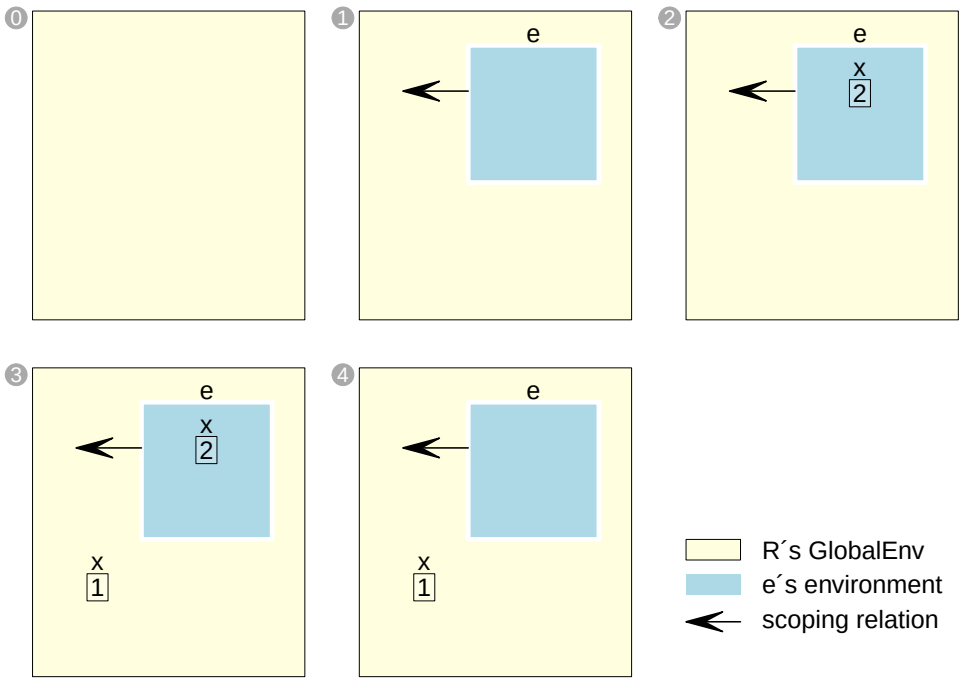


Fig. 3.1. Representation of the objects in memory when executing the example in the text. 0 There is initially no object in memory. 1 The environment `e` is created. 2 The vector `x` is created inside `e`. 3 The vector `x` is created in the global global environment. 4 `x` inside `e` is deleted.

environment. From Section 2.2.2, we remember that environments are objects in R, so we can create and manipulate them with the appropriate functions:

```
> e <- new.env()
> assign("x", 2, envir = e)
> x <- 1 # same than assign("x", 1)
```

With these three commands, we have created an environment `e`, then a vector `x` inside `e` using the function `assign`, and a vector `x` using the usual `<-` operator (Fig. 3.1). We have two objects with the same name, `x`, but they are in distinct environments:

```
> ls.str()
e : <environment: 0x27d7648>
x :  num 1
> get("x", envir = e)
[1] 2
> get("x") # same than x
[1] 1
```

Now let us see the mechanism of enclosing frame into action. We first

delete `x` inside `e` (the one with the value 2), and then try to print this value from this same environment as we did above:

```
> rm(x, envir = e)
> get("x", envir = e)
[1] 1
```

R has found the object `x` stored in the global environment. The reason for this is that the parent environment of `e` is precisely the global environment:

```
> parent.env(e)
<environment: R_GlobalEnv>
```

And what would happen if the object is not in the global environment? R would then follow the *search path* which can be displayed with the function `search`, for instance from a newly started R session:

```
> search()
[1] ".GlobalEnv"        "package:stats"     "package:graphics"
[4] "package:grDevices" "package:utils"     "package:datasets"
[7] "package:methods"   "Autoloads"         "package:base"
```

However, this search path applies only for the objects manipulated by the user. Things are different for objects manipulated inside a package. When a package (say `ape`) is loaded into memory, three environments are created:

- `namespace:ape` containing all objects created by `ape`'s R code;
- `package:ape` containing the objects exported by `ape`;
- an environment containing the objects imported by `ape`.

The search path for `ape`'s R code is:

1. `namespace:ape`
2. `package:ape`
3. `ape`'s imports
4. `package:base`
5. the normal search path

3.2 Arguments

The arguments represent the main way to give information to a function. They are defined when the function is created, although they are optional. Here is an example of a simple function with zero argument:

```
> f <- function() cat("Hello World!\n")
> f()
Hello World!
```

We note that the parentheses are needed to execute the function even if there is no argument. A function can have:

- no argument;
- a fixed number of argument(s);
- an undefined number of arguments;
- some (or all) arguments defined with a default value;
- some arguments may be left missing.

Arguments have names which represent objects within the environment of the function. Consider these two commands:

```
hist(x = rnorm(1000))
hist(x <- rnorm(1000))
```

Both commands are valid and will produce similar plots (although not identical), but they will have different effects with respect to the object `x`: in the first command `x` is created in the environment of the function `hist` and then lost once the histogram is drawn; whereas in the second command, `x` is created and stored in the global environment of R. Note also that the first command works without error because the function `hist` has an argument called `x`:

```
> args(hist)
function (x, ...)
NULL
```

It is more typical to use of this function with something like:

```
y <- rnorm(1000)
hist(x = y)
```

When `hist()` is called (or any function), an environment is created which includes the object(s) passed as argument(s) together with the object(s) created within the function. In the last example, `y` is a vector in the global environment (a.k.a. the workspace); on the other hand, `x` is in the environment of the function: as long as `x` is not modified by the code within `hist()`, both objects are identical.

3.2.1 Matching Arguments in Function Calls

R has two ways to match the objects or values passed as arguments with those in the function definition: by position or by name. Both ways can be used

together. For instance, the four commands below have exactly the same effect ("lightgray" is the default value of col):

```
hist(y)
hist(x = y)
hist(col = "lightgray", x = y)
hist(y, col = "lightgray")
```

Matching by position is slightly more efficient than matching by names, but a significant difference in performance can be noticed only for functions with many arguments (several tens) and if they are called a larger number of times (several thousands) .

We will see below that hist is a generic function and that it calls, most of the times, its default method, hist.default:

```
> args(hist.default)
function (x, breaks = "Sturges", freq = NULL, probability = !
    freq,
    include.lowest = TRUE, right = TRUE, density = NULL, angle =
        45,
    col = "lightgray", border = NULL, main = paste("Histogram of
        "),
        xname), xlim = range(breaks), ylim = NULL, xlab = xname,
    ylab, axes = TRUE, plot = TRUE, labels = FALSE, nclass =
        NULL,
    warn.unused = TRUE, ...)
NULL
```

This is a typical example of an R function: only the first argument is mandatory. In many functions, the first argument specifies the data to be analysed and is commonly named x. It is a very common practice to use these functions without naming the first argument (e.g., hist(y)).

3.2.2 Missing and NULL Arguments

There are two mechanisms in R to handle an argument with no well-defined default value: either define the default value for this argument as NULL, or detect that the argument is missing when the function is called. The latter works with the function missing (which also avoids an error if the argument has no default value):

```
> foo <- function(x) if (missing(x)) cat("'x' is missing\n")
> foo()
'x' is missing
```

The code with x defined as NULL by default is:

```
> bar <- function(x = NULL) if (is.null(x)) cat("'x' is NULL\n")
```

```
> bar()
'x' is NULL
```

It is not trivial whether to use one or the other, but they can be more or less equivalent thanks to the two functions `missing` and `is.null`.

A subtle fact is that an argument appears to be present in the environment of the function as shown by the output from `exists` or `ls`; however, printing the object might fail if it is missing with no default value:

```
> f <- function(x)
+ {
+     print(exists("x"))
+     print(ls())
+     print(x)
+ }
> f()
[1] TRUE
[1] "x"
Error in print(x) : argument "x" is missing, with no default
```

This error can be solved by assigning a value to the object if it is left missing during the call:

```
> g <- function(x)
+ {
+     print(exists("x"))
+     if (missing(x)) x <- NULL
+     print(x)
+ }
> g()
[1] TRUE
NULL
```

3.2.3 The '...' Argument

The '...' (pronounce "dot-dot-dot") argument is a powerful way to specify arguments which are not defined a priori. There are several usages of this.

The first usage is to pass arguments from one function to another which is called by the first one. Let us take a simple example: suppose we want to show some data that always vary between 1 and 1000. We thus write a custom plot function where the x-axis is always from 1 and to 1000. That would be something like:

```
customplot <- function(x, y)
    plot(x, y, xlim = c(1, 1000))
```

However, `plot` is a very rich function with many options (see `?plot.default`) and it would be good to be able to use them in `customplot()`. A simple modification of the previous code can make this possible:

```
customplot <- function(x, y, ...)
    plot(x, y, xlim = c(1, 1000), ...)
```

With this new version, all arguments which are not named `x` or `y` are "collected" in the '`...`' and passed to `plot()`. If `xlim` is one of them, then an error happens. To avoid this, a further modification is to move this argument as an option:

```
customplot <- function(x, y, xlim = c(1, 1000), ...)
    plot(x, y, xlim = xlim, ...)
```

The second usage of the '`...`' is to define a function where the number of arguments is unlimited as is illustrated below with the simplest case where '`...`' is the only argument of `foo`:

```
> foo <- function(...) print(list(...))
> foo(1)
[[1]]
[1] 1

> foo(1, 1:3)
[[1]]
[1] 1

[[2]]
[1] 1 2 3

> foo(1, 1:3, 1:5)
[[1]]
[1] 1

[[2]]
[1] 1 2 3

[[3]]
[1] 1 2 3 4 5
```

We note that the '`...`' is first "captured" in a list before being manipulated. This can also be useful in the first usage of the '`...`' to check the names of the arguments, for instance:

```
> foo(col = "blue")
$col
[1] "blue"
```

The '...' is now a list with names so its elements can be assessed individually. Alternatively, the list can be created (and manipulated if needed) before calling the function, and passed efficiently by using the function `do.call` with:

```
> L <- list(col = "blue")
> do.call(foo, L)
$col
[1] "blue"

> L$lty <- 1 # add an element to the list
> do.call(foo, L) # repeat the do.call
$col
[1] "blue"

$lty
[1] 1
```

3.3 Return Value

Functions are used to make calculations and, if the calculations are successful, return a result. This last one is a single object and called the *return value* of the function.

3.3.1 Implicit and Explicit Returns

The most common way to return a value is to write the name of an object (or an expression) at the end of the code of the function (the body): in that case there is no need to use the function `return` (this is an implicit return). On the other hand, it is possible to do an explicit return by using the latter: in this case the called function is stopped and the value is returned. The two following function definitions have the same effect:

```
foo <- function(x)
{
    if (!is.numeric(x)) return(NULL)
    ## ....
    x
}
```

```
foo <- function(x)
{
    if (!is.numeric(x)) return(NULL)
    ## ....
    return(x)
}
```

A function always returns something, even if nothing is done during its execution in which case `NULL` is returned invisibly:

```
> foo <- function() if (FALSE) 1
> foo()
> o <- foo()
> o
NULL
```

It may be needed to perform some commands before returning the return value. This can be done by writing some lines of code before the last command; however, the code might not be executed if there is an error, or an a call to `return`, before it is reached. The function `on.exit` takes one or several lines of commands, and delays their execution until when the function call is closed:

```
> bar <- function()
+ {
+       on.exit({
+           cat("End.\n")
+           cat("Goodbye!\n")
+       })
        cat("Starting... ")
+       0
+ }
> bar()
Starting... End.
Goodbye!
[1] 0
```

3.3.2 Assignment and Superassignment

In addition to returning a single object, a function can modify or create objects at any step of its execution. There are two ways to do this: either with the function `assign` or with the superassignment operator `<<-`. We have already seen `assign()`, let us see its arguments now:

```
> args(assign)
function (x, value, pos = -1, envir = as.environment(pos),
    inherits = FALSE, immediate = TRUE)
```

The arguments `pos` and `envir` are alternative ways to specify where the object should be created: the latter is the name of the environment, whereas the former is its position in the search list. From the previous section, we remember that `assign("x", 1)` is similar to `x <- 1`. On the other hand, `assign` cannot be used to modify an element of `x` (i.e., there is no equivalent of `x[1] <- 1`). Instead, `x` can be read with `get`, modified locally, and then (back-)copied with `assign`.

Funnily, `assign()`, or the backtick operator ‘, makes possible to create objects with non-conventional names (i.e., which would give an error with the standard assignment operators) so that we can show that $2 + 2 = 5$:

```
> ‘2‘ <- 3
> ‘2‘ + 2
[1] 5
```

The superassignment operator is a practical short-cut to assign an object in the parent environment:

```
foo <- function(x)
{
    y <<- 0 # y is created in the parent environment
    x
}
```

After loading this function in R, we can try it with:

```
> ls()
[1] "foo"
> foo(x = 1)
[1] 1
> ls()
[1] "foo" "y"
> y
[1] 0
```

Initially, `foo` is the only object in memory. After executing the command `foo(x = 1)`,[1] an object named `y` has been created; but we note that there is no object `x`.

By contrast to `assign()`, an existing object can be modified directly by `<<-`.

```
bar <- function(x)
{
    ## y must exist in the parent environment:
    y[] <<- 1
    x
}
```

After loading `bar`, we can try it with:

```
> bar(2)
[1] 2
> y
[1] 1
```

[1] This could have been `foo(1)`; see Sect. 3.2.

This is particularly relevant for recursive functions (see next section).

3.4 Recursive Functions

A recursive function is a function that can call itself. The concept of *recursion* applied to computer programming must not be confused with the concept of a *mathematical recursive function* usually written as $x_{i+1} = f(x_i)$. Such mathematical formulas can usually be programmed efficiently without a recursive function.

Almost all modern computer languages implement recursive functions. A common exercise for programming a recursive function is to apply this concept to the mathematical factorial which is denoted with an exclamation mark ($x!$) and defined as:

$$x! = 1 \times 2 \times 3 \times \ldots x.$$

It is indeed recursive:

$$(x+1)! = (x+1) \times x!,$$

so that an R function can easily be written as:

```
fact <- function(x)
{
    if (x <= 1) return(1)
    fact(x - 1) * x
}
```

As mentioned above, this is not the most efficient way to program the factorial in R. Three alternatives, with exactly the same outputs, are:

```
fact1 <- function(x)
{
    if (x <= 1) return(1)
    res <- 1
    i <- 2
    while (i <= x) {
        res <- res * i
        i <- i + 1
    }
    res
}

fact2 <- function(x)
{
    if (x <= 1) return(1)
    res <- 1
```

```
    for (i in 2:x)
        res <- res * i
    res
}

fact3 <- function(x)
{
    if (x <= 1) return(1)
    prod(2:x)
}
```

In practice, the mathematical factorial is rarely used directly for two reasons. First, the product grows very quickly with x so that it can be computed only for $x \leq 170$:[2]

```
> prod(1:170)
[1] 7.257416e+306
> prod(1:171)
[1] Inf
```

Second, the factorial is used in some probability density functions (e.g., the Poisson distribution: $\lambda^x \mathrm{e}^{-\lambda}/x!$) which are mostly used in likelihood functions; however, the likelihood (a product of probability densities) is usually transformed on a logarithmic scale, so that products are transformed into sums and we need to compute the log-factorial:

$$\ln x! = \ln 1 + \ln 2 + \ln 2 + \cdots + \ln x.$$

R has the functions `factorial` and `lfactorial` to perform these calculations efficiently (actually using the Γ function since $x! = \Gamma(x+1)$):

```
> factorial(171)
[1] Inf
> lfactorial(171)
[1] 711.7147
```

After these considerations, we can define two simple guidelines before considering coding a recursive function in R:

- Do not build a recursive function to program a simple mathematical recursive function: they are usually efficiently coded with simple iterations, either with vectorisation, or with a `for` loop. In some common cases, they may be already implemented directly in R (see `factorial` and `lfactorial`).

[2] We will see in Section 5.4 why 171!, which is obviously a finite integer number, is considered as $+\infty$ by R.

- Recursive calls of R functions are useful—and sometimes indispensable—when there are nested (`for`) loops and their number cannot be determined a priori.

The fact is that recursive functions are rarely needed, but when this happens they are the only solution to the problem and are particularly efficient. Here are some examples I encountered in practice:

1. We want to create a random binary tree by splitting a set of n units: the first step is to choose two random integers, say a and b, so that $a+b=n$. Then the same step is applied, separately to a and b, until each is equal to one (or two). The number of required iterations is unknown because it depends on the sequence of a's and b's.
2. ZIP files are compressed archives of one or several files. Sometimes ZIP files contain files which are themselves ZIP files. The function `unzip` can list and extract files from a ZIP archive, but what if some are also ZIP files? A conditional `for` loop can be used if we are sure that there is not more than two levels, but this will not work if there are more levels.
3. We want to list all possible combinations of a number of units which can take different states (say A, B, and so on). The number of units can vary. A combination can have the same unit several times and their order is not important (i.e., AAB, ABA, and BAA are the same).

All these problems can be solved with a recursive function. In practice there are a few points to keep in mind to implement a recursive function:

- The recursive function is generally included in another function which is called by the user.
- The recursions are controlled by a conditional command (e.g., `if`) in order to avoid infinite recursions.
- The final result is usually modified by superassignment (`<<-`) so that the successive recursions modify the same object which, in the end, is returned by the main function called by the user.

The last point is important and makes recursive functions very useful in R. Figure 3.2 shows the structure of a hypothetical code. The left-hand side shows the outline of an imaginary function, `foo`, which is called by the user. `foo` includes a recursive function, `bar`. The final result, `res`, is created before calling `bar` which does some operations resulting in modifying `res` by superassignment. If the size of `res` is known in advance, this could be `res[i] <- ....` instead. Finally, `bar` calls itself or not depending on some conditions specific to the task. `res` is returned by `foo` by an implicit return. We note that `bar` has no explicit return value.

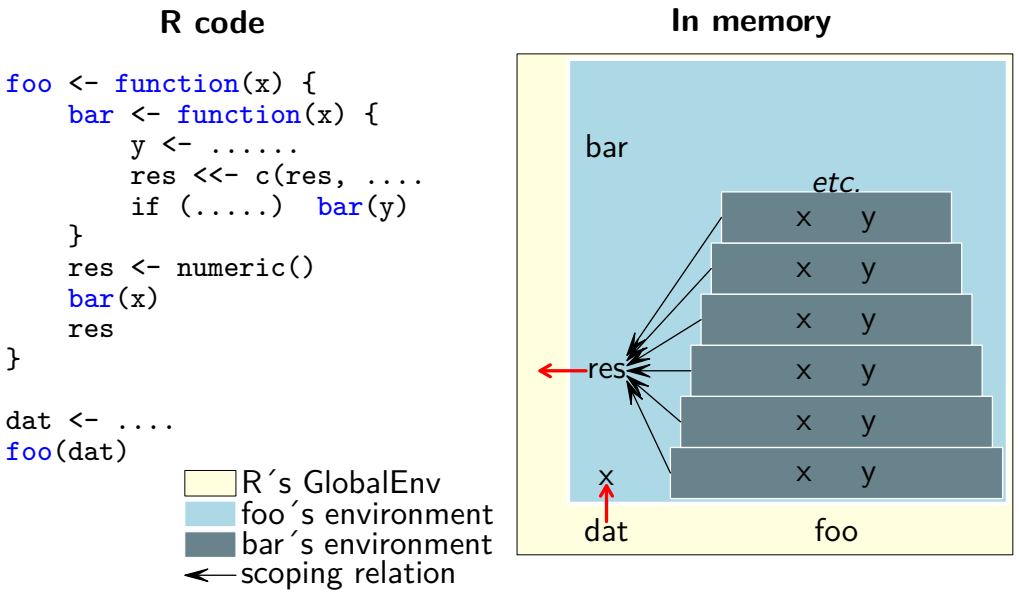


Fig. 3.2. A recursive function (`bar`) within another function (`foo`). The red arrows show the exchange of data between the user and `foo`. The stack of `bar`'s environments represents the recursive calls (from bottom to top).

The trick of recursive functions in R is that the parent environment (see Sect 3.1) of a function is set to the environment when it is created—not the environment when it is called. In Fig. 3.2, `bar` is created in `foo`, so the parent environment of `bar` is the environment of `foo` whatever the level of recursions. Therefore, the object `res` is unique and located in the environment of `foo`. On the other hand, the formal arguments of `bar` (`x`) and the objects eventually created inside `bar` (`y`) are all different for each call of this function.

To give an example of the use of a recursive function in R, let's take the last example from the above list. In practice this problem arises with genetic data when enumerating all possible genotypes. A human individual has two chromosomes, one from each parent, so that they have two copies of all genes. The genotype is the composition of these two genes which may or may not be different (the variants of the same gene are called *alleles*). Although many living beings, like humans, have a pair of each chromosome, some have four, six, eight (rarely three) copies of the same chromosomes, and many have a single copy. Besides, the number of alleles very greatly with the genes. Thus it is of interest to be able to list the possible genotypes in all cases. The following code defines `k` the number of alleles, and `ploidy` the number of chromosomes. The number of possible genotypes is given by the number of combinations of `ploidy` out of `k + ploidy - 1`. Let's take an example with three alleles and four chromosomes:

```
> k <- 3
```

```
> ploidy <- 4
> (N <- choose(k + ploidy - 1, ploidy))
[1] 15
```

So there are 15 possible genotypes. The recursive function to list all of them, `bar`, is:

```
bar <- function(i, a) {
    for (x in a:k) {
        g[i] <<- x
        if (i < ploidy) {
            bar(i + 1L, x)
        } else {
            j <<- j + 1
            ans[j, ] <<- g
        }
    }
}
```

It can now be called after setting the required objects:

```
> ans <- matrix(NA, N, ploidy)
> j <- 0
> g <- integer(ploidy)
> bar(1L, 1L)
```

The execution of `bar` on the last line modified the objects already created (`ans`, `j`, and `g`), so nothing is returned or printed. We finally check that `ans` now stores the results:

```
> ans
      [,1] [,2] [,3] [,4]
 [1,]    1    1    1    1
 [2,]    1    1    1    2
 [3,]    1    1    1    3
 [4,]    1    1    2    2
 [5,]    1    1    2    3
 [6,]    1    1    3    3
 [7,]    1    2    2    2
 [8,]    1    2    2    3
 [9,]    1    2    3    3
[10,]    1    3    3    3
[11,]    2    2    2    2
[12,]    2    2    2    3
[13,]    2    2    3    3
[14,]    2    3    3    3
[15,]    3    3    3    3
```

This application is further explored in the exercises at the end of this chapter.

3.5 Classes and Generic Functions

In Chapter 2, we have seen that the attributes of an object have an important role by storing additional information about the data stored by this object. One of these attributes is widely used in this respect: the *class*. It is an optional attribute; however, it is very useful to learn a few details about it since many functions in R are *generic* (e.g., `print`, `summary`, `plot`).

There are three distinct mechanisms that make use of the class in R: S3, S4, and R6. Before exploring them in the following sections, let us define a few terms used in this section:

class: an optional attribute made of a vector of mode character with length ≥ 1 (rarely more than 3).

generic: a function which operates depending on the class of its argument.

method: a function which is called by a generic function.

3.5.1 S3

This is the basic system of class used in R and is implemented in the package base. The idea behind the S3 system, and S4 and R6 as well, is the concept of *object-oriented programming*: it is very common that some analyses or computations depend on the type of data under consideration. This is, according to the above definition, what a generic function does.

To illustrate this mechanism, let's take the simple operation of printing an object in R which is done by the function `print`. Because a function is an object, we can print it in R:

```
> print
function (x, ...)
UseMethod("print")
<bytecode: 0x5555ce3a76c8>
<environment: namespace:base>
```

`print` is a *generic* function: it looks for another function that does the actual work of printing the object given as argument. This latter function is a *method* and its name is `print.toto` to print objects of class `"toto"`, or `print.titi` to print objects of class `"titi"`, etc.

It's a simple matter to set the class of an object: use the function `class` to do it. The class is an optional attribute, so it can also be created or modified like other optional attributes (Sect. 2.3.2). The class is a vector of mode character which can be of length one or more. In other words, an object can have several classes. In that case, the order of these classes is important. Suppose the class is something like:

```
> class(x)
[1] "class1" "class2" "class3" ..... "classn"
```

Then the command `print(x)` will first search for the function `print.class1`: if it exists (to be more accurate, if it is loaded in memory), then it is used to print `x` and the operation is done; otherwise, `print.class2` is searched for, and so on until `print.classn`. If none of these functions are available, the function `print.default` is searched for: if this last function is not available, an error happens.[3]

Although simple, the S3 class system can become tricky if little care is taken in building the chain of classes.

3.5.2 S4

In the late 1990's, another scheme was defined: S4 classes (see Chambers's book [1] for a historical account). Two main reasons motivated the development of S4. First, when an object of a given S3 class is created, R does not check whether its contents is correct or matches with what some functions might expect. For instance, nothing prevents us to create an object of class "dist" which is not a distance matrix:

```
> x <- 1
> class(x) <- "dist"
> str(x)
 'dist' num 1
> x
Error in matrix(0, size, size) : non-numeric matrix extent
```

Clearly, the function `print.dist` cannot handle `x`—and surely other functions will have the same issue.

The second problem is more subtle and is related to class inheritance in S3. We can give a simplified example of this by adding a class to the object `x` (say "toto"), and writing an appropriate `print` method:

```
> class(x) <- c("toto", "dist")
> print.toto <- function(x, ...) print.default(x)
```

`x` can now be printed with no error:

```
> x
[1] 1
attr(,"class")
[1] "toto" "dist"
```

[3] In this hypothetical example of a composite class, the default method `print.default` actually exists in the package base. If you create an S3 generic function, it is up to you to create its default method.

As mentioned above, the order of the classes is important and inverting them results in the same problem as above:

```
> class(x) <- c("dist", "toto")
> x
Error in matrix(0, size, size) : non-numeric matrix extent
```

This becomes more difficult to track down with more than two classes. This problem is called *inconsistent class inheritance* by Chambers [1].

The functions needed to handle S4 classes are in the package methods which is loaded by default when R starts. By contrast to S3, an S4 class must be defined explicitly in R before we can create an object. This is done with the function `setClass`. As a simple example, we want to create a database of scientists with the names and dates of birth, and we call this S4 class "persons":

```
setClass("persons",
         representation(Name = "character",
                        Year = "numeric",
                        Month = "numeric",
                        Day = "numeric"))
```

Loading the above code defines the class "persons" stored in an object named `.__C__persons` (hence hidden to `ls()` by default):

```
> ls(all.names = TRUE)
[1] ".__C__persons"
> .__C__persons
Class "persons" [in ".GlobalEnv"]

Slots:

Name:       Name      Year     Month       Day
Class: character   numeric   numeric   numeric
```

The elements of an S4 class are called *slots*. An S4 object is created with the function `new`. Let's try with two famous naturalists:

```
> x <- new("persons",
+          Name = c("Charles_Darwin", "Alfred_Wallace"),
+          Year = c(1809, 1823),
+          Month = c(2, 1),
+          Day = c(12, 8))
> x
An object of class "persons"
Slot "Name":
[1] "Charles_Darwin" "Alfred_Wallace"

Slot "Year":
```

```
[1] 1809 1823

Slot "Month":
[1] 2 1

Slot "Day":
[1] 12  8
```

The slots are extracted with the `@` operator:

```
> x@Name
[1] "Charles_Darwin" "Alfred_Wallace"
> x@Year
[1] 1809 1823
```

What happens if we try to create an object with an incorrect element?

```
> new("persons", Name = "Charles_Darwin",
+     Year = 1809, Month = "Feb", Day = 12)
Error in validObject(.Object) :
  invalid class "persons" object: invalid object for slot "Month
     " in class "persons": got class "character", should be or
     extend class "numeric"
```

We can now avoid this obvious mistake, but we can still create invalid objects:

```
> new("persons", Name = "Charles_Darwin",
+ Year = 1809, Month = 15, Day = 12)
An object of class "persons"
Slot "Name":
[1] "Charles_Darwin"

Slot "Year":
[1] 1809

Slot "Month":
[1] 15

Slot "Day":
[1] 12
```

`setClass` defines the slot `Month` as a numeric but it does not say that it should be in the range 1–12. The function `setValidity` allows us to define what should be a valid S4 "persons" object. We first create a function to perform the tests on the object; we can give it any name:

```
valid.persons <- function(object) {
    if (!is.character(object@Name)) {
```

```
        cat("slot 'Name' not character\n")
        return(FALSE)
    }
    if (!is.numeric(object@Year)) {
        cat("slot 'Year' not numeric\n")
        return(FALSE)
    }
    if (!is.numeric(object@Month)) {
        cat("slot 'Month' not numeric\n")
        return(FALSE)
    }
    if (object@Month < 1 || object@Month > 12) {
        cat("'Month' value invalid\n")
        return(FALSE)
    }
    if (!is.numeric(object@Day)) {
        cat("slot 'Day' not numeric\n")
        return(FALSE)
    }
    if (object@Day < 1 || object@Day > 31) {
        cat("'Day' value invalid\n")
        return(FALSE)
    }
    TRUE
}
```

The function must return a single logical value. We then associate this validity checker with the class (after loading `valid.persons` in memory):

```
> setValidity("persons", valid.persons)
Class "persons" [in ".GlobalEnv"]

Slots:

Name:       Name      Year     Month       Day
Class: character   numeric   numeric   numeric
```

If we try again the previous operation, this now returns an error:

```
> new("persons", Name = "Charles_Darwin",
+ Year = 1809, Month = 15, Day = 12)
'Month' value invalid
Error in validObject(.Object) : invalid class "persons" object:
   FALSE
```

Of course, we can still create an invalid object as long as it passes the tests defined in `valid.persons()`. Further tests can be added to avoid this and

other problems, for instance testing the number of days with respect to each month.

Generic functions and their associated methods are defined with the functions `setGeneric` and `setMethod`, respectively. We may thus create a way to display our S4 object in a more compact way than printed above:

```
> printPerson <- function(object)
+     cat("Database with", length(object@Name), "person(s)\n")
> setMethod("show", "persons", printPerson)
> x
Database with 2 person(s)
```

If an S4 class inherits from another class, this must be defined in the call to `setClass` with the option `contains`: in this case all the slots of the inherited class are included in the new one. Suppose we wish to create a new database of British naturalists, so that the new class, "British_persons", extends the class "persons" with the additional slot "PlaceOfBirth":

```
setClass("British_persons",
         representation(PlaceOfBirth = "character"),
         contains = "persons")
```

A new object can be created in the same way than above:

```
> xb <- new("British_persons",
+           PlaceOfBirth = c("Shrewsbury", "Llanbadoc"),
+           Name = c("Charles_Darwin", "Alfred_Wallace"),
+           Year = c(1809, 1823),
+           Month = c(2, 1),
+           Day = c(12, 8))
> xb
Database with 2 person(s)
```

We can see that the `print` method for the class "persons" has been used, thus avoiding to write explicitly the hierarchy of class like in S3 and preventing inconsistent class inheritance. Alternatively, the function `as` can be used to create a new object from `x` and then modify the slot "PlaceOfBirth" in the usual way:

```
> y <- as(x, "British_persons")
> y@PlaceOfBirth <- c("Shrewsbury", "Llanbadoc")
> identical(y, xb)
[1] TRUE
```

3.5.3 R6

The package R6 is based on S3. Its main function, `R6Class`, creates an environment which includes objects related to the new class.

To see the basic functioning of R6, let us start with a simple generator that returns an empty object:

```
> library(R6)
> X <- R6Class("X")
```

We may check that this is an environment:

```
> class(X)
[1] "R6ClassGenerator"
> mode(X)
[1] "environment"
> is.environment(X)
[1] TRUE
```

There are only two methods related to this class:

```
> methods(class = "R6ClassGenerator")
[1] format print
see '?methods' for accessing help and source code
```

`X` has the required information to create an object of class `"X"`:

```
> ls.str(envir = X)
active :  NULL
class :  logi TRUE
classname :  chr "X"
clone_method : function (deep = FALSE)
cloneable :  logi TRUE
debug : function (name)
debug_names :  chr(0)
get_inherit : function ()
has_private : function ()
inherit :  NULL
is_locked : function ()
lock : function ()
lock_class :  logi FALSE
lock_objects :  logi TRUE
new : function (...)
parent_env : <environment: R_GlobalEnv>
portable :  logi TRUE
private_fields :  NULL
private_methods :  NULL
public_fields :  NULL
public_methods : List of 1
 $ clone:function (deep = FALSE)
self : Class 'R6ClassGenerator' <X> object generator
  Public:
```

```
    clone: function (deep = FALSE)
  Parent env: <environment: R_GlobalEnv>
  Locked objects: TRUE
  Locked class: FALSE
  Portable: TRUE
set : function (which = NULL, name = NULL, value, overwrite =
   FALSE)
undebug : function (name)
unlock : function ()
```

The object is created by calling the function `new` which is inside the environment `X`:

```
> x <- X$new()
> x
<X>
  Public:
    clone: function (deep = FALSE)
> class(x)
[1] "X"  "R6"
> is.R6(x)
[1] TRUE
```

Currently (2022-10-03), 439 packages on CRAN rely on R6.

3.6 Exercises

1. Create an environment `e` in your workspace. Create another environment `f` which is inside `e`. Explain, eventually with a picture, the scoping relation of these environments.

2. Suppose there is a function with two arguments. How many ways are there to pass these arguments when calling this function? Answer the question by supposing first that there are no default values, then that both arguments have a default value.

3. Write a function that "captures" the '`...`' argument into a list, modifies this list, and returns it. Explain the usefulness of this manipulation.

4. Explain why the function `foo` at the end of Section 3.3.1 "does nothing".

5. Explain as simply as possible what is happening when executing the code that demonstrates that $2 + 2 = 5$ on page 29.

6. Assess the performances of the different R implementations of the factorial function given on page 30. You will also comment on the memory resources required by these functions. (Hint: you may need to use some resources from Chap. 7.)

7. The Fibonacci series is defined by: $f_0 = 0$, $f_1 = 1$, and $f_i = f_{i-1} + f_{i-2}$ for $i \geq 2$. Propose R functions to implement Fibonacci series either with or without a recursive function. Compare the performances of both functions. (See the hint of the previous question.)

8. Write a function solving the example of nested ZIP archives on page 32.

9. Draw a picture similar to Fig. 3.2 applied to the recursive function listing all genotypes (p. 34).

10. Give the logic (or, better, the algorithm) explaining how the recursive function listing all genotypes (p. 34) works.

11. Create a data structure which associates a similarity matrix with a factor. Give a class (with the name of your choice) to this structure, and write a `print` method for it. Use this method, then delete it from your workspace and print the structure. Explain what you observe.

4

Data Manipulation

Data manipulation is a vast topic. Although there are many particularities depending on the field of interest, it is possible to define some important, general topics about data manipulation that are addressed in this chapter. We first explore some general issues before treating indexing[1], the powerful tool for manipulating data in R.

4.1 Missing Data

There is no ideal solution on how to handle missing data in statistical analyses, and how this issue is approached varies a lot from a field of research to another.

4.1.1 Missing Values in Data Files

With the exception of the `"raw"` mode, all modes in R have a specific internal coding for missing values (Table 2.1). By contrast, many data coding systems use a specific value to code for missing data (for instance, many standards use -9 for missing data if the possible values are positive, such as body mass, population size, ...)

Because there is no universal rule to code missing values in data files, functions such as `read.table` or `scan` have the option `na.strings`. For these two functions, the default of this option is `"NA"` (and not `NA`; see next section for the difference). This option accepts a vector of length two or more, for example, `na.strings = c("NA", "na", "Na")`.

[1] Indexing is usually defined as "the act of classifying and providing an index in order to make items easier to retrieve" (`https://wordnet.princeton.edu/`). In R, indexing is the operation to manipulate data using the `[` operator.

Table 4.1. How special values are tested.

x	is.na(x)	is.nan(x)	is.finite(x)	is.infinite(x)
NA	TRUE	FALSE	FALSE	FALSE
Inf[a]	FALSE	FALSE	FALSE	TRUE
NaN[b]	TRUE	TRUE	FALSE	FALSE

[a]E.g., $\log(0)$, $1/0$.
[b]E.g., $\log(-1)$.

4.1.2 NA *vs.* NaN

NA (Not Available) is the code for missing data, whereas NaN is returned by operations that produce 'Not a Number'. However, it makes sense to treat the latter as missing values, so `is.na(NaN)` returns TRUE while `is.nan(NA)` returns FALSE (Table 4.1). We note that to be a missing value, NA should not be quoted (see Sect. 4.3 for more details on defining character strings in R):

```
> is.na(NA)
[1] TRUE
> is.na("NA")
[1] FALSE
> is.na("")
[1] FALSE
```

The default mode of NA is logical, and it is eventually converted when assigned into a vector of a different mode:

```
> y <- NA
> mode(y)
[1] "logical"
> x <- 1:2
> x[2] <- y
> x
[1]  1 NA
> mode(x)
[1] "numeric"
> is.na(x[2])
[1] TRUE
> is.na(y)
[1] TRUE
```

There are built-in NA's of different storage modes which can be used for more efficiency:

```
> L <- list(NA_character_, NA_complex_, NA_integer_, NA_real_)
> unlist(L)
```

```
[1] NA NA NA NA
> sapply(L, mode)
[1] "character" "complex"   "numeric"   "numeric"
> sapply(L, storage.mode)
[1] "character" "complex"   "integer"   "double"
> sapply(L, is.na)
[1] TRUE TRUE TRUE TRUE
```

4.1.3 Missing Data in Data Analyses

With a brief experience in data analysis, a researcher quickly learns that the occurrence of missing data is a rule rather than an exception. This raises a number of issues, although most of these are field-specific (we will see a small example below in the case of regression models).

We have seen that missing values are stored with a special value (`NA`) whatever the data mode. When operating on a vector "value-wise" (i.e., so that the result is a vector of the same length than the input vector), things seem to be straightforward: a missing value which is transformed is still a missing value:

```
> x <- c(1, NA)
> log(x)
[1]  0 NA
```

As a side-note, the logarithm is able to generate all kinds of special values:

```
> log(c(-1, 0, 1, NA))
[1]  NaN -Inf    0   NA
Warning message:
In log(y) : NaNs produced
```

But what happens if we use a function that returns a single value, for instance, the sum of the values in `x`?

```
> sum(x)
[1] NA
```

Indeed, the addition of a number with an unknown value is, quite logically, also unknown. The function `sum`, like several others,[2] have the option `na.rm` (which is always `FALSE` by default):

```
> sum(x, na.rm = TRUE)
[1] 1
```

[2]These other functions are: `mean`, `var`, `median`, `quantile`, `max`, `min`, `range`, `prod`. Note that `mean` is a generic function which does not have the option `na.rm` but `mean.default` has it.

The occurrence of missing values but also their arrangement in a data set are important. Consider the following data frame:

```
      x    y  z
Ind1 11 -0.4  1
Ind2 12  1.8  2
Ind3 13 -1.2 NA
```

A linear regression performed with `lm(y ~ x)` will use the three rows of the table, whereas adding `z` as a predictor in the model (i.e., `lm(y ~ x + z)`) will imply to remove the third row when fitting this second model. A consequence is that both model fits are not comparable (the function `anova` gives a warning if the comparison is attempted). However, if the first fitted model were `lm(y ~ z)`, then the comparison would be valid. Numerical methods usually do not handle missing data, so that `lm` removes rows with at least one `NA`. This step can be handled explicitly by the user with the generic function `na.omit`.

4.2 Logical Vectors

Logical operations exist in all computer languages in order to control computations with statements such as `if`, `else`, or `while`. In R, additionally to this, logical values can be stored in vectors making them a powerful way to manipulate data.

Logical vectors are returned mainly by two operations:

- Comparison operators which are all binary:[3]

`==`	equal to
`!=`	different
`>`	greater than
`<`	less than
`>=`	greater than or equal
`<=`	less than or equal

- A function which tests a feature of a vector or another object with a name usually starting with `is.` (`is.na`, `is.numeric`, ...)

Logical values are internally coded with 0 (`FALSE`) and 1 (`TRUE`) resulting in efficient storage and handling. To illustrate this, let's simulate ten million random normal variates and test how many are greater than five:

```
> x <- rnorm(1e7)
> system.time(test.x <- x > 5)
   user  system elapsed
  0.019   0.016   0.035
```

[3]The *arity* of an operator (or a function) is its number of argument(s). An operator is unary, binary, or ternary, if it takes one, two, or three arguments, respectively.

This is a fast operation considering that 10^7 values were tested. We may then be interested to count the numbers of `TRUE` and of `FALSE` values in `test.x`. An intuitive solution might be to do this with `table()`:

```
> system.time(tabx <- table(test.x))
   user  system elapsed
  1.908   0.160   2.073
```

Using `tabulate()` is a much more efficient solution to count logical values:

```
> system.time(tabx2 <- tabulate(test.x + 1L))
   user  system elapsed
  0.054   0.008   0.062
> system.time(tabx2 <- tabulate(test.x + 1L, 2L))
   user  system elapsed
  0.034   0.020   0.054
```

This is because logical values are stored as integers. Note that we added one to `test.x` because `tabulate` handles only strictly positive integers. The results are identical to those returned by `table` but without the names:

```
> tabx
test.x
  FALSE    TRUE
9999997       3
> tabx2
[1] 9999997       3
```

In the present situation, `tabulate` is simpler and faster than `table` as the former accepts a single integer vector as its main argument:

```
> args(tabulate)
function (bin, nbins = max(1L, bin, na.rm = TRUE))
```

In practice, there are two other possibilities to obtain the same result: either using `sum` (remember that `TRUE` is 1 and `FALSE` is 0), or first calling `which()` which returns the indices of the `TRUE`'s then simply calling `length()`:

```
> sum(test.x)
[1] 3
> length(which(test.x))
[1] 3
```

How these two solutions perform in terms of running times?

```
> system.time(sum(test.x))
   user  system elapsed
  0.007   0.000   0.007
> system.time(length(which(test.x)))
```

```
   user  system elapsed
  0.007   0.000   0.008
```

`sum` is slightly faster than `which`, but the latter also gives the positions of the `TRUE` values in `test.x`.

In many cases, data selection or manipulation requires multiple criteria: one way to approach this is to use several logical vectors. There are three operators and one function to combine logical vectors:[4]

`&`	returns `TRUE` if both values are `TRUE` (AND)
`|`	returns `TRUE` if at least one value is `TRUE` (inclusive OR)
`!`	inverts logical values
`xor()`	returns `TRUE` if only one value is `TRUE` (exclusive OR)

The last one is itself built on the previous operators as shown by its code:

```
> xor
function (x, y)
{
    (x | y) & !(x & y)
}
```

This also shows that `&` and `|` are binary while `!` is unary (see above footnote about arity). All four return a logical vector of the same length than the arguments.

If there are several criteria, it is best to compute the logical vectors separately first, then combine them as needed. Below is an example similar to the previous one but this time with two vectors (`a` and `b`) each with 10^5 random values; we then create two logical vectors by testing the values greater than 2, and perform different tests combining these two logical vectors in order to address the question given as comment after each command:

```
> a <- rnorm(1e5)
> b <- rnorm(1e5)
> test.a <- a > 2
> test.b <- b > 2
> sum(test.a & test.b) # both a and b > 2
[1] 41
> sum(test.a | test.b) # either a or b (or both) > 2
[1] 4517
> ## the next one gives the difference of the two previous ones:
> sum(xor(test.a, test.b)) # either only
[1] 4476
> sum(!test.a & test.b) # a <= 2 and b > 2
[1] 2220
```

[4]The double versions `&&` and `||` are used only in control statements such as `if ()` or `while ()`.

Table 4.2. Three ways to initialise a character vector in R.

Command	`length(x)`	`is.na(x)`
`x <- character()`	0	`logical(0)`
`x <- ""`	1	`FALSE`
`x <- NA_character_`	1	`TRUE`

These operators can be combined in the usual way so that more than two selection criteria can be used together (e.g., `test1 & test2 & test3 & ...`).

4.3 Character Strings and Text

We have seen that character strings are stored in R so that each element of a vector is a string (p. 9). This has a number of consequences on manipulating strings and characters. Table 4.2 shows the distinction between a character vector of length zero, an empty string, and a missing value (all of mode character).

4.3.1 Encodings

Defined simply, an encoding is a mapping between the computer bits and the characters printed on a screen or on a printer. Although that sounds simple, encoding is complicated because there are many ways to define this mapping. Besides, there are a very large number of characters used by humans, and they may use different encodings.

One of the most well-known encoding is called ASCII (American standard code for information interchange): it uses 7 bits and was defined around 1965. Shortly after, around 1967, manufacturers of computers agreed to standardise their products so that the smallest quantity of information manipulated simultaneously is 8 bits. A consequence of this was that ASCII-encoded characters had an unused bit; this was used to define new encodings based on 8 bits, for instance Latin-1 widely used in Western Europe. Many of these encodings are known under different names because they went through different processes of standardisation. For instance, Latin-1 is also known as ISO-8859-1.

Unicode is a standard aimed at providing a unified encoding for all characters recognized in human writing systems, including those not in use today (e.g., Egyptian hieroglyphs). Initially, a system based on 16 bits (= 2 bytes) was defined, but the $2^{16} = 65\,536$ combinations were filled quickly (e.g., Chinese has around 50 000 characters used in different periods). Another encoding based on 32 bits was eventually defined giving more than four billion combinations.

Having an encoding based on 32 bits implies that all characters are stored on four bytes. However, in practice it is rather uncommon to mix Roman

letters with Egyptian hieroglyphs and Nordic runes. Besides, this makes all ASCII-encoded files incompatible with any of the 16- or 32-bit encodings. A compromise is given by UTF-8: in this encoding, the first byte indicates whether a character is coded on one or several bytes. If the last bit of this byte is 0, then the character is only coded by this byte. This makes UTF-8 compatible with ASCII, but not with Latin-1, although Latin-1 is itself compatible with ASCII.[5]

Because of its flexibility, UTF-8 is widely used nowadays and the default encoding of many computer applications, including R. The function `iconv` converts a character vector among different encodings. Unfortunately, because of the difficulties and complications outlined above, its use can be complicated. It depends on libraries installed on the machine. For instance, on a Linux system, there could be more than one thousand encodings:

```
> length(iconvlist())
[1] 1173
```

It is not a common to change the encoding of a character vector, but we can try it in order to see that non-ASCII characters are, as explained above, coded on more than one byte with the UTF-8 encoding:

```
> nchar("é")
[1] 1
> nchar("é", type = "bytes")
[1] 2
> z <- iconv("é", "UTF-8", "Latin1")
> z
[1] "\xe9"
> nchar(z, type = "bytes")
[1] 1
```

The function `nchar` returns by default the number of characters in each string of a vector of mode character; the option `type = "bytes"` asks this function to return the number of bytes instead of the number of characters. Because the encoding in usage in the present session of R is UTF-8, the vector `z` (which is encoded in Latin-1) is printed as an escape sequence with the prefix `\x` followed by the byte coded in hexadecimal. From Table A.1 (p. 136), we can find that the hexadecimal sequence `e9` corresponds to the bit sequence `11101001`. This can found also with the following command:

```
> rev(rawToBits(charToRaw(z)))
[1] 01 01 01 00 01 00 00 01
```

The functions `scan` and `read.table` (among others) have the options `fileEncoding = ""` and `encoding = "unknown"` which help to read character strings correctly.

[5] To be more accurate, ASCII is a subset of UTF-8 and also a subset of Latin-1.

`Encoding()` makes possible to extract or to change the encoding attribute of a vector of mode character without changing the sequence of bits:

```
> x <- "é"
> Encoding(x)
[1] "UTF-8"
```

From the previous example, we know that 'é' is coded with two bytes under UTF-8. We now change the encoding attribute of `x` to Latin-1 so that these two bytes will be interpreted as two characters under this new encoding:

```
> Encoding(x) <- "Latin1"
> x
[1] "Ã©"
> Encoding(x)
[1] "latin1"
```

We can check that `x` is now made of two characters coded by two bytes:

```
> nchar(x)
[1] 2
> nchar(x, type = "bytes")
[1] 2
```

Finally, we change the encoding of `x`—this time not only its attribute—to UTF-8:

```
> y <- iconv(x, "Latin1", "UTF-8")
> y
[1] "Ã©"
> nchar(y)
[1] 2
> nchar(y, type = "bytes")
[1] 4
```

Since these two characters are not part of the ASCII set, they are coded each by two bytes in UTF-8.

4.3.2 Regular Expressions

Regular expressions (often abbreviated 'regexp') is a powerful tool to search patterns in character strings or text files. In R, `grep()` provides a way to do search tasks that cannot be performed with operations such as logical comparisons. For instance, the operator `==` compares strings and returns `TRUE` only if they are identical. Consider the following vector of mode character:

```
> x <- c("Homo sapiens", "Homo erectus",
+        "Pan troglodytes", "Pan paniscus",
+        "Gorilla gorilla", "Gorilla beringei")
```

Table 4.3. Common simple regular expressions.

Regexp	Meaning
`.`	any character
`[fghjkmo]`	any one of the character within brackets
`[a-e]` or `[0-9]`	same than `[abcde]` or `[0123456789]`
`[^a]`	any character but `a`
`a{5}`	same than `aaaaa`
`a{5,}`	`a` is repeated five times or more
`a*`	same than `a{0,}`
`a+`	same than `a{1,}`
`a{n,m}`	`a` is repeated between n and m times
`^aze`	start of the string
`rty$`	end of the string

They are the names of six closely related species of humans and apes. It is possible to find which one is identical to "Homo sapiens" with `==`:

```
> which(x == "Homo sapiens")
[1] 1
```

But this operator cannot find the strings that contain "Homo" only. However, `grep` does exactly this:

```
> grep("Homo", x)
[1] 1 2
```

The first argument is a single character string (i.e., a vector of mode character and length one) and gives the regexp; the second argument is a vector of mode character.

In a regexp, the patterns (repetitions, alternatives, ...) are coded with special characters. Table 4.3 gives the most commonly used of these codes. In practice, and with a little bit of experience, these different codings can be combined, increasing the usefulness of regexps. For instance, the last search could have been done with `grep("^Homo", x)` to make sure to not match "Homo" in the middle of a string. Table 4.4 gives more codes for regexps.

In practice, it is easier to build a regexp step by step and test it while building it on a simple character vector to check that it works as expected.

Finding regexps can be complicated in practical applications, but the potential gains are important. For instance, it is not rare that data files have minor errors created during data input such as extra spaces before or after the text.[6] In these situations, regexps can be powerful to check the data. If

[6]Numeric data usually do not have this problem since leading and trailing spaces are ignored.

Table 4.4. Regular expressions with classes.

Regexp	Meaning
`[:alpha:]`	upper- and lowercase letters
`[:digit:]`	digits
`[:lower:]`	lowercase letters
`[:space:]`	spaces, tabulations and new lines
`[:upper:]`	uppercase letters
`[:alnum:]`	`[:alpha:]` and `[:digit:]`

we suspect that extra spaces have been inadvertently typed when entering the above names, the search could be done with `grep("^ *Homo", x)`.

Regexps are not only used for pattern searching, but also for operations such as pattern replacement or splitting strings (Table 4.5). Let's see the options of `grep` with their default values because most of them are common with other functions:

```
> args(grep)
function (pattern, x, ignore.case = FALSE, perl = FALSE, value =
    FALSE,
    fixed = FALSE, useBytes = FALSE, invert = FALSE)
```

`ignore.case`: whether to distinguish uppercase and lowercase letters (e.g., `"A"` and `"a"` are considered identical if `TRUE`).

`perl`: whether to use regexp PERL code.

`value`: whether to return the values of the vector instead of the indices.

`fixed`: whether to not interpret the first argument as a regexp; e.g., the two following commands have the same effect:

```
grep("\\^Homo", x)
grep("^Homo", x, fixed = TRUE)
```

`useBytes`: whether to match the regexp byte-wise instead of character-wise.

Table 4.5. Functions accepting a regular expression.

Function	Description
`apropos`	Search for object names loaded
`gsub`	Substitute pattern(s) in strings
`sub`	As above but only the first occurrence is replaced
`strsplit`	Split character strings

`invert`: whether to invert the search (i.e., return the indices of the elements that do not match the regexp if `TRUE`).

Finally, the functions `regexec`, `gregexec`, and `regexpr` return more details about the occurrence of the pattern within the strings, with information on their locations as additional attributes (see Table 4.6 below).

4.3.3 Approximate String Distance

The approximate string distance (or generalized Levenshtein distance) is tightly connected to the idea of pattern searching with regexps. This distance is defined by the minimal number of changes needed to modify a character string into another; it is implemented in the function `adist`. This function is flexible and can accept a single vector, in which case the distances are calculated between each of its strings, or two vectors, in which case the distances are calculated between each string from each vector (so the output matrix can be rectangular). Its arguments are:

```
adist(x, y = NULL, costs = NULL, counts = FALSE, fixed = TRUE,
      partial = !fixed, ignore.case = FALSE, useBytes = FALSE)
```

Several of the options are similar to those of `grep` (`fixed`, `ignore.case`, and `useBytes`). For instance, we have the trivial result:

```
> adist(LETTERS[1:3])
     [,1] [,2] [,3]
[1,]    0    1    1
[2,]    1    0    1
[3,]    1    1    0
```

Another simple example is to find how many changes are needed to convert the string "Julien" into "Julia" (we use here the option `counts`):

```
> adist("Julien", "Julia", counts = TRUE)
     [,1]
[1,]    2
attr(,"counts")
, , ins

     [,1]
[1,]    0

, , del

     [,1]
[1,]    1

```

```
, , sub

     [,1]
[1,]    1

attr(,"trafos")
     [,1]
```

Alternatively, we can input the two strings into a single vector:

```
adist(c("Julien", "Julia"), counts = TRUE)
     [,1] [,2]
[1,]    0    2
[2,]    2    0
attr(,"counts")
, , ins

     [,1] [,2]
[1,]    0    0
[2,]    1    0

, , del

     [,1] [,2]
[1,]    0    1
[2,]    0    0

, , sub

     [,1] [,2]
[1,]    0    1
[2,]    1    0

attr(,"trafos")
     [,1]     [,2]
[1,] "MMMMMM" "MMMMSD"
[2,] "MMMMSI" "MMMMM"
```

We notice that the attribute `"counts"` is an array.

The functions `agrep`, `agrepl`, and `aregexec` are similar to `grep`, `grepl`, and `regexec` but based on the approximate string distance: they have the option `max.distance = 0.1`.

Table 4.6 gives an overview of the functions for regexp search in R. Using these different functions depend on the context, and they complement each others. For example, `which(grepl(pat, x))` is the same than `grep(pat, x)`. So, the output from `grepl` can be combined with other logical operations.

Table 4.6. Overview of the functions doing regular expression search.

Function	Description
`agrep`	Pattern search using approximate distance with a threshold
`agrepl`	Same than `agrep` but returns a logical vector
`aregexec`	Same than `gregexec` using approximate distance
`gregexec`	Same than `regexec` with indices returned in a matrix
`gregexpr`	Same than `regexpr` but with disjoint matches
`grep`	Pattern search using regexp
`grepl`	Same than `grep` but returns a logical vector
`grepRaw`	Same than `grep` with raw vectors
`regexec`	Same than `regexpr` but with multiple matches of parenthesized subexpressions
`regexpr`	Same than `grep` but returns positions of the regexp within each string

4.3.4 Building Strings in R *vs.* in Files

We known that text strings can be input directly in R by typing them within straight quotes. But there are a few subtleties, for example including quotes themselves in the string. The quotes are *delimiters* of the string and are not included in it; only straight quotes are allowed (see Table 1.3), but other types of quotes (e.g., guillemets) can be inside the string if the encoding allows them. The opening and closing delimiters must be identical: if they are double quotes, then straight single quotes can be included inside the string without escaping them, and vice versa.

The backslash character \ is used to include special characters in strings when they are built interactively in R. More precisely, the character following a backslash is interpreted in a specific way by R according to a syntax which is summarised in Table 4.7. The function `cat` interprets these special characters and prints them as they look like in a file.

For example, `\n` codes for a new line (a.k.a. linebreak) and is considered as a single character (and it requires one byte to be stored):

```
> cat("\n") # blank line follows

> nchar("\n")
[1] 1
> nchar("\n", type = "bytes")
[1] 1
```

We note the particular behaviours of `\r` and `\b`:

```
> cat("aaa\r")
> cat("aaa\b")
```

```
aa> cat("aaa\b\n")
aa
```

Also it is sometimes necessary to combine these escaped characters in a single string. For instance, if several backslashes must be printed in a row, they must all be doubled inside the string:

```
> cat("\\\\\\\t\\\\\t\\\n")
\\\ \\  \
```

Table 4.7. Special (escaped) characters in R's strings and how they appear in a file (LF: linefeed; CR: carriage-return).

R string build	In file
`"\""` or `'"'`	"
`"'"` or `'\''`	'
`"\\"`, `"\\\\"` (etc.)	\, \\ (etc.)
`"\n"`	`<linebreak>` (LF or LF/CR)
`"\t"`	`<tabulation>`
`"\b"`	`<backspace>` (deletes the previous character)
`"\r"`	`<carriage return>` (deletes the line)

4.4 Indexing

Identifying or localizing a specific observation, variable, or value in a data set is a routine operation in data analysis. R has three types of indexing: numeric, logical, and character. These have been introduced in many places. Indexing is one of the main strengths of R. There is no rule to prefer one type of indexing over the others. In fact, these three types of indexing can interact efficiently (Fig. 4.1). Indexing is used a lot in R so that instead of repeating the generalities, we focus here on a few points.

- The vector of indices can be omitted in which case all elements of the object will be considered. For example, `x[] <- 1` will replace all values of `x` with the value one, which is more efficient than `x <- rep(1, length(x))`.
- There is a lot of flexibility in indexing a vector so that there is no error in a wide range of situations:

```
> x <- 1:3
> x[-5] # negative out of range, no error
[1] 1 2 3
> x[5] # positive out of range, returns NA
```

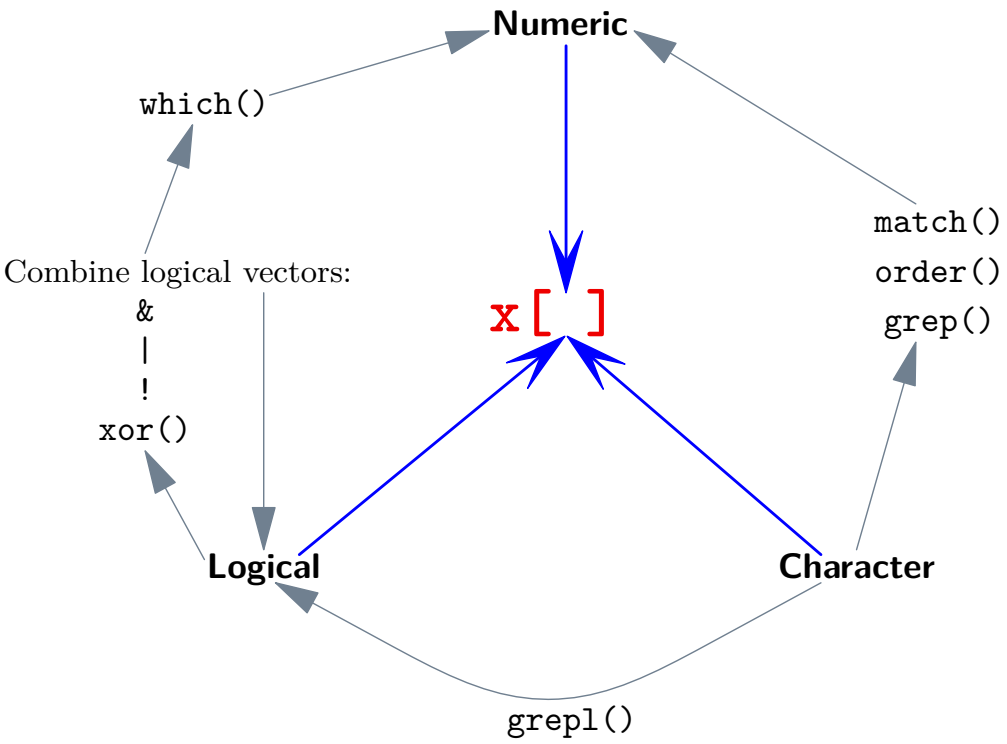


Fig. 4.1. The three types of indexing and how they can interact.

```
[1] NA
> x[5] <- 5 # out of range, x is extended
> x
[1]  1  2  3 NA  5
> x[0] <- 0 # x is unchanged
> x
[1]  1  2  3 NA  5
> x[-5] <- 0
> x
[1] 0 0 0 0 5
```

However, with a matrix there is less flexibility unless (implicit) conversion as a vector can be done:

```
> M <- matrix(1:4, 2)
> M[, 3]
Error in M[, 3] : subscript out of bounds
> M[5] # vector indexing
[1] NA
```

```
> M[5] <- 5
> M
[1] 1 2 3 4 5
```

- Different types of indexing can used in the same expression; for instance, `X[1:2, "bodymass"]` will consider the first two rows together with the column named "bodymass" in the matrix or data frame `X`.

- For lists and data frames, there is a subtle difference between the two extraction operators `$` and `[[`: the former takes a variable name not quoted whereas the latter accepts quoted variable names (or numeric). A consequence is that `[[` is more useful in programming such as:

```
for (v in names(X))
    X[[v]]
```

  Note that `v` is not quoted. It also makes easier to extract columns with non-standard names, such as `X[["2022"]]` (not to be confused with `X[[2022]]`; see Exercises), although the same command could use backticks `X$‘2022‘` but this last one is not easy to include in a program (e.g., it is easier and often more useful to use something like `v <- "2022"; X[[v]]`).

- Indexing lists for replacement can be done either with `[[` or with `[`. For instance if `X` is a list and `i` is an integer:

```
X[[i]] <- y
## y may be any type of object
## i must be of length one (numeric or names)
## i >= 1 & i <= length(X) must be TRUE
```

  In other words, when changing an element of a list with `[[`, the index must point to a single element that exists (an out-of-range index results in an error).

  When changing element(s) of a list with `[`, the same rules defined for vectors apply also for lists.

  With data frames, the objects will be checked to conform to the constraints of this type of objects (e.g., with the number of rows).

- Numeric indexing is the most efficient of the three types, but it is not always the best solution because the order of names or labels may vary in a data frame or other data structures.

  For logical indexing, combining logical vectors is very efficient and often simpler than combining numeric values. Furthermore, performance differences between the types of indexing are very likely to be noticeable only for very large data sets.

4.4.1 Recoding Data With Indexing

Indexing is a powerful way to recode data. This is especially efficient if there are a limited number of values, either numeric or character. In the case of numeric data, this is straightforward if the data are integer values. Suppose, the possible values are 1 and 2, and we want to recode them so that 1 is now 2, and 2 is now 1. We build a vector (here named `newcode`) with the new values (2 and 1), and use the data (here named `x`) as index to this vector. The procedure is illustrated below with ten values drawn randomly:

```
> newcode <- 2:1 # or: newcode <- c(2, 1)
> x <- sample.int(2, 10, replace = TRUE)
> x
 [1] 2 1 1 2 1 1 1 2 1 2
> newcode[x]
 [1] 1 2 2 1 2 2 2 1 2 1
```

This is simple, flexible, and efficient. We may have done, somehow intuitively, something like `x[x == 1] <- 2` with the side-effect all values would be equal to 2. This might be fixed by storing separately the result of `x == 1`, but this would become cumbersome if there are more than two possible values. Furthermore, the solution with `newcode` can be easily extended by adding more values to this vector.

This procedure can also be used if these values are not contiguous. For instance, if the possible values are 50, 60, 70, and 190 and we want to recode them as 1–4:

```
> newcode <- integer(190)
> newcode[50] <- 1
> newcode[60] <- 2
> newcode[70] <- 3
> newcode[190] <- 4
```

It might be simpler to create two vectors of the same length with the old and new values and use the first one as index of the vector `newcode` so it will be easier to manage more values:

```
> old <- c(50, 60, 70, 190)
> new <- 1:4
> newcode <- integer(max(old))
> newcode[old] <- new
```

One can visualise the recoding with `cbind(old, new)`.

With character strings, the recoding is also straightforward if the number of possible values is known. The recoding vector is now a vector of mode character with names; these names are used as indices during recoding. For instance, let's say we have a vector (`x` in the example below) with "Female"

and “Male” but also “FEMALE” and “MALE”, and we want to recode all these as either “F “or “M”:

```
> newcode <- c("Male" = "M", "MALE" = "M",
+              "Female" = "F", "FEMALE" = "F")
> x <- c("Male", "MALE", "Female", "FEMALE")
> newcode[x]
  Male   MALE Female FEMALE
   "M"    "M"    "F"    "F"
> unname(newcode[x]) # maybe better
[1] "M" "M" "F" "F"
```

Like above, we can prepare the recoding by creating two vectors `old` and `new`:

```
> old <- c("Male", "MALE", "Female", "FEMALE")
> new <- c("M", "M", "F", "F")
> newcode <- new        # OR: newcode <- setNames(new, old)
> names(newcode) <- old #
```

It is also possible to check that the length of the new codes is equal to the length of the unique values in the data:

```
> length(newcode) == length(unique(x))
[1] TRUE
```

4.5 Exercises

1. In the example on page 45, explain why `L` is a list and not a vector. What is the result of `mode(unlist(L))`?

2. One of your colleagues has a data set arranged in a data frame with 100 rows and 1000 columns. Overall, there is 0.1% of missing data, and your colleague thinks this is not a problem. Explain why this could be a problem bigger than initially thought.

3. Generate one million random variables following a normal distribution using the default parameters. How many of these variables satisfy the condition $|x| > 4$? Repeat this exercise but setting the variance of the normal distribution to $\sigma^2 = 2$. Could you calculate the numbers expected in both cases? (Hint: use `pnorm()`).

4. Find the regular expression (regexp) that will match the string ‘R’ and only this one. Look at the options in `grep` (and other functions) and find the one that bypasses the need to find this regexp.

5. List all objects (functions, data, ...) loaded in memory with a name starting with “lm”.

6. You want to split character strings into single words: find the efficient code to do this operation with `strsplit()`.

7. Explain how `match()` can be used when handling several objects, particularly data frames.

8. Explain why logical values used as indices are recycled but not numeric ones.

9. Explain why `v` is not quoted in the above example (p. 60). What would happen if it were quoted (i.e., `X[["v"]]`)?

10. Explain the difference between `X[["2022"]]` and `X[[2022]]`.

11. Compare the performance of logical and numeric indexing for vectors of different sizes (up to 10^8).

12. Write a program to perform Dawkins's weasel problem.[7] You will use the approximate string distance to evaluate the fitness of the new mutants. In addition to `adist`, you will probably need the following functions: `runif`, `sample`, `substr`, `which.min`, and others introduced in this chapter. Compare your results with an implementation that uses a fitness function based on the Hamming distance.[8]

[7] `https://en.wikipedia.org/wiki/Weasel_program`
[8] `http://rosettacode.org/wiki/Evolutionary_algorithm#R`

5

Special Topics

5.1 Expressions

Expressions are objects created after transforming some text into something that R can interpret. The step of creating an expression is intermediate before executing the command.

Consider three different objects, `x`, `y`, and `z`, all with the content "one":

```
> x <- 1
> y <- "1"
> z <- expression(1)
> ls.str()
x :  num 1
y :  chr "1"
z :   expression(1)
```

They have clearly different modes and distinct interpretations: the vectors `x` and `y` store a number and a character string, respectively. `z`, on the other hand, is of mode expression.

Expressions can be manipulated similarly to lists:

```
> length(z)
[1] 1
> str(z[1])
  expression(1)
> str(z[[1]])
 num 1
> z[[1]] <- 2
> z
expression(2)
```

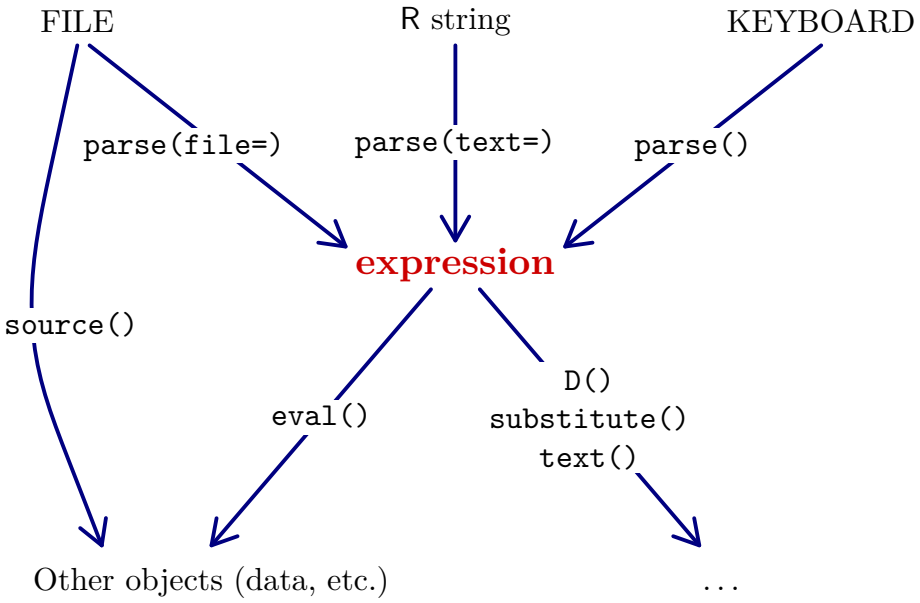


Fig. 5.1. An overview of the ways to create expressions and how they are used in R.

This kind of manipulation is rarely done since expressions are easily created from text in files, entered with the keyboard, or in R character string (Fig. 5.1).

Expressions are required by some functions, for instance `D` which computes partial derivatives:

```
> D(expression(log(x)), "x")
1/x
```

The second argument gives (as a character string, not an expression) the variable with respect to which the derivation is done, so the above command computes $\partial \ln x / \partial x$.

An expression can be evaluated with `eval()`:

```
> e <- expression(rnorm(5))
> e
expression(rnorm(5))
> eval(e)
[1] -0.4482620  1.9125654  0.5736916 -0.8275439  0.5217176
```

An expression can be created with `parse()`, either by calling it without argument, in which case the user enters the code at the prompt marked with a question mark:

```
> parse()
```

```
?1 + 3
expression(1 + 3)
```

Or by using the option `text` which takes a character string:

```
> a <- paste(1, 2, sep = " + ")
> a
[1] "1 + 2"
> e2 <- parse(text = a)
> e2
expression(1 + 2)
> eval(e2)
[1] 3
```

Expressions can be concatenated like any R objects, for instance using the `[` operator:

```
> e3 <- expression(x <- rnorm(5))
> e3[2] <- expression(y <- runif(5))
> eval(e3)
> x
[1]  1.2714243 -0.8285636  1.1887390 -1.9306594  0.8050333
> y
[1] 0.2794713 0.9192962 0.1273966 0.8500026 0.4434050
```

Expressions can be used to print formatted text in graphics: the usual R syntax is interpreted in a specific way when the expression is passed to the function `text`, `mtext`, `legend`, or a few others. A small difficulty is that operators used here as symbols cannot be input without being preceded by another symbol. A solution is to prefix them with `NULL`:

```
> expression(^2)
Error: unexpected '^' in "expression(^"
> expression(NULL^2)
expression(NULL^2)
```

This makes possible to combine strings with symbols in an expression to print something like "[Area covered]$_n$". Because of the space between the two words, it is necessary to quote them within an expression so that the string can be concatenated with symbols using the `*` operator:

```
> expression(Area covered[n])
Error: unexpected symbol in "expression(Area covered"
> expression("Area covered"*NULL[n])
expression("Area covered" * NULL[n])
```

Expressions can be combined together, separated by commas, and passed to `expression()`:

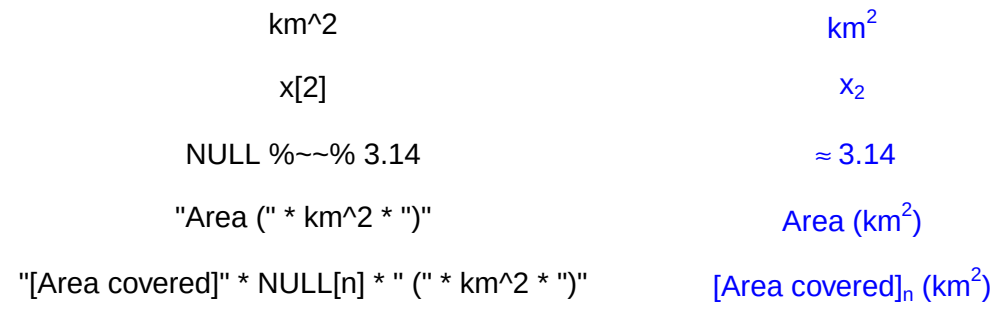


Fig. 5.2. Some examples of using expressions (in black) to annotate graphics (in blue).

```
> e <- expression(km^2,
+             x[2],
+             NULL %~~% 3.14,
+             "Area ("*km^2*")",
+             "[Area covered]"*NULL[n]*" ("*km^2*")")
> e
expression(km^2, x[2], NULL %~~% 3.14, "Area (" * km^2 * ")",
    "[Area covered]" * NULL[n] * " (" * km^2 * ")")
> (n <- length(e))
[1] 5
```

Figure 5.2 is the result of the following commands:

```
plot(NA, type = "n", ann = FALSE, axes = FALSE,
     xlim = 1:2, ylim = c(1, n))
text(1.25, n:1, as.character(e))
text(1.85, n:1, e, col = "blue")
```

There is a complete description of the syntax (including mathematical symbols, Greek letters, and others) in `?plotmath`. This help page gives also information about differences related to operating systems.

An argument passed to a function is considered an expression as long as it has not been evaluated. There are several functions that make possible to manipulate these objects within the function, although this is not always intuitive. The functions `quote` and `substitute` are two of these and return a result which mode depends on the argument (Table 5.1). This is mostly useful when manipulating formulas within a function (see Sect. 5.2).

`substitute()` is also useful when getting the (symbol) name of an argument passed to a function:

```
> foo <- function(x) cat("argument:", substitute(x), "\n")
> foo(E)
```

Table 5.1. Different outputs from `quote()`.

Command	`mode( )`	`str( )`
`quote(1)`	numeric	num
`quote(x)`	name	symbol
`quote(x + 1)`	call	language

```
argument: E
```

However, if the argument is of mode `"call"` (Table 5.1), then the expression must be deparsed with the function `deparse` which does the opposite operation to the function `parse` seen above:

```
> foo(x + 1)
argument: Error in cat("argument:", substitute(x), "\n") :
  argument 2 (type 'language') cannot be handled by 'cat'
> bar <- function(x)
+     cat("argument:", deparse(substitute(x)), "\n")
> bar(x + 1)
argument: x + 1
> bar(E)
argument: E
```

5.2 Formulas

Objects of class "formula" code for relationships among variables. They are mainly encountered in regression or other models. Formulas are of mode "call" which was introduced in the previous section. They illustrate several concepts already encountered in the previous sections and chapters so that it is interesting to examine them further.

Formulas are created with the ~ operator which separates the left-hand side term to the right-hand side one. A formula has two additional attributes: the class "formula" and an environment :

```
> m <- y ~ x
> m
y ~ x
> mode(m)
[1] "call"
> str(m)
Class 'formula'  language y ~ x
  ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
```

If we delete these attributes, the formula is very similar to the type of objects we have seen in the previous section:

```
> attributes(m) <- NULL
> m
y ~ x
> str(m)
 language y ~ x
> mode(m)
[1] "call"
```

This makes possible to include variable transformations inside the formula since no evaluation is done when it is created.

The environment attribute says where to look for the variables included in the formula. It can be modified or overridden by other arguments, for instance the option `data` in `lm()` and other model-fitting functions.

5.3 Dates and Times

There are many ways to print a date which are defined according to national standards.[1] On the other hand, computer scientists and engineers have defined a standard widely on computers: a date is stored as an integer giving the number of days since the origin (the date numbered zero) which is defined as the first day of 1970 (1970-01-01, in ISO 8601 notation). The class "Date" in R implements this standard. The function `as.Date` converts character strings into this class. A simple example with the dates input on the keyboard is:

```
> x <- c("1969-12-31", "1970-01-01", "1970-01-02")
> z <- as.Date(x)
```

When printed, the two objects `x` and `z` look identical but `mode()` and `str()` show that `z` is not a vector of mode character:

```
> x; z
[1] "1969-12-31" "1970-01-01" "1970-01-02"
[1] "1969-12-31" "1970-01-01" "1970-01-02"
> mode(x)
[1] "character"
> mode(z)
[1] "numeric"
> str(z)
 Date[1:3], format: "1969-12-31" "1970-01-01" "1970-01-02"
```

We can check that these three dates can be converted into numbers, but not the character strings:

```
> as.numeric(z)
```

[1] https://en.wikipedia.org/wiki/Date_format_by_country

```
[1] -1  0  1
> as.numeric(x)
[1] NA NA NA
Warning message:
NAs introduced by coercion
```

Logically, today's date should be the number of days since 1970-01-01:

```
> today <- as.Date("2022-10-03")
> as.numeric(today)
[1] 19268
> today - z[2]
Time difference of 19268 days
```

We note that the result is not a simple numeric subtraction but something which is meaningful for the class of these objects: this is because the minus operator is a generic function (Sect. 3.5).

The first argument to `as.Date()` is mandatory and can be an empty string in which case `NA` is returned. The second argument specifies some details and depends on the type of the first one (`as.Date()` is actually generic):

```
> as.Date("")
[1] NA
> as.Date("", "")
[1] "2022-10-03"
```

Surely, the interesting thing from this function is the possibility to read dates written in (virtually) any format thanks to the second argument which specifies the format using a coding system summarised in Table 5.2. The default is the international standard format (ISO 8601) `"%Y-%m-%d"`:

```
> as.Date("2022-02-03")
[1] "2022-02-03"
> as.Date("2022-02-03", "%Y-%m-%d") # identical to the previous
[1] "2022-02-03"
```

Use of day and month names (`%A`, `%a`, `%B`, `%b`) depends on the locale, that is the settings of the computer, particularly the language. For instance, "January" (abbreviated "Jan") is expected on an English locale whereas "janvier" (abbreviated "janv.") is expected on a French one. Dates in different languages can be mixed by changing the locale setting within R:

```
> Sys.setlocale(locale = "en_US.UTF-8")
[1] "LC_CTYPE=en_US.UTF-8; ..............." # <output skipped>
> today.us <- as.Date("February 3, 2022", "%B %d, %Y")
> Sys.setlocale(locale = "fr_FR.UTF-8")
[1] "LC_CTYPE=fr_FR.UTF-8; ..............." # <output skipped>
> as.Date("February 3, 2022", "%B %d, %Y")
```

Table 5.2. Syntax to read dates and times from text.

Code	Meaning
`%d`	day (01–31)
`%A`	weekday (full name)[a]
`%a`	weekday (abbreviated)[a]
`%u`	weekday (1–7, Monday is 1)[a]
`%m`	month (01–12)
`%B`	month (full names)[a]
`%b`	month (abbreviated)[a]
`%Y`	year (4 digits)
`%y`	year (2 digits, use with care[b])
`%H`	hours (00–23)
`%I`	hours (01–12)
`%p`	AM/PM (used with `%I`)
`%M`	minutes (00–59)
`%S`	seconds (00–61)

[a]System-dependent (locale) but partial matching.
[b]See examples.

```
[1] NA
> today.fr <- as.Date("3 février 2022", "%d %B %Y")
> identical(today.us, today.fr)
[1] TRUE
```

In this example, we first set the locale to the US one, and input a date formatted in the usual way in this country (stored in the object `today.us`). We then set the locale to the French one, and try to input a date with the exact same command but this returns `NA`. If the date is input with the syntax consistent with the French locale, the result is now correct (stored in `today.fr`).

The use of the 2-digit code for years (`%y`) must be done with care:

```
> as.Date("68-01-01", "%y-%m-%d") # 21st century
[1] "2068-01-01"
> as.Date("69-01-01", "%y-%m-%d") # 20th century
[1] "1969-01-01"
```

The (generic) function `format` does the reverse operation than `as.Date` by printing a date as a character string using the format specified by the same coding as in Table 5.2:

```
> format(z, "%d/%m/%Y") # FR style
[1] "31/12/1969" "01/01/1970" "02/01/1970"
> format(z, "%m-%d-%Y") # US style with dash separator
[1] "12-31-1969" "01-01-1970" "01-02-1970"
```

```
> format(z, "%Y") # only the year
[1] "1969" "1970" "1970"
```

When reading years (i.e., under the `"%Y"` format code), only values ranging from `"0"` to `"9999"` are accepted.[2] Note that R counts a "year zero", so the day before 1st January 1 is 31st December 0:

```
> format(as.Date("1-01-01") - 1, "%d %B %Y")
[1] "31 December 0"
```

We can add or subtract numbers to a "Date" object in order to define dates outside of the limits defined when reading dates (0–9999), so the day before 1st January 0 is 31st December −1:

```
> format(as.Date("0-01-01") - 1, "%d %B %Y")
[1] "31 December -1"
```

This is year 2 BCE in the Gregorian calendar which has no year zero. Considering the calendar system changes through History, some care must be taken when handling dates depending on the context of the research.

Finally, we note that the year is manipulated correctly if a large number of days is subtracted or added to a date:

```
> format(as.Date("0-01-01") - 1e9, "%d %B %Y")
[1] "28 December -2737908"
> format(as.Date("0-01-01") + 1e9, "%d %B %Y")
[1] "04 January 2737907"
```

The coding of times follows the same logic than with dates but, obviously, with the additional difficulty of recording the time in addition to the date. There are two classes in R: "POSIXct" and "POSIXlt" which differ only in the way information is stored. To illustrate this, we take the current time from the system and do a few operations:

```
> z <- Sys.time()
> class(z)
[1] "POSIXct" "POSIXt"
> storage.mode(z)
[1] "double"
> zlt <- as.POSIXlt(z)
> class(zlt)
[1] "POSIXlt" "POSIXt"
> storage.mode(zlt)
[1] "list"
```

[2] Leading zero's are accepted as long there are no more than four digits, so `"0"`, `"00"`, `"000"`, and `"0000"` are all equivalent, but `"00000"` gives an error.

Objects of class "POSIXct" store the number of seconds since the beginning of 1970 in the UTC time zone. Objects of class "POSIXlt" store more components in a list:

```
> attributes(zlt)
$names
 [1] "sec"    "min"    "hour"   "mday"   "mon"    "year"
 [7] "wday"   "yday"   "isdst"  "zone"   "gmtoff"

$class
[1] "POSIXlt" "POSIXt"

$tzone
[1] ""    "+07" "+07"
```

By contrast, the same time in the "POSIXct" class has only its class as additional attribute:

```
> attributes(z)
$class
[1] "POSIXct" "POSIXt"
```

But both classes when printed look very similar:

```
> z
[1] "2021-07-21 18:49:44 +07"
> zlt
[1] "2021-07-21 18:49:44 +07"
```

Unlisting the "POSIXlt" object shows more clearly its elements:

```
> is.list(zlt)
[1] TRUE
> is.list(z)
[1] FALSE
> unlist(zlt)
                sec                 min                hour
"44.7480847835541"                "49"                "18"
               mday                 mon                year
               "21"                 "6"               "121"
               wday                yday               isdst
                "3"               "201"                 "0"
               zone              gmtoff
              "+07"             "25200"
```

Both classes return the same value when converted with `as.numeric`:

```
> as.numeric(z)
```

```
[1] 1626868185
> as.numeric(zlt)
[1] 1626868185
```

Conversions from character strings follow the same mechanism than with `as.Date` with the additional option `tz` (time zone):

```
> x <- as.POSIXct("1970-01-01 00:00:00",
+                 format = "%Y-%m-%d %H:%M:%OS",
+                 tz = "GMT")
> x
[1] "1970-01-01 GMT"
> as.numeric(x)
[1] 0
> z - x
Time difference of 19026.73 days
```

5.4 Numerical Precision

The basic issue of numerical precision is that there are an infinite number of numbers while the computing resources used to code a number (the number of bits) are necessarily limited. Say that n bits are used to code a number: then the number of numbers that can be represented is at most 2^n.[3] This has several consequences:

1. There is value (say α) so that numbers which are smaller cannot be represented.
2. Similarly, there is a value (say ω) so that numbers which are larger cannot be represented.
3. For real numbers, there is an infinite number of numbers in a finite interval, so that only a portion of them can be represented.

Consequence 3 obviously does not apply to integer numbers.

There are a lot of ways to code numbers in computers (we will come back to this in Sect. 8.3), but this kind of details is usually of little concern when analysing data. In particular, users often do not need to bother whether '1' should be considered as an integer number or a real number. On the other hand, the difference matters in many internal computations.

R has two basic numeric types: 32-bit integer and 64-bit floating-point reals. As implied by their names, storage of a number under each type requires four or eight bytes, respectively. There are fundamental differences between these two types (Fig. 5.3). The first difference is trivial: integers are a discrete

[3] This limitation is not specific to numbers: see Section 4.3.1 on encodings. The ASCII encoding is based on seven bits, so that at most $2^7 = 128$ characters can be represented.

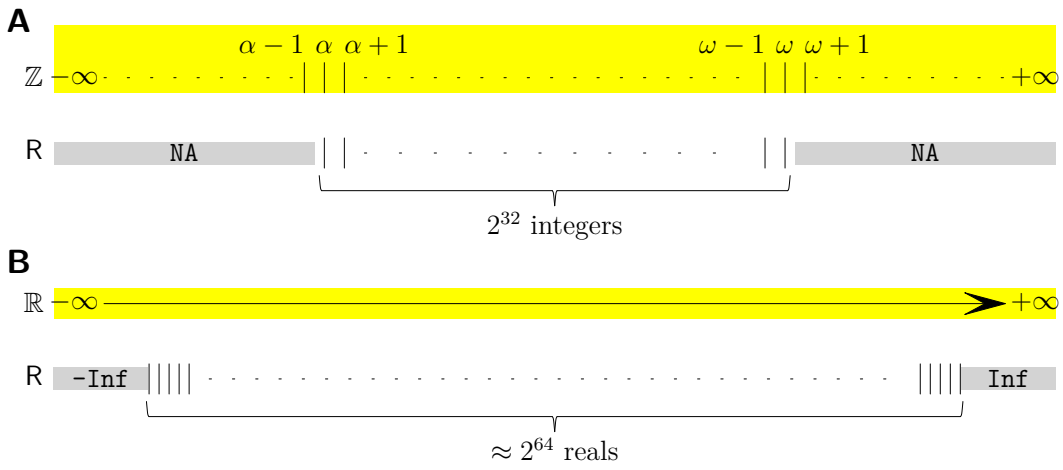


Fig. 5.3. (A) The set of integer numbers ($\mathbb{Z}$) on yellow background and how they are represented in R. (B) The set of real numbers ($\mathbb{R}$) on yellow background and how they represented in R. The grey bands show the "out-of-range" numbers.

set which makes possible a one-to-one match between them and their binary representation for all numbers between α and ω. Put more simply: there is no number between α and $\alpha + 1$, or between $\omega - 1$ and ω. On the other hand, a finite interval contains an infinite number of real numbers and only some of them can be represented on a computer.

For integers, the largest possible value in R is $\omega = 2\,147\,483\,647$ $(= 2^{31} - 1)$ and is stored in a list:

```
> .Machine$integer.max
[1] 2147483647
```

The smallest possible is symmetric around zero: $\alpha = -\omega$.

There are two main ways to force a number to be stored as an integer: either with the function `as.integer`,[4] or by suffixing it with `L`:[5]

```
> str(1)
 num 1
> str(1L)
 int 1
> 1 == 1L
[1] TRUE
> identical(1, 1L)
```

[4]The function `storage.mode` has the same effect.

[5]This syntax originates from older systems when integers were commonly coded on 16 bits, so that 32-bit integers were considered as "Long". Nowadays, most systems code standard integers on 32 bits, and long integers on 64 bits.

```
[1] FALSE
```

It is interesting to note that values out of range (i.e., $< \alpha$ or $> \omega$) are not treated similarly with respect to these two ways; for instance, if we want to force the storage of the number $\omega + 1$ as integer, the result will depend on the method used:

```
> as.integer(2147483648)
[1] NA
Warning message:
NAs introduced by coercion to integer range
> 2147483648L
[1] 2147483648
Warning message:
non-integer value 2147483648L qualified with L; using numeric
    value
```

We can now examine some of the practical side-effects of the binary coding of numbers. For instance, 10^{500} is a number that we can write with '1' followed by 500 '0', so it's certainly not the infinity (∞) and we can write the mathematical inequality $10^{500} < \infty$. But in R, we have:

```
> 10^500 < Inf
[1] FALSE
> 10^500
[1] Inf
```

This number is (much) larger than the largest representable real number (the equivalent of ω for integers) which is stored in the same list as above:

```
> .Machine$double.xmax
[1] 1.797693e+308
```

Like for integers, the smallest representable value is symmetric to this one: all values smaller than to it (or larger to the above one) are represented as `-Inf` (or `Inf`). A number divided by zero is by definition ∞, so that we have the (mathematically wrong) equality:

```
> 10^500 == 1/0
[1] TRUE
```

Another value stored in the list `.Machine` is the smallest value which is larger than zero:

```
> .Machine$double.xmin
[1] 2.225074e-308
```

For instance, 10^{-300} is written with '0.[299 zeros]1', so it is a very small number but surely greater than zero:

```
> 10^-300 > 0
[1] TRUE
```

The number 10^{-330} is even smaller than the previous one but still greater than zero; however:

```
> 10^-330 > 0
[1] FALSE
```

10^{-330} is among the many numbers (actually an infinity of them) which are not representable in a binary coding, so the "closest" representable number is used instead (zero in this case).

All these subtleties can have even more intruiguing results:

```
> 1.2 - 0.8 == 0.4
[1] FALSE
```

It happens that 0.4 is not representable, so depending on the way this number is computed, different representable numbers may be returned. Note that this is not rounding error; here is an example of the latter:

```
> x <- rnorm(10)
> y <- round(x, 6)
> all.equal(mean(x), mean(y)) # this is rounding error
[1] "Mean relative difference: 7.794554e-08"
> all.equal(mean(x), mean(y), tolerance = 1e-6)
[1] TRUE
```

Take a moderately large number such as 10^{16} and add one to it: surely the result will be greater than the this number because $10^{16} < 10^{16} + 1$. But:

```
> 10^16 < 10^16 + 1
[1] FALSE
```

Let's take the (slightly) smaller number 10^{15}:

```
> 10^15 < 10^15 + 1
[1] TRUE
```

We may add one several times to 10^{16}, this will not change the final result:

```
> 10^16 == 10^16 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
[1] TRUE
```

This is because the additions are performed sequentially. If we ask R to sum the one's first and add this them to 10^{16}, then:

```
> 10^16 == 10^16 + (1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1)
[1] FALSE
```

It some cases it is possible to "catch" this kind of issue before it can occur and so avoid it: this is the case when calculating $\ln(1+x)$ when $|x| << 1$. In this calculation, the result is close to but not equal to zero:

```
> log(1 + 1e-8)
[1] 1e-08
> log(1 + 1e-18)
[1] 0
```

The last result is expected to be 10^{-18}. The function `log1p` gives the correct result:[6]

```
> log1p(1e-8)
[1] 1e-08
> log1p(1e-18)
[1] 1e-18
```

This last value will still be "ignored" if added to a large value (e.g., 1), but it can be used to compute ratios. For instance, the next two operations should logically give the same result:

```
> log1p(1e-18) / log1p(1e-17)
[1] 0.1
> log(1 + 1e-18) / log(1 + 1e-17)
[1] NaN
```

5.5 Exercises

1. Figure 5.1 could have an additional arrow. Explain why this is obvious.

2. Build the following expression `e <- expression(x <- x + 1)`. Then, run the command `eval(e)`. Do you expect an error? Explain your answer.

3. Run the command `plot(0, 0, "n")`. Add the annotation $\sqrt{2\pi}$ at the center of the plot.

4. Find the second partial derivative of the logarithm of x (i.e., $\partial^2 \ln x / \partial x^2$) using the function `D`.

5. Build the formula `y ~ x1 + x2`. Explain why no addition is made when building this formula.

6. Read a date written in the standard US format "01/31/2022" into an object of class `"Date"`.

[6] A similar function is available in most computer languages, usually with the same name.

7. Below are the five top rows of a file:

```
Year   Month   Day
2019    Feb    12
2020    May    15
2021    Nov    14
2021    Dec    14
```

   How would you convert these data into the class `"Date"`?

8. You have data including dates marked with either "BCE" or "AD" (the latter could be "CE" in a more recent notation). What special care (and eventually manipulation) you should take when calculating time intervals with these data?

9. Type the following command in R: `1e400 < 1e500`. Explain the results using a figure.

10. Explain the following result:

```
> A <- 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 2^52
> B <- 2^52 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1
> A > B
[1] TRUE
```

    What if instead of adding six times 0.1, we would add five times this same number?

    Show (and explain) how parentheses could lead to a more expected result.

11. The mass of a blue whale is about 30 tons and the mass of a bacterium is about 1 pg ($= 10^{-12}$ g). Logically a whale and a bacterium are heavier than a whale alone: how can you show this using `log1p`? (Reminder: 1 ton = 1 Mg = 10^6 g.)

12. Try the following command:
    `(1.2 - 0.8) * 1e16 == 0.4 * 1e16 - 1`
    Explain the results.

13. Compare these two commands and explain the results:
    `sqrt(2)^2 == 2`
    `sqrt(2^2) == 2`

6

Debugging

After a script or a function has been written, it is not rare that some issues happen: the results may not look good, some error (or warning) messages may occur, or other unexpected outputs. There are usually two main causes for these:

- Some errors have been inserted in the code: typos, wrong function names, misplaced parentheses, ...;
- The input data have some features that were not expected or planned by the author of the code.

In practice, errors happen all the time, and there is a simple way to help to solve them: *read the message printed when the error occurs*. Error messages may seem obscure to beginners, but they must be able to make sense of them. This is particularly true when considering that, in my experience, the majority, maybe around 50%, of errors are due to typing errors which are easy to correct. Another large portion of errors, maybe around 40%, are due to trivial errors or mistakes which are also easy to fix. Thus, it seems that around 90% of errors in R code are straightforward to solve. We examine in this chapter tools to help solve the remaining 10%.

6.1 Strategies to Avoid Errors

We see in this section a few rules that help to avoid errors when writing R code.

Write lines of code as simple as possible.—This rule could be stated as: *Several short lines of code are better than a long one.* For example, suppose we have a data frame `DF`, then the line:

```
1 for (i in 1:nrow(DF)) ....
```

can, in many cases, be replaced by:

```
n <- nrow(DF)
for (i in 1:n) ....
```

The human eye is more able to find errors in short than in longer lines. Besides, it is very common that `n` will be needed elsewhere in the code (e.g., `if (n < 10) warning("not enough observations to compute SEs")`.

Test code progressively.—It is (very) discouraging to write many lines of code and finding out that they produce an error. A lot of these errors can be avoided by testing progressively the code, maybe even each line individually. Not only this makes to possible to find and fix errors progressively, but it is stimulating for the user to see that the code under progress is working as expected.

Use your favourite IDE.—We have seen in the first chapter that there are several ways to interact with R and choosing a specific interface is a matter of personal choice. Most of these interfaces (also called integrated development environments, IDE) have tools to help developers such as highlighting the matching parentheses or brackets. So choose one and use it when developing your code.

6.2 Interactive Execution of Functions

From Section 3.1, we have seen that when a function is called, a new environment is created where the objects manipulated inside the function are stored. Once the function call is finished, with or without error, this environment is deleted; thus, in principle, it is not possible to examine these objects. This is actually possible with the function `debug`.

Let us say we calculate Euclidean distances and we want to see some details of how this is done by the function `dist`. We first put this function in "debug mode":

```
> debug(dist)
```

Then calling this function will open a browser and the calculations are done interactively under the user's control:

```
> dist(1:0)
debugging in: dist(1:0)
debug: {
    if (!is.na(pmatch(method, "euclidian")))
        method <- "euclidean"
    METHODS <- c("euclidean", "maximum", "manhattan", "canberra"
        ,
        "binary", "minkowski")
....
```

Table 6.1. Commands of the R function browser.

Command	Meaning
`c`	continue the commands of the call
`f`	finish execution of the current loop/function
`n`	next command (steps over function calls)
`s`	next command (steps into function calls)
`Q`	quit

```
}
Browse[2]> n
debug: if (!is.na(pmatch(method, "euclidian"))) method <- "
    euclidean"
Browse[2]>
```

where `n` means "next". Inside the browser, a few specific commands can be used and are given in Table 6.1. This implies that if an object within the function's environment has one of these names (`n` is often used for the number of observations), one must use the function `get` to see its contents (e.g., `get("n")`).

The reverse function of `debug` is `undebug` which cancels the effect of the former:

```
> undebug(dist)
```

A nice feature of `debug()` is that it also works with non-exported (or hidden) functions of a package. These functions are not directly called by users so that, in principle, it is not possible to see what is being computed inside them. However, debugging can be done on such functions if the function name is prefixed with the name of the package and the triple-colon operator `:::`.[1] As a hypothetical example, suppose we want to see the calculations done when plotting a hierarchical clustering previously computed by `hclust()`: it is easy to find that this function returns an object of class "hclust", so logically the function to plot it is `plot.hclust()`:

```
> plot.hclust
Error: object 'plot.hclust' not found
```

It appears this method is not exported. The function `getAnywhere` (from the package utils) is able to find an object from any environment (see Sect. 3.1):

```
> getAnywhere("plot.hclust")
A single object matching 'plot.hclust' was found
It was found in the following places
```

[1] The triple-colon operator is somewhat similar to the double-colon operator `::`, but the latter can only access exported objects.

```
  registered S3 method for plot from namespace stats
  namespace:stats
with value
....
```

The function is thus in the package stats. The next command will open the debugger every time an object of class "hclust" is plotted:

```
> debug(stats:::plot.hclust)
```

The following is an example with simulated data:

```
> hc <- hclust(dist(rnorm(5)))
> plot(hc)
debugging in: plot.hclust(hc)
debug: {
    merge <- x$merge
    ....
}
Browse[2]> ls()
 [1] "ann"         "axes"        "check"       "frame.plot" "hang"
 [6] "labels"      "main"        "sub"         "x"          "xlab"
[11] "ylab"
Browse[2]> x

Call:
hclust(d = dist(rnorm(5)))

Cluster method   : complete
Distance         : euclidean
Number of objects: 5

Browse[2]>
```

6.3 Using Standard Tools

Instead of using `debug()`, it may be simpler to use standard R functions to access the contents of the objects within a function's environment. This can be useful for code developed by the user.

- The functions `print`, `sprintf`, or `cat` can be inserted in a function: they will be executed normally and the contents will be displayed while the function is executed. `cat()` can print its argument(s) in a file.
- The superassignment operator `<<-` (Sect. 3.3.2) can be inserted in a function and the object on the left-hand side will be returned in the global environment.

- The function `browser` can be inserted within a function: when it is executed, the debugger (see previous section) is open and the user can examine the objects in the environment of the function.

6.4 Catching Errors

When an error occurs within a function, a message is printed, the execution is stopped and nothing is returned. This means that the environment of the function is lost and it is therefore impossible to know the values or contents of the objects inside. The (global) option called `"error"` is a way to avoid this problem:

```
> options("error")
$error
NULL
```

Apart from the default `NULL`, this global option can take two choices:

1. `options(error = recover)` which makes possible to explore the environments successively opened until the error.

2. `options(error = dump.frames)` saves an object named `last.dump` in the global environment which can be explored with the function `debugger`.

We try the first procedure with a simple function:

```
> foo <- function(x) {
+     x <- log(x)
+     if (anyNA(x)) stop("NA not allowed")
+     sum(x)
+ }
```

It is pretty obvious that an error will occur if negative value(s) are passed with the argument `x`:

```
> foo(runif(10)) # error impossible
[1] -89.24168
> foo(rnorm(10)) # error expected in 50% of values
Error in foo(rnorm(10)) : NA not allowed
In addition: Warning message:
In log(x) : NaNs produced
```

As explained above, the values of `x` within `foo()` are lost and it is impossible to know the details of what happened, for instance, how many values were missing. Let's try now to access these values after setting the option `error` as explained above:

```
> options(error = recover)
> foo(rnorm(10))
Error in foo(rnorm(10)) : NA not allowed
In addition: Warning message:
In log(x) : NaNs produced

Enter a frame number, or 0 to exit

1: foo(rnorm(10))

Selection:
```

There is a single environment open (called "frame" here), so we select it and print its contents:

```
Selection: 1
Called from: top level
Browse[1]> ls()
[1] "x"
Browse[1]> x
  [1]         NaN          NaN -0.078739513          NaN
  [5]         NaN -0.517230009 -4.940709077 -1.798025933
  [9] 0.331829889 -0.532842058
Browse[1]>
```

Since this procedure is interactive, it is possible to use usual R commands on the listed objects:

```
Browse[1]> table(is.na(x))

FALSE  TRUE
    6     4
```

When the exploration is finished, we can quit the browser:

```
Browse[1]> Q
```

It may be useful to reset the option `error` after the diagnostic has been completed:

```
> options(error = NULL)
```

This example is simplistic in the hope that it helps to understand the mechanism of the procedure. Nevertheless, two points can be made of it. First, it is possible to test a code or a function with simulated data in a way that no problem is encountered. In the first test of `foo()`, we used random uniform variates which cannot be negative, so that we can repeat the command `foo(runif(10))` as many times we want without getting any error. In real

applications, this problem can happen, for instance, with incorrect data in files. Second, in the present example, the successive function calls (what is called the call stack; see next chapter) is simple: `log()` is called by `foo()`. But in real applications, this is rarely as simple. For instance, if a function calls `lm()`, then this last one calls a number of other functions so that if any error occurs, it might be not obvious to know in which of these it happened.

Finally, we note the functions `traceback()`, `recover()`, and `debugger()` which give information on the last error.

6.5 Exercises

1. Type the three following commands respecting the (lack of) spaces:

```
x<-rnorm(10)
x<-1
x<+1
```

   Explain the results and comment on the good practice of writing R code.

2. Matching parentheses (or brackets, or braces) are common problems when writing R code. How to avoid these problems?

3. Suppose we want to execute the sum of a vector step-by-step. We first run `debug(sum)`, then `sum(rnorm(1000))`. Explain what you observe.

4. Suppose you wrote a script with 50 lines of commands; you then try to run your script with `source()` and an error occurred. What is the best strategy to solve this problem?

5. Explain why the functions `mean`, `var`, `median`, `quantile`, `max`, `min`, `range`, and `prod` may lead to errors when missing values are likely to be present in the data. Propose two ways to handle this problem.

6. Give some examples of data checking you could use in your code to avoid errors.

7. You are writing a method associated to a generic function (see Sect. 3.5): is it useful to check the class of the input data?

8. A common situation when programming a general purpose function is that no data in a vector (or a data frame) meet some requirement(s). How would you test for this and return a message to the users?

9. What attribute(s) would you test to check the data type input to a function?

10. Why is it useful to reset the global option `options(error = NULL)` after finishing to debug a function?

7

Performance Optimisation

7.1 Background

There are several basic rules to keep in mind when attempting to optimise code or improve its performance:

Rule #1: *The correctness of the results is more important than the running time.* This rule implies that there is no need to try to optimise the performance of a function as long as its development is still in progress—and this chapter comes after the chapter on bug fixing. For instance, if the developer is still uncertain about the structure of the return value of a function, or some algorithmic details, these issues should be solved before trying to make the code faster.

Rule #2: *The performance gains are inversely proportional to the time spent on improving the code.* This rule is well-known by computer programmers. In other words, the more time you spent on making a function faster, the less worth the effort. One reason for this is because code performance needs to be measured carefully. The most important factor affecting the performance of computer code (besides the code itself) is data size.

Data size is itself made of several components: the number of observations, the number of variables, and the structure (or patterns) within the data. Whether the performance of a function is affected by one or several of these components depends on the computational methods implemented. For instance, the running time of the singular value decomposition (SVD) of a matrix is much more affected by the number of columns than by the number of rows. Thus, this will affect the performance of the principal component analysis (PCA) done by SVD as implemented in `prcomp()`. On the other hand, the classical PCA by eigendecomposition (implemented in `princomp()`) is done on the variance-covariance matrix so that the number of observations will be

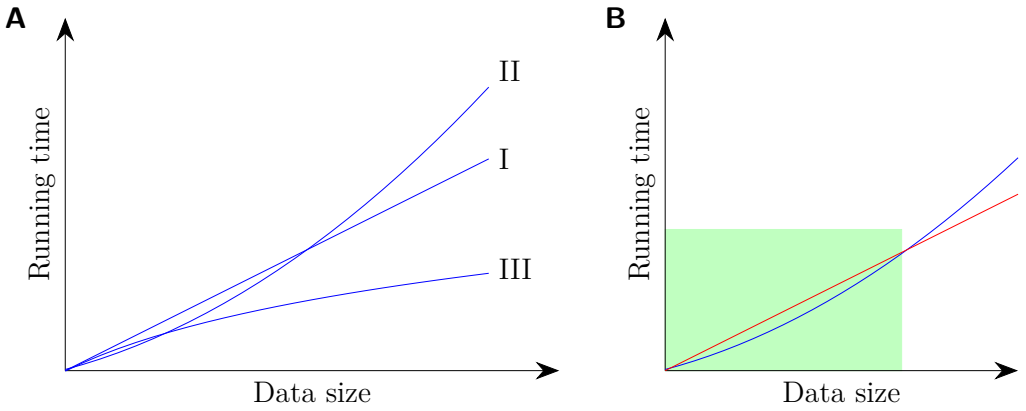


Fig. 7.1. (A) The three types of relationship between data size and running time. (B) Performance improvement may depend on data size (see text for details).

important only when computing the VCV matrix.[1]

Quite logically, running time increases with data size: there are three main types of relationship between these two quantities (Fig. 7.1A): linear (I), 'exponential' (II), and 'logarithmic' (III). The first two are the most common. It must be kept in mind that the shapes of these curves are important, but their positions on this graph are also crucial and there is no way to find them except by measuring the running times for a range of data sizes. Figure 7.1B shows a hypothetical example with the running times of two solutions (functions, algorithms, scripts, or else). Because one solution (in red) if 'linear' (type I), and the other one (in blue) is 'exponential' (type II), the former may seem better than the latter. However, if the typical data size in practice is relatively reduced (as shown with the green rectangle), this difference might not lead to a decisive improvement.

R has several functions that run in times independent of data size, such as `length`, `mode`, or `class`, because these attributes are stored separately from the data (see Chap. 2). So checking the data with these functions is much faster than checking, say, the missing values which requires to scan all the data.

`system.time()` is the main tool to assess running times of a function, code, or script. It can be used to measure the running time of a single command, a block of commands, or a script which is given as the main argument:

```
> n <- 1e6
> x <- rnorm(n)
> y <- rnorm(n)
> system.time(z <- x + y)
   user  system elapsed
```

[1] `princomp()` will not run if the number of columns is greater than the number of rows.

```
  0.002   0.000   0.002
> system.time(for (i in 1:n) y[i] <- x[i] + y[i])
   user  system elapsed
  0.084   0.000   0.084
> identical(y, z)
[1] TRUE
```

Usually, only the third value returned by `system.time()` is of interest, particularly it is the most comparable among different machines and OSs. This function calls `proc.time()` which returns five values, although only three are printed by default; for instance, with a session started a bit more than 22 min ago:

```
> a <- proc.time()
> a
    user   system  elapsed
   5.055    0.212 1364.769
> a[] # or unclass(a)
 user.self   sys.self    elapsed user.child  sys.child
     5.050      0.211   1364.769      0.005      0.001
```

Unlike `system.time()`, this function takes no argument. The times are counted from the start of the current R session. Even if only the value under `"elapsed"` is generally the one of interest, it is good to know what these five measures are:

`user.self`: the CPU time used for executing the user's instructions during the session;

`system.self`: the CPU time used for executing the system instructions during the session;

`elapsed`: the total time of the session;

`user.child`: the cumulated times of the child processes initiated by the user's instructions during the session;

`syst.child`: the cumulated times of the child processes initiated by the system instructions during the session.

All but the third one depend on how the OS implements computing times. The resolution is also OS-dependent: it is typically 1 ms, except for Windows where it is 10 ms.

`system.time()` simply calls `proc.time()` before and after executing the code given as argument, and returns the difference.

The above examples take very little system time because the call to `rnorm()` and the other operations mostly do not need to call the system. On the other hand, accessing files (reading or writing data on the hard disk) call the computer OS. For instance, we write a file with a single value and repeat the operation one thousand times:

```
> system.time(for (i in 1:1e3) saveRDS(0, "tmp.rds"))
   user  system elapsed
  0.041   0.065   0.116
> unlink("tmp.rds") # clean-up
```

In this example, more than 50% of the operation was done by the system. Besides, this value varied substantially if the above is repeated several times, maybe because the system needs to find free space on the hard disk to perform the command.

It is worth noting here that the running time of file accessions is expected to vary considerably with the OS and, particularly, with the file system (the way the OS manages the hard disk space, also known as disk formatting). This should be kept in mind if the code implies a lot of file writing and/or reading.

Rule #3: *When measuring running times, everything is relative.* Many factors influence the running times of a code, so that a code may be fast in a situation but could be slow in another, and this is generally difficult to predict.

7.2 `Rprof`

The analysis of a data set is often made of successive steps implying several function calls. Besides, a function sometimes calls several other functions (many of them from the base package). In this situation, and assuming that Rule #1 is respected, it is useful to know where the code spends time running. This can be done with the function `Rprof`.

The idea of `Rprof()` is to sample at regular time intervals (0.02 s by default) the *call stack* made of the successive calls of a function which calls another function which itself calls another functions, and so on. During this profiling procedure, the call stack is written into a file ('Rprof.out' by default). Knowing that each row of this file is separated by a fixed time interval, it is possible to infer how much time R has spent on each function call during the execution of the code.

Of course, this is just an estimation of these times, not an actual measure. If a function execution takes less than the specified sampling interval, then it is possible that this would be missed by `Rprof()`. It is possible to decrease the sampling interval, but this has the consequence of slowing down R because the execution is effectively stopped during sampling of the call stack.

To illustrate the use of `Rprof()`, we generate a random matrix with $n = 10^5$ rows and $p = 100$ columns:

```
> n <- 100000
> p <- 100
> X <- matrix(rnorm(n * p), n, p)
```

We are now interested in profiling the PCA by eigendecomposition which is straightforward using the default options:

```
> Rprof()
> pca.eig <- princomp(X)
> Rprof(NULL)
```

The most direct way to analyse the output is to call `summaryRprof()` which prints a summary of the function calls sorted "by self" and "by total":

```
> summaryRprof()
$by.self
                    self.time self.pct total.time total.pct
"aperm.default"          0.56    39.44       0.56     39.44
"array"                  0.30    21.13       0.30     21.13
"%*%"                    0.26    18.31       0.26     18.31
"crossprod"              0.10     7.04       0.10      7.04
"princomp.default"       0.04     2.82       1.42    100.00
"sweep"                  0.04     2.82       0.90     63.38
"is.data.frame"          0.04     2.82       0.04      2.82
"cov.wt"                 0.02     1.41       0.38     26.76
"colSums"                0.02     1.41       0.06      4.23
"all"                    0.02     1.41       0.02      1.41
"apply"                  0.02     1.41       0.02      1.41

$by.total
                    total.time total.pct self.time self.pct
"princomp.default"        1.42    100.00      0.04     2.82
"princomp"                1.42    100.00      0.00     0.00
"sweep"                   0.90     63.38      0.04     2.82
"aperm"                   0.86     60.56      0.00     0.00
"scale.default"           0.72     50.70      0.00     0.00
"scale"                   0.72     50.70      0.00     0.00
"aperm.default"           0.56     39.44      0.56    39.44
"cov.wt"                  0.38     26.76      0.02     1.41
"array"                   0.30     21.13      0.30    21.13
"%*%"                     0.26     18.31      0.26    18.31
"crossprod"               0.10      7.04      0.10     7.04
"colSums"                 0.06      4.23      0.02     1.41
"is.data.frame"           0.04      2.82      0.04     2.82
"all"                     0.02      1.41      0.02     1.41
"apply"                   0.02      1.41      0.02     1.41
"fix"                     0.02      1.41      0.00     0.00

$sample.interval
[1] 0.02

$sampling.time
```

```
[1] 1.42
```

We will come on these results below.

It is also instructive to open the file 'Rprof.out' and examine its contents; this also helps to understand the previous output. The first lines of this file are:

```
sample.interval=20000
"princomp.default" "princomp"
"princomp.default" "princomp"
"all" "cov.wt" "princomp.default" "princomp"
"is.data.frame" "colSums" "cov.wt" "princomp.default" "princomp"
"is.data.frame" "colSums" "cov.wt" "princomp.default" "princomp"
"colSums" "cov.wt" "princomp.default" "princomp"
"array" "aperm" "sweep" "cov.wt" "princomp.default" "princomp"
.... [etc]
```

The first line gives the sampling interval (in µs), then from the second line the successive call stacks are printed on separate lines. For each of these lines, the leftmost character string is the function running on the top of the stack. So that, this can be interpreted as follows:

- After 0.02 s, `princomp.default()` was running after being called by `princomp()`.
- After 0.04 s, the call stack was the same than after 0.02 s.
- After 0.06 s, `all()` was running after being called by `cov.wt()` itself called by `princomp.default()` itself called by `princomp()`.
- After 0.08 s, `is.data.frame()` was running after being called by `colSums()`, etc...
- ...

The summary statistics named "by.self" are the percentages of the functions appearing on the leftmost position of the lines, whereas "by.total" are the percentages of those appearing anywhere on each line. Since we only called `princomp`, this function has therefore a "by.total" of 100%. The same percentage is observed for `princomp.default` since `princomp` is a generic function and the call of the method is shorter than 0.02 s.

A close examination of the above output from `summaryRprof()` shows something quite remarkable: the function `eigen` does not appear. This is likely due to the relatively small size of the VCV matrix (100 rows × 100 columns). This explanation can be validated by increasing the number of columns (see Exercises).

We now turn on the the same PCA analysis but using SVD as implemented in `prcomp()`:

```
> Rprof("Rprof2.out")
> pca.svd <- prcomp(X)
> Rprof(NULL)
> summaryRprof("Rprof2.out")
$by.self
                self.time self.pct total.time total.pct
"La.svd"             5.88    90.74       5.94     91.67
"%*%"                0.30     4.63       0.30      4.63
"aperm.default"      0.16     2.47       0.16      2.47
"any"                0.04     0.62       0.04      0.62
"is.finite"          0.04     0.62       0.04      0.62
"sweep"              0.02     0.31       0.20      3.09
"array"              0.02     0.31       0.02      0.31
"matrix"             0.02     0.31       0.02      0.31

$by.total
                 total.time total.pct self.time self.pct
"prcomp.default"       6.48    100.00      0.00     0.00
"prcomp"               6.48    100.00      0.00     0.00
"svd"                  5.98     92.28      0.00     0.00
"La.svd"               5.94     91.67      5.88    90.74
"%*%"                  0.30      4.63      0.30     4.63
"sweep"                0.20      3.09      0.02     0.31
"scale.default"        0.20      3.09      0.00     0.00
"scale"                0.20      3.09      0.00     0.00
"aperm"                0.18      2.78      0.00     0.00
"aperm.default"        0.16      2.47      0.16     2.47
"any"                  0.04      0.62      0.04     0.62
"is.finite"            0.04      0.62      0.04     0.62
"array"                0.02      0.31      0.02     0.31
"matrix"               0.02      0.31      0.02     0.31

$sample.interval
[1] 0.02

$sampling.time
[1] 6.48
```

This time, almost 91% of the computation is kept busy by `La.svd()` (called by the generic function `svd`). This contrasts with the previous analysis showing that three functions occupied most of the running time. Thus, in this last case, very little improvement is expected unless one has a (faster) alternative to perform the SVD.

7.3 Memory Usage

Profiling memory usage can be done with `Rprofmem()` which has options similar to `Rprof()` but returns a different output. Let us try it with the same PCA analyses than in the previous section:

```
> Rprofmem()
> pca.svd <- princomp(X)
> Rprofmem(NULL)
> Rprofmem("Rprofmem2.out")
> pca.svd <- prcomp(X)
> Rprofmem(NULL)
```

`Rprofmem()` writes a file with each line recording each memory allocation with its size and the call stack. For instance, the first five lines of 'Rprofmem.out' are:

```
400048 :"rep_len" "princomp.default" "princomp"
400056 :"princomp.default" "princomp"
400048 :"princomp.default" "princomp"
448 :"princomp.default" "princomp"
80000048 :"princomp.default" "princomp"
.... [etc]
```

It is not so straightforward to read such a file because the number of items per line is not the same for all rows. A solution is to use `read.table` with the option `sep = ":"` so that the first item on each row will be read as a numeric vector and the quoted character strings will be read together as a single character vector:

```
> df.eig <- read.table("Rprofmem.out", sep = ":")
> str(df.eig)
'data.frame': 347 obs. of  2 variables:
 $ V1: num  400048 400056 400048 448 80000048 ...
 $ V2: chr  "rep_len princomp.default princomp " "princomp.
    default princomp " "princomp.default princomp " "princomp.
    default princomp " ...
> df.svd <- read.table("Rprofmem2.out", sep = ":")
> str(df.svd)
'data.frame': 18 obs. of  2 variables:
 $ V1: num  848 80000048 80000048 40000048 40000048 ...
 $ V2: chr  "colMeans scale.default scale prcomp.default prcomp
    " "array aperm sweep scale.default scale prcomp.default
    prcomp " "aperm.default aperm sweep scale.default scale
    prcomp.default prcomp " "svd prcomp.default prcomp " ...
```

These show that `princomp()` required 347 memory allocations during its execution, while `prcomp()` required 18 such allocations. The sums of these vectors give the total memory used:

```
> sum(df.eig$V1) / 1e6
[1] 764.7379
> sum(df.svd$V1) / 1e6
[1] 560.4937
```

Thus the eigen-based PCA used a bit more than 760 MB while the SVD-based one used 560 MB. It is possible to look at the distributions of these allocation size to see that the first analysis implied a large number of allocations of small bits of memory (maybe involved in the computations of the variances and covariances) and only a few large allocations, whereas the second analysis required only a few large allocations (Fig. 7.2):

```
> layout(matrix(1:2, 2))
> hist(df.eig$V1)
> rug(df.eig$V1)
> hist(df.svd$V1)
> rug(df.svd$V1)
```

7.4 Some Tricks to Write Efficient R Code

Before detailing some tricks specific to R, we list some general tips that can help to write better code in most computer languages.

- Avoid repetitions by creating temporary objects.
- Avoid to write overloaded loops: loops with many commands are likely to be redundant.
- Simplify mathematical formulas (see Exercises).
- Avoid useless calculations (e.g., replace 1/4 with 0.25).[2]
- Arrange the code in an economical way.
- Add comments, this will help you to maintain the code.

7.4.1 Avoid Simple `for` Loops

If a `for` loop contains very little code, then it is likely that there is a simpler and faster alternative. A trivial example is given below:

[2]Some tools correct for these "mistakes": in R, once a function is compiled (which is done almost always except when first used), and in C with the basic optimisation options of most compilers.

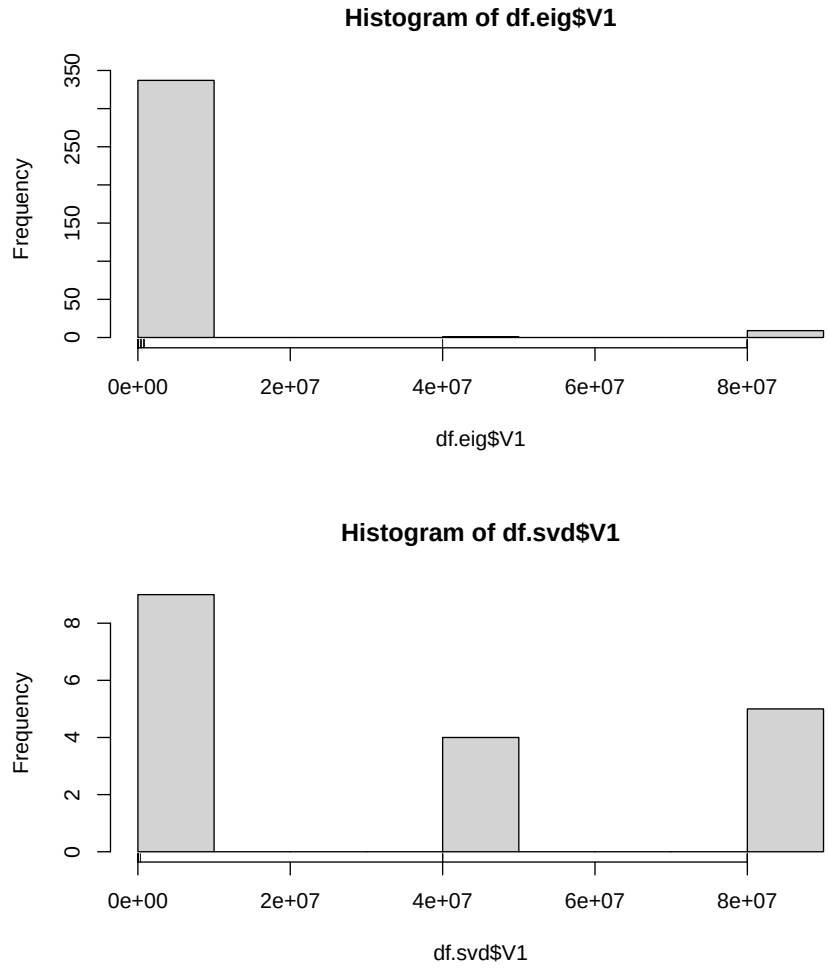


Fig. 7.2. Distributions of memory allocations when performing a PCA by eigendecomposition (top) or SVD (bottom).

```
> x <- numeric(1e6)
> system.time(for (i in 1:1e6) x[i] <- 1)
   user  system elapsed
  0.044   0.000   0.045
> system.time(x[] <- 1)
   user  system elapsed
  0.004   0.000   0.002
```

A slightly less trivial example involves a comparison which can be replaced by logical indexing:

```
> y <- x <- rnorm(1e6)
> system.time(for (i in 1:1e6) if (x[i] < 0) x[i] <- 0)
   user  system elapsed
```

```
  0.048   0.000   0.048
> system.time(y[y < 0] <- 0)
   user  system elapsed
  0.004   0.000   0.004
```

7.4.2 Prefer Numerical Indexing to Indexing with Names

Sorting numbers is faster than sorting character strings. Besides, the `names` attribute can use a significant quantity of memory resources. Substantial or important gains in performance can be achieved in a function or in a script by first arranging or sorting the data at the start so that all the rows, observations, subjects, etc, are arranged in the same way in the different data sets (avoiding to search to match the objects repeatedly later).

7.4.3 Unclass Objects

Searching for the methods of generic functions can slow down considerably some computations if indexing is heavily used in a loop. The reason for this is because the indexing operator is a generic function:

```
> methods("[")
 [1] [.acf*                  [.AsIs
 [3] [.bibentry*             [.data.frame
....
```

So that the command `x[1]` first checks the class of `x` and call the appropriate method as described in Section 3.5. As an example, we build a function to calculate the sum of each column of a table `x`:

```
> f <- function(x) {
+     p <- ncol(x)
+     res <- numeric(p)
+     for (i in 1:p) res[i] <- sum(x[, i])
+     res
+ }
```

We try this function with a data frame containing 1000 rows and 10 000 columns:

```
> DF <- as.data.frame(matrix(rnorm(1e7), 1e3, 1e4))
> system.time(of <- f(DF))
   user  system elapsed
  0.144   0.000   0.147
```

Now suppose we have written a function `g` which is similar to `f` but where the argument `x` has been unclassed:

```
> system.time(og <- g(DF))
   user  system elapsed
  0.052   0.000   0.052
> identical(of, og)
[1] TRUE
```

The result is identical to the one returned by `f()` but the running time is divided by three. In some other applications, the difference can be more substantial (10 times or more). It is up to you to find the code of `g` (see next).

7.5 Exercises

1. Write down a list of the data analysis methods that you use commonly (e.g., correlation, ANOVA, PCA, and so on). Try to associate each of these methods to the curve of type I, II, or III displayed on Fig. 7.1A.

2. Suppose we have a sample of n observations from a standard uniform distribution $\mathcal{U}(0,1)$. The largest value of the sample is a random variable with variance given by the two mathematically identical expressions:

$$\frac{n}{n+2} - \left(\frac{n}{n+1}\right)^2 = \frac{n}{(n+2)(n+1)^2}.$$

   Which one should be preferred to code in a computer program?

3. The k-means method is an unsupervised classification method that finds structure (grouping) from continuous data (see `?kmeans`). Assess the running time of this method with 10^6 observations from two normal distributions with means -2 and 2 (with equal sample size in each group). Repeat the same analysis but with all observations drawn from a normal distribution with mean zero.

4. Repeat the comparison between the functions `princomp` and `prcomp` using `Rprof()` but this time setting $n = 10^4$ and $p = 1000$. Comment on the differences with the results in this chapter.

5. Explain why it is common that the code of functions include the command `n <- length(x)`. Give other examples of similar commands.

6. You need to run simulations that are expected to take several days. The output of each simulation replicate will then be analysed with a short R script that you have downloaded from Internet. After looking at the code of the script, you realise that the code can be easily improved to make it faster. What is your decision?

7. Build a matrix with random values of your choice with n rows and p columns. Evaluate the running times for several values of n and p. Represent the results graphically.

8. Do performance profiling of the command `x <- rnorm(1)`. Explain the observed results. Repeat this analysis with larger values passed to `rnorm`. Which option of `Rprof` you may need to adjust and why?

9. Perform the memory profiling of the command `x <- rnorm(1e7)`. Are the results as expected?

10. Find the code of the function `g` used at the end of this chapter.

8

R–C Interfaces

The topic in this chapter require, in addition to R, to have a C compiler installed and properly configured to work with R, so that compiled C code can be used directly from R (see Table 1.2). The GNU C compiler (GCC) is certainly one of the best choices for this. For practical details relative to their specific platforms, the readers are referred to the manuals *Writing R Extensions* and *R Installation and Administration* (Appendix C: C Platform notes), both on CRAN and also installed with R (see `help.start()`).

8.1 Why Use C With R

C is often, but not always, a very good solution to improve the speed of computations in R. However, the cost is clear: the time needed to interface a C code with R can be significant. It is not easy to decide whether this is the best, or even an appropriate, strategy. Two situations seem quite common.

8.1.1 Standard R Vector Operations Cannot Be Used

Operators in R are actually functions, so avoiding to call them repeatedly is likely to result in more efficient code. For instance, take the following command aimed at replacing all missing values in a vector by the mean of its values:

```
x[is.na(x)] <- mean(x, na.rm = TRUE)
```

The indexing operator `[` is called once, which is the right thing to do here. On the other hand, it could seem more intuitive[1] to write a command based on a `for` loop, which would be slower (for the same result):

[1] It is often easier to first write R code with a `for` loop, then to vectorise it (if possible). An interpretation of this fact might be that the human brain naturally thinks element-wise or value-wise (e.g., "if a value is missing, replace it by the mean") rather than vector-wise.

```r
mx <- mean(x, na.rm = TRUE)
for (i in seq_along(x)) if (is.na(x[i])) x[i] <- mx
```

Vectorised expressions are not always possible, however, especially when doing complicated operations involving multiple comparisons for each element of a vector. Furthermore, when non-linear or recursive operations are involved, vectorisation is almost impossible. In those cases, a C program is an interesting alternative.

8.1.2 A C Program Already Exists

Recoding a complete C program into R may be expensive in terms of time and other resources. In this situation, writing an interface between R and an existing C program might be a better solution. Nevertheless, this could not be very trivial, and some care is needed as explained in this chapter.

8.2 Basics on C

The C language was introduced in the early 1970's and was an important element in the success of the UNIX operating systems. Its features made it attractive for programmers who focus on efficiency, so that C progressively superseded assembler languages.

C is a functional language: a program is made of functions which operate on data. A very wide range of elementary data types can be handled, and more complex data types can be constructed by the programmer. An essential feature of C is the manipulation of pointers; this is detailed in the next section.

C is a declarative language: all variables used in a program must be declared explicitly with their types. Finally, C is a compiled language: a program is written in a text file (ASCII-encoded) which is then compiled into an executable program. Therefore, a C program needs a compiler to be run. However, the C code may be itself (relatively or completely) OS-independent.

We can now write our first C program (in the file 'helloworld.c'):

```c
#include <stdio.h>

void main()
{
    printf("Hello World!\n");
}
```

Each line of this program can teach us something about C. On line 1, the `#include` statement informs the compiler that a function used in our program is defined in the file 'stdio.h' which is a header file: without this statement, the compiler would not know the behaviour of the function `printf`, and an error would occur. If other functions are used in a program, the appropriate header

Table 8.1. The main data types in C. All integer types can be signed or unsigned. The declarations of the form `intXX_t` are implementation-independent.

Data	Declaration		Size (bytes)
	Usual	Generic	
Integer	`char`	`int8_t`	1
	`short`	`int16_t`	2
	`int`	`int32_t`	4
	`long`	`int64_t`	8
	`long long`		16
Real	`float`		4
	`double`		8
	`long double`		16

files must be included in the same way (each on a separate line; see below for an example).

On line 3, the function `main` is defined: this is a standard name for the function which interacts with the user, although it will be called under this name (see below). Like in R, the argument(s) are listed within parentheses, and also like in R, they are optional so a C program can have no argument. The type of the returned value is a mandatory statement and is given before the name of the function: in the present case, nothing is returned, so `void` is written.

On lines 4 and 6, we see that curly braces are used in a way similar than in R to delimitate blocks of commands or instructions.

Finally, on line 5 the command calling the function `printf` is terminated by a semicolon which is required in C (in R, the semicolon can be used to separate distinct commands on the same line).

The program can now be compiled with the command (the `$` symbol is the system prompt):

```
1 $ gcc helloworld.c -o helloworld
```

And its execution gives:

```
1 $ ./helloworld
2 Hello World!
```

8.2.1 Data Types in C

C has many data types and we review only the main ones here. We start with data types for numbers which can be classified into two groups (Table 8.1):

- Integers can be of different sizes and could be either *signed* (with one

bit coding for the sign, so the value can be negative, zero, or positive) or *unsigned* (only zero and positive values).

- Real numbers (called *floating-point numbers*) also of different sizes but always signed.

Character strings are made of an array of bytes (i.e., `char` or `unsigned char`). We will see below the meaning of the word 'array' in C.

Variables must be declared in the program, for example if a function uses two integers, n and i, and a real number, x, the following lines of code must be written, usually at the beginning of the program but always before the variable is used:

```
int i, n;
double x;
```

The usual declarations (`int`, `long`, ...) are dependent on the implementation: in a modern system `long` declares a 64-bit integer, but this used to be a 32-bit integer in older systems.[2] Some declarations forces the size of the integers independently of the system and the implementation. Most compilers have built-in constants storing the limits on these different data types. We can write a small program (in the file 'intminmax.c') to print these values:

```
#include <stdio.h>
#include <stdint.h>

void main()
{
  printf("Type                  Smallest                 Largest\n");
  printf("-------------------------------------------------------\n");
  printf("int8_t    %*d  %*d\n", 20, INT8_MIN, 22, INT8_MAX);
  printf("uint8_t   %*d  %*u\n", 20, 0, 22, UINT8_MAX);
  printf("int16_t   %*d  %*d\n", 20, INT16_MIN, 22, INT16_MAX);
  printf("uint16_t  %*d  %*u\n", 20, 0, 22, UINT16_MAX);
  printf("int32_t   %*d  %*d\n", 20, INT32_MIN, 22, INT32_MAX);
  printf("uint32_t  %*d  %*u\n", 20, 0, 22, UINT32_MAX);
  printf("int64_t   %*ld  %*ld\n", 20, INT64_MIN, 22, INT64_MAX);
  printf("uint64_t  %*d  %*lu\n", 20, 0, 22, UINT64_MAX);
}
```

After compiling, the execution of the program prints:

```
$ ./intminmax
Type                  Smallest                 Largest
-------------------------------------------------------
int8_t                    -128                     127
```

[2] This explains why the 'L' suffixed to a number in R forces this number to be stored as an integer.

```
uint8_t                     0                   255
int16_t                -32768                 32767
uint16_t                    0                 65535
int32_t           -2147483648            2147483647
uint32_t                    0            4294967295
int64_t   -9223372036854775808   9223372036854775807
uint64_t                    0   18446744073709551615
```

When declared in a program, all data types can be one of the three followings:

- a scalar (a single value as in the previous example),
- an array,
- a pointer.

An *array* is a set of variables all of the same type declared with the `[` operator. It is necessary to specify the size of an array in the program so that the required quantify of memory is given when the program is executed.

A *pointer* is a variable that stores the address of a variable; pointers are declared with an `*` written between the data type and the name of the variable (sometimes with a white space between them). For instance, the two following lines declare an integer pointer `z`, a real pointer `x`, and an array of ten reals `y`:

```
int *z;
double *x, y[10];
```

The next section explains how pointers are used to allocate memory and create arrays of sizes which are not specified a priori when writing the program.

Finally, we note that C has other data types (e.g., Boolean, files, functions), and that user-defined structures can be created with `struct`, for instance the following declaration:

```
typedef struct mydata {
    int n;
    double *x;
} mydata;
```

creates a data structure with an integer `n` and a double pointer `x` and the name of this user-defined data type is `mydata`. The next lines of code create a variable `X` of the previously defined type, assign the value of 10 to its element `n`, and allocates the memory to store ten values to its second element (see next section about this last command):

```
mydata *X;
x = (mydata*)R_alloc(1, sizeof(mydata));
X->n = 10;
X->x = (double *)R_alloc(x->n * sizeof(double));
```

8.2.2 Memory and Pointers

A pointer is a variable which stores the address of a variable in the (RAM) memory of the computer. Computers use a register to find data in the RAM. The type of CPU determine the size of its register: a 64-bit CPU can address up to 2^{64} bytes. Most computers in use until the early 2000's had a 32-bit CPU. The elementary unit of memory in the register is the byte (= 8 bits).

Figure 8.1 gives an image of a few bytes of the RAM of a computer: each cell represents one byte (there are one billion such cells in 1 GB of RAM). Suppose there are five integer values that we want to analyse: they are represented as five rectangles each with four cells (recall than an integer is stored on four bytes). To access these data in C, the programmer needs to use a pointer to store the address of the first value; this pointer is named `x` and is shown on the right of the figure. This pointer occupies eight bytes, that is 64 bits, which makes sense since this variable should be able to take as many values than the limit addressable by the CPU.

Since `x` stores an address, it is possible to access these data with an operation called *dereferencing* which is performed with the operator `[`. The value given inside this operator is the number of bytes multiplied by the size of the data type under consideration that should be shifted in order to read the data. So, `x[0]` reads the first value (zero byte starting from the address stored in `x`), `x[1]` reads the second value (1×4 bytes shifted from the address in `x`), `x[2]` reads the third value (2×4 bytes), and so on. It is clear to see the similarity with the operator `[` in R, but there are important differences:

- In R, this operator is called *indexing* and can take a vector with several values and of different types (Sect. 4.4). On the other hand, dereferencing in C takes only a single integer value.
- In addition to pointers and arrays, C can manipulate scalars which are made of a single value (e.g., `y` in Fig. 8.1). On the other hand, R has no scalar: vectors can be of length zero or more.
- In R, out-of-range index values are admitted and can give different results

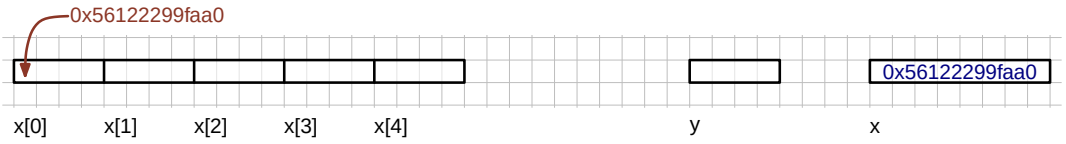


Fig. 8.1. Memory and pointers in C. Each cell delimited by thin grey lines represents one byte (= 8 bits). In red: the address of the byte pointed by the arrow. In blue: the variable `x` contains this address (so `x` is an integer pointer). In black: the variable and data manipulated by the programmer. The variables are 32-bit integers: each value takes four bytes (= 32 bits). Five values have been allocated to `*x`, whereas `y` is a scalar.

(although this may result in an error). In C, out-of-range values in dereferencing are a common source of error and must be avoided. This is further detailed below.

Because R gives the possibility to call C code, the size of pointers can be printed in R with:

```
> .Machine$sizeof.pointer
[1] 8
```

The typical use of pointers in a C program is to: (1) declare a pointer, (2) allocate some memory to this pointer. You may notice that we do not give a value to the pointer: this is done by the memory allocation operation which finds a suitable area of the active memory to store the data. To code in C what is represented in Fig. 8.1, the program will include these lines:

```
int *x, y;
x = (int *) R_alloc(5, sizeof(int));
```

The function `R_alloc` allocates the required memory with two arguments: the number of elements and the size in bytes of each element. To make this command as portable as possible, the function `sizeof` is used which returns the size of the data type given as argument. After the two above lines, the values of these two variables can be manipulated with, for instance:

```
y = 1;
x[0] = 2;
y = x[0];
```

Note that the command `y[0]` is an error (eventually found during the compilation). These operations are detailed with examples below in the context of calling C code from R.

8.2.3 Numerical Operators in C

Numerical operations in C are very similar to those in R, but slightly more complicated. Because it is very common to do simple additions (increment a value by one) or subtractions (decrement a value by one), there are some simplified versions of these operators in C which are more efficient than the general operators (Table 8.2).

8.3 A Second Look at Data Structures in R

Now that we know how data are coded in C, we can see how R and C can interact and exchange data. Table 8.3 gives the correspondence between the modes in R and the types in C.

Table 8.2. Comparisons of numerical operators in C and in R.

C		R
Efficient	General	
`x++;`	`x = x + 1;`	`x <- x + 1`
`x--;`	`x = x - 1;`	`x <- x - 1`
`x += y;`	`x = x + y;`	`x <- x + y`
`x -= y;`	`x = x - y;`	`x <- x - y`
`x *= y;`	`x = x * y;`	`x <- x * y`
`x /= y;`	`x = x / y;`	`x <- x / y`

First, we notice that all R data correspond to a pointer in C: this makes sense since the basic data structure in R is the vector which is comparable to an array in C.

Second, character vectors are passed from R to C as an array of pointers. Again this makes sense since vectors of mode character are vectors of strings, and a string in C is an array of data type `char`.

Third, R knows only two types of numbers among the many which can be coded in C (see Table 8.1): 32-bit integer and 64-bit floating-point real (Fig. 8.2). However, other data types can be manipulated in a C program called by R: this is just that these data cannot be passed from C to R.[3]

8.4 `.C`

The function `.C` is a simple interface between R and C. It requires a C function which has only pointers as arguments as given in Table 8.3 and returns nothing (`void`). Other variables can be defined and used inside the C code. As a

[3]In fact, any data can be passed from C to R using the raw mode.

Table 8.3. Correspondence between data modes in R and data types in C.

R		C
Mode	Storage mode	Data type
numeric	real	`double *`
	integer	`int *`
character		`char **`
logical		`int *`
complex		`Rcomplex *`
raw		`unsigned char *`
list		`SEXP`

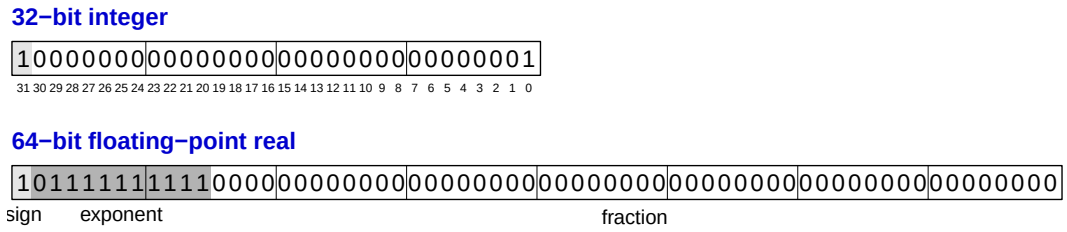


Fig. 8.2. Binary representations of the two types of numbers in R.

simple example, we write a function doing the sum of a vector of numerical (real) values in the file 'sum_C.c':

```
#include <R.h>

int i;

void sum_C(double *x, int *n, double *s)
{
    for (i = 0; i < *n; i++) *s += x[i];
}
```

There are different ways to write this function. For instance, `n[0]` and `s[0]` are similar to `*n` and `*s`.

To be used from R, this function must be compiled but unlike the program compiled above, we will create here a *shared object*, that is an executable code that can be called from another program. R has a specific command to do this operation directly:

```
$ R CMD SHLIB sum_C.c
```

The result of this compilation is the creation of a file with an extension which depends on the OS: 'sum_C.so' on Linux and MacOS or 'sum_C.dll' on Windows. There is one more step before being able to use this C code from R: it must be loaded into R (and we can check that the code is effectively loaded):

```
> dyn.load("sum_C.so")
> is.loaded("sum_C")
[1] TRUE
```

We can now use this C code, for instance with:

```
n <- 100L
x <- rnorm(n)
s <- 0
.C("sum_C", x, n, s)
```

The data types are important here so that we must be sure that `x` and `s` are stored as double, and `n` is stored as an integer. Most users know that `rnorm`

returns decimal values, so we are sure that `x` satisfies the necessary condition. However, the above code will not be robust, for instance if we instead do `x <- 1:100`. Thus, it is common to make sure that the correct data types are used when calling the C code:

```
.C("sum_C", as.double(x), as.integer(n), as.double(s))
```

Let us try this with ten values:

```
> n <- 10
> x <- rnorm(n)
> s <- 0
> x
 [1]  0.80846896  0.79679473 -0.33983748 -0.75843156 -0.85790593
 [6] -0.58028839  0.50513436  0.36157941  0.07925696 -1.28663167
> s <- 0
> .C("sum_C", as.double(x), as.integer(n), as.double(s))
[[1]]
 [1]  0.80846896  0.79679473 -0.33983748 -0.75843156 -0.85790593
 [6] -0.58028839  0.50513436  0.36157941  0.07925696 -1.28663167

[[2]]
[1] 10

[[3]]
[1] -1.271861
```

We see that `.C` returns a list with the values passed as arguments to the C code with the eventual changes performed by the latter.

What happens if there are missing values in `x`?

```
> x[1] <- NA
> .C("sum_C", as.double(x), as.integer(n), as.double(s))
Error: NA/NaN/Inf in foreign function call (arg 1)
```

There is an option, `NAOK`, to control whether missing values are passed or not: the default is `FALSE`, so `NA`'s are not accepted:

```
> args(.C)
function (.NAME, ..., NAOK = FALSE, DUP = TRUE, PACKAGE,
    ENCODING)
NULL
```

This option can be switched to `TRUE` to prevent the error:

```
> .C("sum_C", as.double(x), as.integer(n), as.double(s), NAOK =
    TRUE)
[[1]]
```

```
 [1]           NA  0.79679473 -0.33983748 -0.75843156 -0.85790593
 [6] -0.58028839  0.50513436  0.36157941  0.07925696 -1.28663167

[[2]]
[1] 10

[[3]]
[1] NA
```

But the result is now `NA`: see Section 4.1.3 for an explanation.

Finally, we note the option `DUP = TRUE` which is deprecated (i.e., it cannot be `FALSE`) so that data are always duplicated.

8.5 `.Call`

The `.C` interface has several limitations:

- Only atomic vectors can be passed, thus excluding lists.
- The attributes of these vectors are ignored.
- The number of vectors passed to C is fixed as well as their modes.
- The data are always duplicated.
- No new object is returned, so the results must be returned in a vector created beforehand, so its mode must be known in advance as well as its maximal length.
- Long vectors cannot be passed (see below Sect. 8.5.4).

All these limitations are relaxed with `.Call`. This function shares some common features with `.C` but it is much more sophisticated at the cost of some complications in the C code.

Like `.C`, `.Call` has for first argument a C function which has been previously compiled. The number of objects passed to C must also be defined beforehand, but these can be any kind of R objects. Furthermore, there are several C functions to extract attributes making possible to manipulate R objects efficiently. Table 8.4 compares `.C` and `.Call` by showing how the same operation can be done with both interfaces (and in R as well).

A C file with functions aimed to be called with `.Call` must start with:

```
#include <R.h>
#include <Rinternals.h>
```

We now see each type of R data and their particularities at the C level. Where possible, the C code includes comments with the equivalent command(s) in R.

Table 8.4. Indexing a vector and a matrix in R and in C.

x	R	.C	.Call[a]
vector	`n <- length(x)`	[b]	`n = LENGTH(x);`
	`x[1]`	`x[0]`	`xp[0]`
	`x[n]`	`x[n - 1]`	`xp[n - 1]`
	`x[i] i=1...n`	`x[i] i=0...n - 1`	`xp[i] i=0...n - 1`
matrix	`n <- nrow(x)`	[b]	`n = nrows(x);`
	`p <- ncol(x)`	[b]	`p = ncols(x);`
	`x[1, 1]`[c]	`x[0]`	`xp[0]`
	`x[n, p]`[d]	`x[n * p - 1]`	`xp[n * p - 1]`
	`x[i, j]`	`x[i + n * j]`	`xp[i + n * j]`
	`i=1...n`	`i=0...n - 1`	`i=0...n - 1`
	`j=1...p`	`j=0...p - 1`	`j=0...p - 1`

[a] `xp` is a pointer to `x`.
[b] Must be passed as argument with `.C`.
[c] Identical to `x[1]`.
[d] Identical to `x[n * p]` or `x[length(x)]`.

8.5.1 Vectors

All R objects are of type `SEXP` when handled in a C program called with `.Call`.[4] We must distinguish two situations: either a vector is passed from R to C, or a vector is created within a C code to be eventually returned to R. For instance, if we want to analyse a single vector from R, the first few lines of the C function will look like this:

```
1 SEXP FOO(SEXP x)
2 {
3     PROTECT(x = coerceVector(x, REALSXP));
4     double *xp;
5     xp = REAL(x);
6     ....
```

There are some new information in these few lines of code, so let us look at them in details. First, all data types are R objects, so the declaration in line 1 has only SEXP data types, including the returned value (unlike `.C` which returns `void`). The command on line 3 has two aims: it states explicitly the C data type (with `REALSXP`), and it protects `x` from being deleted by the memory manager of R. We then declare a pointer of type double (line 4). This pointer is used on the next line with the function `REAL` which extracts the address of

[4] SEXP means "S expression" and is a reminder of the ancestry of R as a dialect of the S language.

x: this facilitates the manipulation of the vector at the C level (e.g., xp[0] is the first value in x).

There could be more than one vector passed to C, for instance, if we want to analyse three vectors, the first lines of the C function would be something like:

```
SEXP FOO(SEXP x, SEXP y, SEXP z)
{
    PROTECT(x = coerceVector(x, REALSXP));
    PROTECT(y = coerceVector(y, REALSXP));
    PROTECT(z = coerceVector(z, REALSXP));
    double *xp, *yp, *zp;
    xp = REAL(x);
    yp = REAL(y);
    zp = REAL(z);
    ....
```

In many cases, we want to return a vector so that we need to create one in C. The following is a snippet of code that creates a vector res and a pointer resp to manipulate this vector:[5]

```
SEXP res;
int n;
double *resp;
n = 10;
PROTECT(res = allocVector(REALSXP, n));
resp = REAL(res);
/*  equivalent to:    */
/*  n <- 10           */
/*  res <- numeric(n) */
```

There is an important difference with the numeric(n) command: the values allocated to x are not initialised to zero as R does. What allocVector does is only to allocate the required memory to the object x, but this part of the memory may have been used previously (and freed since then). This may or may not be a problem depending on what we want to do with x: if this vector is destined to accumulate some sums, then it is certainly required to initialise all values in x with zero. A simple way to do this is to loop over the array and assign zero to each element:

```
for (int i = 0; i < n; i++) xp[i] = 0;
```

But a simpler and faster way is to use the C function memset:

```
memset(xp, 0, n * sizeof(double));
```

[5]The lines or blocks of lines delimited between '/*' and '*/' are comments in a C program. Most C compilers also accept '//' at the start of a comment (until the end of the line).

Note that we give the pointer as first argument (not the SEXP object). If the value used for initialisation is not zero, then a `for` loop must be used:

```
/* equivalent to: x[] <- 1 */
for (int i = 0; i < n; i++) xp[i] = 1;
```

Similarly to `.C()`, reals and integers must be treated differently. If the vector passed from R to C with `.Call()` must be treated as integers, then `REALSXP` must be substituted by `INTSXP`, the pointers declared in C must be of the appropriate type, and the function `INTEGER` is used to get the vector address:

```
int *xp;
PROTECT(x = allocVector(INTSXP, n));
xp = INTEGER(x);
memset(xp, 0, n * sizeof(int));
/* equivalent to: integer(n) */
```

For logical vectors, the `SEXP` type is `LGLSXP` but the data type in C is integer:

```
int *xp;
PROTECT(x = allocVector(LGLSXP, n));
xp = INTEGER(x);
memset(xp, 0, n * sizeof(int));
/* equivalent to: logical(n) */
```

For complex numbers, the `SEXP` type is `CPLXSXP`, the C data type is `Rcomplex` which is made of two real numbers, and the address is extracted with `COMPLEX`:

```
Rcomplex *xp;
PROTECT(x = allocVector(CPLXSXP, n));
xp = COMPLEX(x);
/* equivalent to: complex(n) */
xp[0]->r; /* equivalent to: Re(x[1]) */
xp[0]->i; /* equivalent to: Im(x[1]) */
```

For vectors of mode raw, the `SEXP` type is `RAWSXP`, the C data type is `unsigned char`, and the address is extracted with `RAW`:

```
unsigned char *xp;
PROTECT(x = allocVector(RAWSXP, n));
xp = RAW(x);
memset(xp, 0, n * sizeof(unsigned char));
/* equivalent to: raw(n) */
```

The case of character vectors is treated below (Sect. 8.5.3). Finally, the function returns a `SEXP` object and ends with something like:

```
    UNPROTECT(3);
    return x;
}
```

where UNPROTECT removes the protection on the objects created within the function: its argument is the number of protected objects (if this is unbalanced, a warning message is printed during compilation). Note also that return is a statement, not a function: there are no parentheses.

8.5.2 Lists

As we have seen several times in the previous chapters, lists in R are vectors of objects: the SEXP type is VECSXP but there is no matching C basic type so that the elements of the list must be accessed with the special functions VECTOR_ELT and SET_VECTOR_ELT:

```
SEXP x, y, z, L, a;

PROTECT(L = allocVector(VECSXP, 3));
/* equivalent to: L <- vector("list", 3) */
/*            or: L <- list(); length(L) <- 3 */

SET_VECTOR_ELT(L, 0, x); /* equivalent to: L[[1]] <- x */
SET_VECTOR_ELT(L, 1, y); /* equivalent to: L[[2]] <- y */
SET_VECTOR_ELT(L, 2, z); /* equivalent to: L[[3]] <- z */

a = VECTOR_ELT(L, 0); /* equivalent to: a <- L[[1]] */
```

Since a list is a vector of objects, each element it contains is obviously a data of type SEXP. Note that there is no C equivalent of R's $ operator for lists (Table 8.5).

8.5.3 Character Vectors

Vectors of mode character in R are vectors of strings (not of characters). Strings in C are coded with arrays where each element is a character. This difference complicates things a bit, although there are similarities with the other kinds of vectors. The SEXP type is STRSXP:

```
PROTECT(x = allocVector(STRSXP, n));
/* equivalent to: x <- character(n) */
```

Each string is accessed by copying its address to a pointer of type char:

```
const char *xp;
xp = CHAR(STRING_ELT(x, 0));
xp[0] <- 'a';
```

Table 8.5. Indexing a list (L) in R and in C called with .Call (M is a list with three elements; y and z are any R objects).

R	C
`n <- length(L)`	`n = LENGTH(L);`
`L[[1]] <- y`	`SET_VECTOR_ELT(L, 0, y);`
`L[[2]] <- z`	`SET_VECTOR_ELT(L, 1, z);`
`y <- L[[1]]`	`y = VECTOR_ELT(L, 0);`
`z <- L[[2]]`	`z = VECTOR_ELT(L, 1);`

`L[1:3] <- M`	`for (i = 0; i < 3; i++)`
	`  SET_VECTOR_ELT(L, i, VECTOR_ELT(M, i));`
	or
	`for (i = 0; i < 3; i++) {`
	`  y = VECTOR_ELT(M, i);`
	`  SET_VECTOR_ELT(L, i, y);`
	`}`

```
/* equivalent to: substr(x[1], 1, 1) <- "a" */
```

We note two particularities: the pointer is declared with the qualifier `const`, and the first element of the vector is accessed with the function `STRING_ELT` (with a similarity to the way elements of a list are accessed). Thus, the pointer `xp` stores the address of the first string in `x`. So here, the C operator `[` accesses characters within a single string of the vector, whereas it accesses elements within a vector for numerical, logical, complex, and raw vectors.

Another similarity with lists is that a string within a character vector is modified with a special function:

```
SET_STRING_ELT(x, 0, mkChar(xp));
/* equivalent to: xp <- "some string" */
/*                x[1] <- xp          */
```

Another particularity is that the string is checked with the function `mkChar` which insures that it is correctly formatted. If the string is built directly within quotes, it should be terminated with `"\0"`:

```
SET_STRING_ELT(x, 0, mkChar("toto\0"));
/* equivalent to: x[1] <- "toto" */
```

There are several C functions to manipulate strings, for instance to extract its number of characters:

```
int l = strlen(x);
/* equivalent to: l <- nchar(x) */
```

Another useful function (standard in C) is `strcmp` which takes as arguments two `char` pointers and returns 0 if the two strings are identical. This makes

possible to build C code to implement indexing by character. For instance, if x is a vector with names and we want to find the value which under the name "Homo_sapiens":

```
SEXP nmsx;
char *str;
str = "Homo_sapiens\0";
nmsx = getAttrib(x, R_NamesSymbol);
for (int i = 0; i < LENGTH(x); i++) {
    if (! strcmp(CHAR(STRING_ELT(nmsx, i)), str)) break;
}
/* equivalent to: i <- which(names(x) == "Homo_sapiens") */
```

When the loop breaks, i contains the appropriate C index value. Clearly, if some names are duplicated only the first occurrence will be found (see Exercises at the end of the chapter).

8.5.4 Long Vectors

The length of an R object is stored as an integer, and because R handles only 32-bit signed integers and 64-bit floating point reals, if a vector is longer than 2.1 billion[6] its length cannot be stored as an integer. Thus, so-called long vectors have their lengths stored as real numbers. We can see this by creating two vectors with 2.1 billion and 2.2 billion values, and print their respective length:

```
> n1 <- length(1:2.1e9)
> n2 <- length(1:2.2e9)
> n1; n2
[1] 2100000000
[1] 2.2e+09
> storage.mode(n1)
[1] "integer"
> storage.mode(n2)
[1] "double"
```

Long vectors cannot be passed to C with .C but this is possible with .Call. The call from R is similar from what we have seen above. However, on the C side we must pay attention to the data types:

```
double n = XLENGTH(x);
long i;

for (i = 0; i < n; i++) {
 ....
```

[6] See Table A.2 (p. 137) for the exact value.

Table 8.6. Missing values in C.

R	C	
	Name	Data type
NA_real_	NA_REAL	double
NA_integer_	NA_INTEGER	int
NA_logical_	NA_LOGICAL	int
NA_character_	NA_STRING	SEXP
NULL	R_NilValue	SEXP

The length of the vector is extracted with `XLENGTH` (instead of `LENGTH`) which returns a 64-bit real (instead of an integer). Additionally, when accessing the values of a long vector, the index (here `i`) must be able to be greater than 2.1 billion (even greater than 4.2 billion), so that a `long` integer is declared.

8.5.5 Missing and Special Values

There are pre-defined variables for missing values for C code which are given in Table 8.6.

Missing and special values are well defined for real numbers and there are functions to test for them in C, namely: `ISNA`, `ISNAN`, and `R_FINITE`. These functions behave in the same way than their R counterparts (see Table 4.1, p. 45).

For integers this is different and some care must be taken with missing values. We remember from Section 5.4 that there is no coding for infinite values in the case of integers: this is true also in C and R uses the smallest possible integer value for a 32-bit signed integer (see Table A.2 below) for this. Therefore, missing integer values should be tested with something like:

```
x == NA_INTEGER;
```

Without this precaution, unexpected results may happen such as the following C commands:

```
int x, y;
x = NA_INTEGER;
y = NA_INTEGER;
Rprintf("%d\n", x - y);
```

which will print (after appropriate compilation):

```
0
```

(The same observation can be made with logical values.) However, the same operation in R gives the correct answer:

```
> NA_integer_ - NA_integer_
[1] NA
```

On the other hand, real missing values do not have this issue:

```
double x, y;
x = NA_REAL;
y = NA_REAL;
Rprintf("%d\n", x - y);
```

The above code prints (after compilation):

```
nan
```

Missing character values are coded with the string "NA", so that a similar problem occurs: the two following comparisons return 0 (i.e., the strings are identical):

```
strcmp(CHAR(NA_STRING), "NA\0");
strcmp(CHAR(R_NaString), "NA\0");
```

8.6 .External

This interface between R and C handles SEXP objects similarly to `.Call`, the difference is that the number of objects does not need to be defined in advance. The C code may look like this:

```
SEXP foo(SEXP obj)
{
    SEXP x = CAR(obj);
    int n = length(obj);
    for (int i = 0; i < n; i++)
        ....
}
```

which may be called from R with `.External("foo", x, y, z)`. However, we will not detail this interface since an alternative is to use `.Call` with a list passed from R to C, its length can be extracted with `LENGTH`, and the object types can be extracted with `TYPEOF`.

8.7 Profiling C Code

`clock()` is the C equivalent of R's `proc.time()`. This function returns a value of type `clock_t`, different values of this type can be subtracted to calculate time intervals. To illustrate its use, we compare the timings of the two methods described above to initialise an array of numbers (with `memset` or with a `for`

loop). The idea is quite simple: insert lines of code that query the computer clock and print the time intervals at different points of the code (the lines of code added for the present profiling are marked with the comment `//+`):

```
#include <R.h>
#include <time.h> //+

void initialize_C(int *n, double *x)
{
    clock_t t0 = clock(); //+

    Rprintf("0 %d\n", clock() - t0); //+

    memset(x, 0, *n * sizeof(double));

    Rprintf("1 %d\n", clock() - t0); //+

    for (int i = 0; i < *n; i++) x[i] = 0;

    Rprintf("2 %d\n", clock() - t0); //+
}
```

We compile and then try this code with:

```
> n <- as.integer(1e3)
> res <- .C("initialize_C", n, numeric(n))
0 1
1 17
2 26
```

So it took 16 µs to complete the `memset` call, but only 9 µs for the loop. In fact, the order of the operations is important here, but let us not bother about this detail and try to increase the value of `n`:

```
> n <- as.integer(1e6)
> res <- .C("initialize_C", n, numeric(n))
0 6
1 1489
2 3345
```

We have now $\approx$ 1.5 ms and $\approx$ 1.9 ms, respectively, and with an even larger value of `n`:

```
> n <- as.integer(1e8)
> res <- .C("initialize_C", n, numeric(n))
0 4
1 44677
2 146135
```

Now the ratio is more than double at the advantage of `memset`.

This procedure can be used in more complex settings. If there is a `for` loop, this makes possible to see which part(s) of the loop take(s) more computing time. A real-life example is taken from an application on k-means [3]. The outline of the C code is:

```
SEXP foo(SEXP x)
{
    int i;
    clock_t t0 = clock();

    for (i = 0; i < *n; i++) {
        Rprintf("1 %d\n", clock() - t0);
        /* ... some code ... */
        Rprintf("2 %d\n", clock() - t0);
        /* ... some more code ... */

        ....

        Rprintf("9 %d\n", clock() - t0);
    }
}
```

The C code is compiled and called from R with:

```
> sink("prof.out")
> res <- .Call("foo", x)
> sink(NULL)
```

The side-effect of these commands is to print each line into the file 'prof.out' where each timing is preceded by an integer between 1 and 9 indicating the point where the timing was measured. It is then straightforward to analyse the file:

```
> o <- read.table("prof.out")
> nr <- nrow(o)
> str(o)
'data.frame': 432 obs. of  2 variables:
 $ V1: int  1 2 3 4 5 6 7 8 9 1 ...
 $ V2: int  15 75 17806 17837 32468 32497 32504 124807 126463
    126492 ...
```

We now calculate the time intervals between successive points with `diff`, and create text labels (`labs`) with the two points defining these intervals. Both variables can be printed, for instance with boxplots (Fig. 8.3):

```
> timings <- diff(o$V2) / 1000
> labs <- paste(o$V1[-nr], o$V1[-1], sep = "-")
```

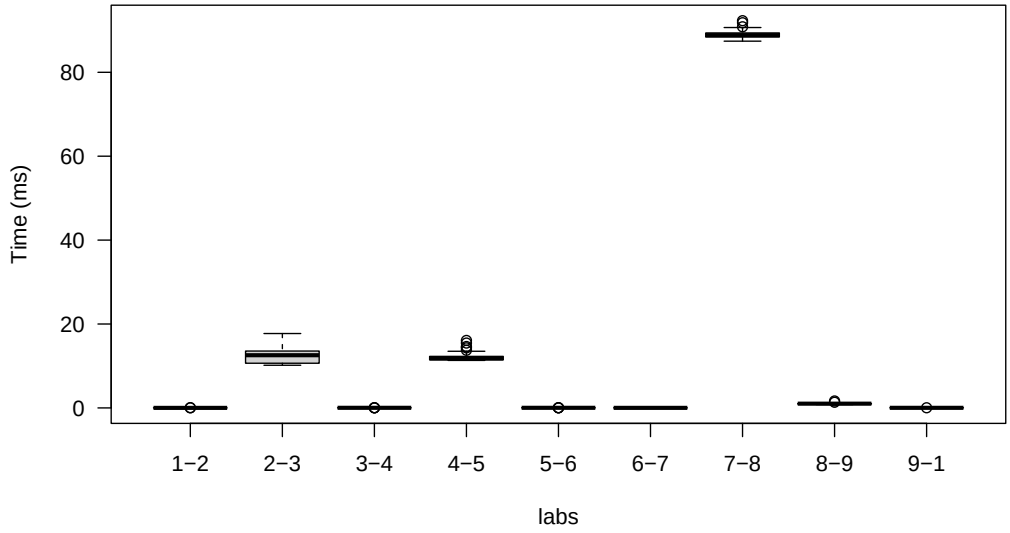


Fig. 8.3. Timing intervals between nine points in a C function (see text for details).

```
> boxplot(timings ~ labs, ylab = "Time (ms)", las = 1)
```

It appears that if the calculations done between points 7 and 8 cannot be improved, then it is very unlikely that the overall performance of the code can better.

8.8 Exercises

1. What is the largest amount of memory usable (i.e., addressable) by a 32-bit CPU?

2. Explain why the size of a pointer is eight bytes on a 64-bit system.

3. How much memory is needed to store one million numerical values in R? What is the gain in terms of memory space if these values are binary?

4. Suppose you need to store numerical values between 0 and 9 in R: what is the most economical way to store them and compare with the standard numeric vectors.

5. Explain why indices in C start at 0 whereas they start at 1 in R.

6. Write the C code which is illustrated in Figure 8.1.

7. Write a C function doing the sum of the values of a vector similar to `sum_C` but using the `.Call` interface. Compare the performance of this version with the one called with `.C` (under different data sizes). Explain the observed differences (if any).

8. Write C code to find the indices of names (or labels) with possibly duplicated names (see p. 116).

9. In the above exercise, what is the value of `i` if the string is not found? (See p. 116.)

10. Write a C program, to be called from R, to do the sum of an indefinite number of vectors.

11. Matrices in C can be coded with an array of pointers, such as `**x`, and manipulated with two sets of indices so that the first element is accessed with `x[0][0]`, and the last one with `x[n - 1][p - 1]`, where `n` and `p` are the numbers of rows and columns, respectively. Explain the difference with the actual system used in R and its C interfaces (see Table 8.4), and why one system is more efficient than the other.

9

Parallel and High Performance Computing

Parallel computing and high performance computing (HPC) have known increased interest with the advent of "BigData" and related issues. However, these are not new topics: parallel computer architectures have existed for decades. These are complicated topics too, so this chapter will look at them in a very pragmatic way with the aim to provide a general understanding, and how some practical solutions can be provided with R.

In order to simplify our view of how computers are built, we consider only three architectures: there could be a single CPU which executes instructions sequentially on data stored on the RAM (Fig. 9.1A). The second architecture has several CPUs, so that instructions can be executed in parallel on data from a single RAM (Fig. 9.1B). So in addition to the transfer of data (which exist also in the simple architecture), there must be some transfers of instructions because the different CPUs must be coordinated in a way or another. The third architecture has several CPUs and several RAM units (and usually several hard disks storing the data permanently; Fig. 9.1C).

We also consider some definitions in order to clarify the followings:

Elementary operation: a basic operation performed by a CPU.

Task: a set of operations performed by a single CPU.

9.1 A Basic Example

Suppose we want to do the sum of four numbers x_1, x_2, x_3, and x_4. The usual (i.e., non-parallel) procedure can be written as (in pseudo-code):

$$s \leftarrow 0;\ s \leftarrow s + x_1;\ s \leftarrow s + x_2;\ s \leftarrow s + x_3;\ s \leftarrow s + x_4;$$

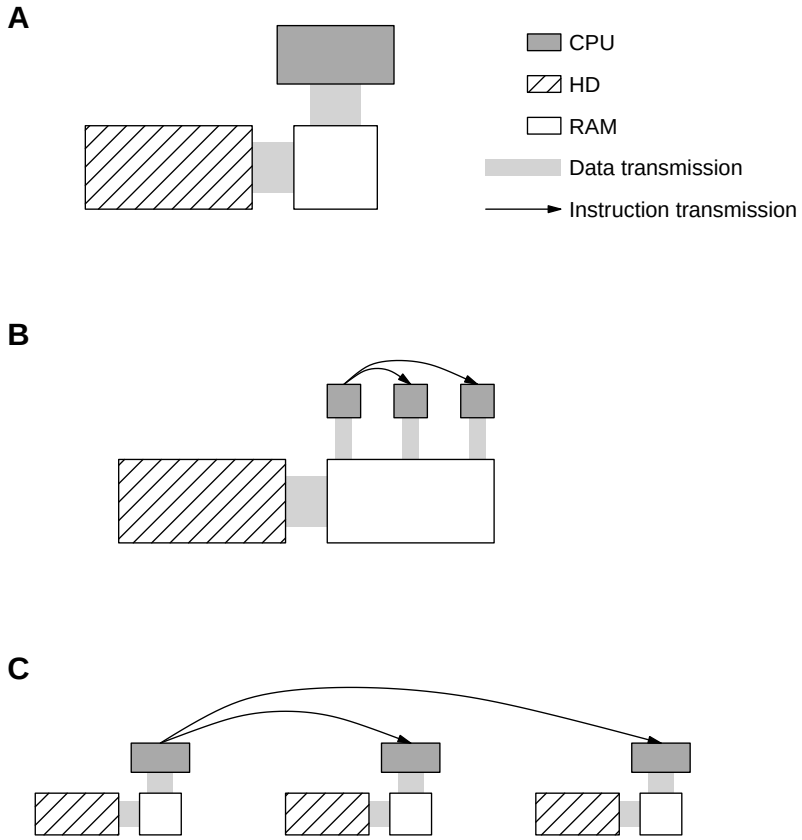


Fig. 9.1. Three basic models of computing architecture: (A) simple architecture; (B) several CPUs sharing the same memory; (C) parallel architecture. CPU: central processing unit; HD: hard disk; RAM: random access (active) memory.

So five elementary operations are required.[1] Let's see now a parallel version of the same procedure. We consider there are two CPUs and the memory is shared. The sum would be done along the following steps:

1. Split the data in two subsets of equal size.
2. Run in parallel:
   (a) $s_1 \leftarrow 0$; $s_1 \leftarrow s_1 + x_1$; $s_1 \leftarrow s_1 + x_2$;
   (b) $s_2 \leftarrow 0$; $s_2 \leftarrow s_2 + x_3$; $s_2 \leftarrow s_2 + x_4$;
3. $s \leftarrow s_1 + s_2$

So three elementary operations have to be done in each parallel task,[2] plus one final elementary operation (the sum of s_1 and s_2). In addition, the first

[1] This could even be reduced to four operations by replacing the first two with $s \leftarrow x_1$.
[2] One elementary operation could be avoided; see previous footnote.

step, splitting the data, is likely to take more time than doing the sum of two numbers because, in the general case, it will require to query the number of values to be added and decide how to split them depending on whether this number is odd or even.

If the same sum is done on a distributed system (Fig. 9.1C), then two additional steps have to be performed:

- Send the data to the CPUs.
- Send back s_1 and s_2.

So doing a sum in parallel does not appear to be a good idea in this simplistic case. The next section explores this question with two specific examples.

9.2 Two Contrasting Examples With `pvec`

To illustrate some properties of parallel computing, we use here the function `pvec` from the package parallel. We want to assess the effect of data size (vector length, n) and the number of cores (mc) on two operations: (1) calculating the square root of numerical values, and (2) converting dates.[3] We choose the values of $n = \{10, 10^2, \ldots, 10^8\}$ and mc $= \{1, 2, 3, 4\}$, create two vectors (`N` and `MC`) to store these values, and a matrix `RES` to store the running times:

```
> N <- 10^(1:8)
> MC <- 1:4
> RES <- matrix(NA, length(N), length(MC))
> dimnames(RES) <- list(N, MC)
```

We now prepare two functions with no argument to perform the parallel (`f()`) and the sequential (`g()`) versions of the square root operation and return only the computing time:

```
library(parallel)
f <- function()
    system.time(x <- pvec(z, sqrt, mc.cores = mc))[3]
g <- function()
    system.time(y <- sqrt(z))[3]
```

Note that this is merely to simplify the writing of the simulation code which is made of two nested `for` loops over the vectors `N` and `MC`:

```
for (i1 in 1:length(N)) {
    z <- 1:N[i1]
    for (i2 in 1:length(MC)) {
        mc <- MC[i2]
        RES[i1, i2] <- if (mc > 1) f() else g()
```

[3] Both applications are inspired from the help page `?pvec`.

```
    }
}
```

The square root is calculated on the values $1, \ldots, n$ (stored in `z`). The results are clear and interpretable by simply printing the matrix:

```
> RES
           1     2     3     4
10     0.000 0.054 0.053 0.061
100    0.000 0.054 0.056 0.054
1000   0.007 0.054 0.054 0.054
10000  0.000 0.063 0.054 0.053
1e+05  0.000 0.085 0.057 0.056
1e+06  0.062 0.128 0.084 0.084
1e+07  0.104 0.671 0.328 0.309
1e+08  1.006 2.920 2.937 2.889
```

So, whatever the size of the data the sequential version is always the fastest one. We note also that using three or four cores are faster than using two only for the largest data sizes ($n \geq 10^4$). The same experiment was done on a computer with a 48-core CPU, this time mc varied between 1 and 36 (and the result matrix is transposed to make it easier to read):

```
> t(RES)
       10   100  1000 10000 1e+05 1e+06 1e+07 1e+08
1  0.000 0.000 0.000 0.001 0.000 0.007 0.105 0.681
2  0.003 0.003 0.004 0.005 0.006 0.027 0.303 2.198
3  0.005 0.004 0.005 0.005 0.006 0.024 0.220 1.951
4  0.006 0.007 0.005 0.006 0.006 0.024 0.225 1.918
5  0.007 0.006 0.008 0.008 0.007 0.023 0.184 1.937
6  0.006 0.009 0.008 0.007 0.008 0.028 0.205 1.955
7  0.007 0.008 0.010 0.007 0.010 0.024 0.183 1.860
8  0.009 0.011 0.009 0.011 0.010 0.027 0.211 1.918
9  0.010 0.009 0.010 0.012 0.011 0.029 0.178 1.860
10 0.012 0.010 0.011 0.012 0.013 0.029 0.199 1.860
11 0.013 0.010 0.015 0.010 0.012 0.029 0.202 1.897
12 0.011 0.012 0.016 0.012 0.013 0.031 0.178 1.956
13 0.011 0.017 0.014 0.013 0.017 0.030 0.212 1.943
14 0.010 0.016 0.016 0.015 0.016 0.033 0.183 2.100
15 0.010 0.015 0.018 0.015 0.021 0.033 0.223 1.985
16 0.012 0.018 0.018 0.017 0.019 0.034 0.195 1.981
17 0.011 0.018 0.019 0.019 0.017 0.035 0.218 2.008
18 0.012 0.017 0.021 0.018 0.022 0.036 0.190 2.009
19 0.011 0.020 0.024 0.019 0.023 0.040 0.224 1.983
20 0.013 0.021 0.025 0.021 0.024 0.048 0.189 2.029
21 0.011 0.022 0.025 0.023 0.027 0.040 0.230 1.998
```

```
22 0.011 0.021 0.023 0.023 0.025 0.039 0.203 2.078
23 0.014 0.021 0.023 0.022 0.027 0.040 0.226 2.112
24 0.012 0.025 0.024 0.027 0.027 0.041 0.207 2.001
25 0.012 0.025 0.025 0.026 0.025 0.043 0.233 1.993
26 0.011 0.027 0.026 0.025 0.029 0.043 0.200 2.037
27 0.011 0.026 0.027 0.028 0.029 0.044 0.233 2.011
28 0.012 0.028 0.027 0.027 0.030 0.046 0.208 2.103
29 0.013 0.028 0.029 0.034 0.032 0.052 0.238 2.079
30 0.012 0.029 0.031 0.030 0.034 0.069 0.211 2.119
31 0.009 0.030 0.032 0.032 0.033 0.059 0.239 2.157
32 0.011 0.031 0.034 0.032 0.037 0.057 0.207 2.119
33 0.012 0.034 0.035 0.034 0.038 0.050 0.239 2.184
34 0.011 0.035 0.035 0.036 0.037 0.052 0.214 2.167
35 0.010 0.036 0.036 0.045 0.038 0.052 0.242 2.268
36 0.010 0.045 0.037 0.037 0.039 0.055 0.219 2.236
```

The results are very similar to the previous one; we note that increasing the value of mc is beneficial only to a limited extent.

We now turn to a more complicated and slower operation: converting random dates (stored in a vector of mode character) into the class "POSIXct" (see Sect. 5.3). The data `z` are generated with:

```
n <- N[i1]
z <- sprintf("%04d-%02d-%02d", as.integer(2000 + rnorm(n)),
       as.integer(runif(n, 1, 12)), as.integer(runif(n, 1, 28)))
```

And the functions `f` and `g` are now:

```
f <- function()
    system.time(b <- pvec(z, as.POSIXct, format = "%Y-%m-%d",
                mc.cores = mc))[3]
g <- function()
    system.time(a <- as.POSIXct(z, format = "%Y-%m-%d"))[3]
```

The simulations are run with the same code than above and the results are:

```
> RES
             1       2       3       4
10       0.000   0.061   0.068   0.065
100      0.001   0.060   0.059   0.061
1000     0.004   0.062   0.062   0.061
10000    0.033   0.077   0.077   0.078
1e+05    0.317   0.248   0.246   0.262
1e+06    3.207   1.834   1.868   1.857
1e+07   32.965  19.889  20.317  20.176
1e+08 327.924 206.488 202.779 199.538
```

Now a gain of the parallel versions is visible but only if $n \geq 10^5$. Furthermore, there is no noticeable difference with respect to the number of cores for mc ≥ 2.

9.3 General Rules

The examples from the previous section make possible to formulate a few general rules about parallel computing.

Rule #1. *Parallel computing is not generic.* Some recent publications in the scientific or technical literature mention parallelisation as a generic (if not "miracle") solution to solve many limitations encountered in computational problems. This is an over-optimistic opinion which is poorly supported by observations.

Rule #2. *Parallel computing depends heavily on the hardware.* This seems obvious, but depending on the architecture, not every parallel code could run on a given machine.

Rule #3. *Parallel computing depends (also) on the software, particularly the OS.* What is possible or not on a computer, especially the access to its hardware, is controlled by the OS. In many machines, the resources (e.g., cores) may be used by other processes (e.g., internet connections), so that not all cores may be available to run tasks in parallel. Besides, the OS may have to run a thread at a given time, and so interrupt temporarily some computations (which have lower priority for the system).

Rule #4. *If the elementary operations are fast, parallelising may not be worthwhile.* This is clearly illustrated with the above examples.

Rule #5. *The more communications among the parallel tasks, the more unlikely parallelisation to be beneficial.* Bootstrap is the typical good candidate for parallel computing because several consecutive operations (resampling, estimation) have to be repeated independently. On the other hand, Markov Chain Monte Carlo (MCMC) have to be updated at each step, so even though the calculations within a given step could be parallelised, the overall procedure must communicate at each iteration. (Of course, independent MCMC chains can be run in parallel.) The same remark can be done for optimisation problems.

Rule #6. *Many computing tasks are parallelisable, but many are not.* Nonlinear models, iterative and recursive functions, etc., can be run in parallel but at a greater cost than the sequential approach.

Rule #7. *Be careful with the vocabulary.* Multithreading is a common term in the literature, but multithreading does not imply necessarily parallel computing. A *thread* is a basic task run by a computer, so that a given application may be made of a single or several threads. A modern computer runs several dozens of threads (e.g., see the commands `ps -aux` and `top` under Linux). R is single-threaded.[4]

[4]For instance, the system command `ps -aux | wc -l` executed on my laptop prints 233, and `top` says there are 231 tasks open. Clearly, not all threads can run in parallel since my

Rule #8. *Beware of code which is already parallel.* Of course, multicore machines are very common, so we can expect many code to be already parallelised. Trying to run such code (see Sect. 9.5 below) with a higher level parallelisation may lead to unexpected problems.

Rule #9. *Avoid using parallel code when running R in a GUI environment.* Many GUIs are multi-threaded so running parallel computing on top of them is likely to result in problems.

9.4 The Package parallel

parallel belongs to the list of R recommended packages, so it is delivered on most installations of R. We see briefly here a few functions from this package, pointing to the options that are common among them.

The function `mcparallel` has the following arguments:

```
> args(mcparallel)
function (expr, name, mc.set.seed = TRUE, silent = FALSE,
    mc.affinity = NULL, mc.interactive = FALSE, detached = FALSE)
```

This function runs the R command specified as an expression (the first argument) in parallel to the current session, so the user can continue to work with R while the computations are done on another CPU. The results are then collected with the function `mccollect`.

We have already used the function `pvec` in the previous section; its arguments are:

```
> args(pvec)
function (v, FUN, ..., mc.set.seed = TRUE, mc.silent = FALSE,
    mc.cores = getOption("mc.cores", 2L), mc.cleanup = TRUE)
```

Compared to `mcparallel`, this one has the option `mc.cores` which specifies how many cores (CPUs) to use.

The function `mclapply` is a parallel version of `lapply`

```
> args(mclapply)
function(X, FUN, ..., mc.preschedule = TRUE, mc.set.seed = TRUE,
    mc.silent = FALSE, mc.cores = getOption("mc.cores", 2L),
    mc.cleanup = TRUE, mc.allow.recursive = TRUE,
    affinity.list = NULL)
```

Both functions work in the same way so their three first arguments are the same. Like `pvec()`, it has the option `mc.cores`.

laptop has four cores.

9.5 C-Level Parallelisations

An alternative to parallelising R code is to perform this at the C level. If this strategy is adopted, parallelisation of the R code must be avoided. We take the same simple example of a sum because we already considered it above—keeping in mind that we are very unlikely to obtain faster code but the simplicity of the problem makes easier to focus on relevant features.

OpenMP is one framework to program C functions to run in parallel. This can be used in C code called from R with a few additional instructions. We apply it below with two functions called with `.C` and `.Call`, respectively. The contents of the file 'sum_openMP.c' is:

```
#include <R.h>
#include <Rinternals.h>
#include <omp.h>

void sum_omp_C(double *x, int *n, double *S)
{
    int K = 4, i, j;
    omp_set_num_threads(K);
    double count[K];

    memset(count, 0, K * sizeof(double));
    /* for (j = 0; j < K; j++) count[j] = 0; */

    #pragma omp parallel for
    for (i = 0; i < *n; i++)
        count[omp_get_thread_num()] += x[i];

    for (j = 0; j < K; j++) S[0] += count[j];
}

SEXP sum_omp_Call(SEXP x)
{
    long i, n;
    int K = 4, j;
    omp_set_num_threads(K);
    double count[K], *p, *s;
    SEXP res;

    PROTECT(x = coerceVector(x, REALSXP));

    memset(count, 0, K * sizeof(double));
    n = (long) XLENGTH(x);
    p = REAL(x);
```

```
34
35      #pragma omp parallel for
36      for (i = 0; i < n; i++)
37          count[omp_get_thread_num()] += p[i];
38
39      PROTECT(res = allocVector(REALSXP, 1));
40      s = REAL(res);
41      s[0] = 0;
42      for (j = 0; j < K; j++) s[0] += count[j];
43      UNPROTECT(2);
44      return res;
45  }
```

First, the header file 'omp.h' is included to permit using the parallel functions. The number of cores used (`K`) is fixed and can be changed which would require to recompile the C code. The number of parallel threads is defined with `omp_set_num_threads(K);`, and an array is declared to store the individual sums computed in each thread (lines 9 and 26). The `for` loop is declared to run in parallel with the statement `#pragma omp parallel for`. Note how the index of `count` is defined (lines 16 and 37). Finally, the individual sums are added together before returning the final result.

By contrast to the procedure explained on page 108, the compilation is done in two steps:

```
1  $ R CMD COMPILE sum_openMP.c CFLAGS=-fopenmp
2  $ R CMD SHLIB sum_openMP.o
```

We can now test this new code with the usual commands:

```
1  > n <- 1e8L
2  > x <- rnorm(n)
3  > s <- 0
4  > system.time(A <- .C("sum_omp_C", x, n, s)[[3]])
5     user  system elapsed
6    1.918   0.170   0.931
7  > system.time(B <- .Call("sum_omp_Call", x))
8     user  system elapsed
9    0.676   0.003   0.214
10 > system.time(C <- sum(x))
11    user  system elapsed
12   0.106   0.000   0.107
13 > A; B; C
14 [1] -6716.085
15 [1] -6716.085
16 [1] -6716.085
17 > A - B
```

```
[1] 0
> A - C
[1] -1.100489e-10
> B - C
[1] -1.100489e-10
```

As expected, the parallel versions are slower than the sequential one.

9.6 Running R on Clusters and Supercomputers

Supercomputers follow the general architecture depicted on Figure 9.1C). Most supercomputers run SLURM (Simple Linux Utility for Resource Management) which is a software to manage the distribution of computing tasks among the resources of the machine. This makes running parallel R sessions relatively easy since SLURM will manage how to distribute the computations among the hardware resources. A simple SLURM script is:[5]

```
#SBATCH --cpus-per-task=1 # Number of cores per MPI rank
#SBATCH --nodes=100 # Number of nodes
#SBATCH --ntasks=100 # Number of MPI ranks
#SBATCH --ntasks-per-node=1 # How many tasks on each node
#SBATCH --ntasks-per-socket=1 # How many tasks on each socket

srun --mpi=pmi2 ~/R/bin/Rscript ~/script.R
```

This will run the code in 'script.R' on 100 nodes.[6] The details will depend on the local configurations. The last line specifies that the files with the executable (here 'Rscript') and the R code are located in the HOME directory of the user (specified with `~/`). It is indeed the practice that users on supercomputers have access to only some directories. Thus, the results of each of the 100 tasks will very likely need to be written in a file in the same directory. The safest way to do this is to name these files randomly. For instance, the file 'script.R' may include commands like:

```
prefix <- sample(c(LETTERS, letters, 0:9), size = 20)
suffix <- "out"
outfile <- paste(prefix, suffix, sep = ".")
```

Different files can be opened with the same prefix, so the user will know that they come from the same node.

This trick, with a few additional steps, can be used to synchronise the different nodes. The difficulty we have here is that the tasks are run fully

[5] The first number sign `#` is part of the command `#SBATCH`, but the second one on the same line starts a comment.

[6] There are other useful commands, such as asking SLURM to send an email to a specified address once the submitted job is completed or stopped.

independently on the different nodes and there is no way for one task to know what the others are doing. Because there is no synchronisation among these tasks, they cannot write data in the same file;[7] however, they can write in different files and each task can query the files which have been written.[8] A solution is that each task writes a file with the name identical to the value given to `prefix`, eventually associated with another set of characters (as prefix and/or suffix) so that these files can be easily identified:

```
Nnodes <- 100
file.create(paste0("THEPREFIX_", prefix))
repeat {
    prefix.files <- dir(pattern = "$THEPREFIX_")
    if (length(prefix.files) == Nnodes) break
    Sys.sleep(1)
}
```

We note the `Sys.sleep(1)` command which leaves one second between two successive `dir()` queries, in case there is a delay in starting some of the tasks.[9] We then include the following commands in the script:

```
prefix.files <- sort(prefix.files)
PREFIX <- gsub("$THEPREFIX_", "", prefix.files)
task.index <- which(PREFIX == prefix)
```

Now each task is identified by an integer `task.index` between 1 and `Nnodes` (remember each task is run on a different node) and it is possible to ask each node to analyse different data. For instance, suppose we have 100 data files, one way to proceed is to list them in a file 'DATAFILES.txt' (of course before launching the script), and the script will include the followings:

```
DATAFILES <- scan("DATAFILES.txt", what = "", sep = "\n")
datafile <- DATAFILES[task.index]
```

The output of the analysis can be written in files individually identified with `prefix`.

9.7 Exercises

1. Write down a list of the data analysis methods that you use commonly (e.g., correlation, ANOVA, PCA, and so on; see Exercises in Chap. 7). Try to write down whether these methods can be parallelised, and if yes sketch the approach which seems appropriate to you.

[7] Actually, they can write in the same file but the results will be unpredictable and very likely useless.

[8] The commands that follow can be adjusted with respect to the directories where the files are written, for instance, `$WORK/` or `$SCRATCH/`.

[9] It is possible to check the current work load of the supercomputer before launching the script.

2. A colleague is performing a data analysis running in parallel on his/her computer while listening to a podcast and checking emails. What advice would you give to them?

3. Try to implement a parallel version of the sum of a vector with `mclapply()`. Compare the performance with `sum()`. Were the results predictable?

4. Do you think a parallel version of the factorial is a good idea?

5. Which model depicted on Fig. 9.1 seems the most appropriate to run a bootstrap in parallel? Same question for Monte Carlo simulations?

6. You need to analyse many data sets with the same method. Do you think this is a good idea to run them in parallel?

7. What is the probability that two files have the same name with the procedure explained in the previous section? What is the required condition to be sure that this does not happen? What if we had added the option `replace = TRUE` in the call to `sample()`?

A

Binary Coding of Numbers

Numbers are coded in computers with sequences of bits. For integers, the logic is simple because there is a one-to-one matching between a sequence of bits and the numbers (Table A.1). The limits of the representation are given by the number of bits and whether one of them is used to code for the sign. Table A.2 gives the smallest and largest values for the main integer types listed in Table 8.1. Figure A.1 gives a graphical representation of their ranges. The largest value is calculated with $\omega = 2^{n-s} - 1$ where n is the number of bits and $s = 1$ if the type is signed, $s = 0$ if unsigned. The smallest value is calculated with $\alpha = -\omega - 1 = -2^{n-s}$ if signed, $\alpha = 0$ if unsigned.

For real numbers, things are more complicated because of the combination of two sets of bits (the fraction and the exponent; Table A.3). Besides, these data types can represent normalized and denormal (or subnormal) numbers. For instance, for 64-bit numbers (i.e., the standard numerical data in R) the smallest resolution is $2^{-52} \approx 2.22 \times 10^{-16}$ (see Fig. B.1); however, smaller numbers can be represented:

```
> 1e-300 > 0 # a denormal number
[1] TRUE
> 1e-400 > 0 # not representable
[1] FALSE
```

But once added to a larger number, such a subnormal number "vanishes":

```
> 1 + 1e-300 > 1
[1] FALSE
```

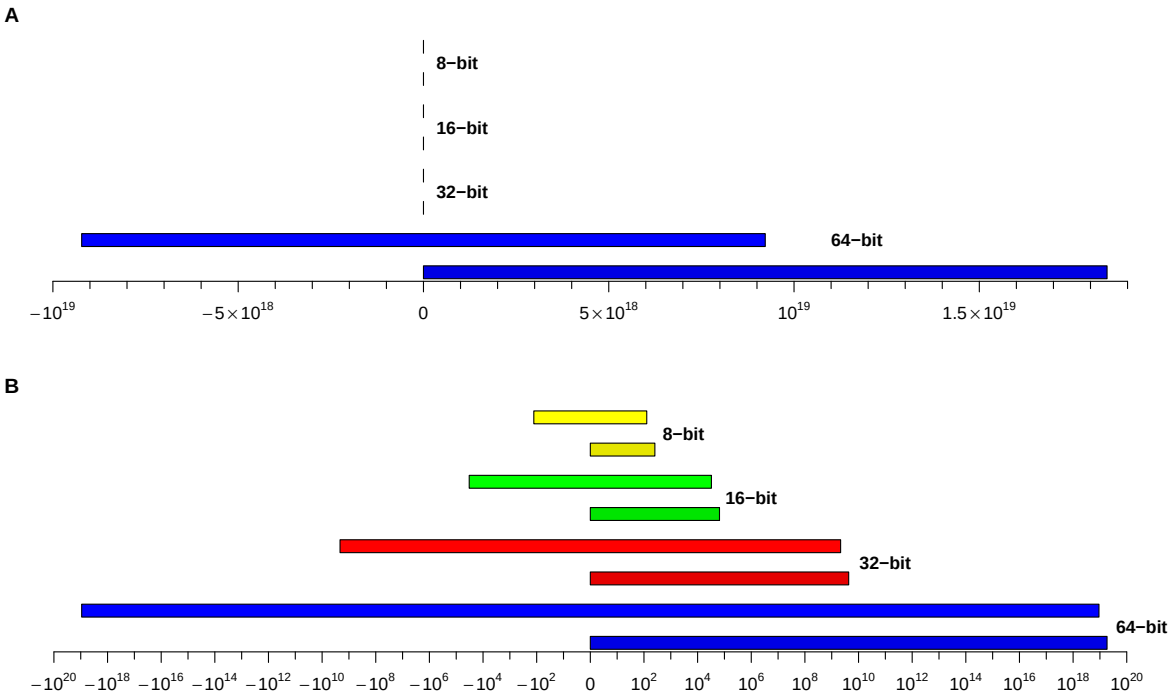


Fig. A.1. Ranges of integer types. (A) linear scale (B) "double logarithmic" scale defined with $\text{sign}(x) \times \log_{10}|x|$ for $x \neq 0$, 0 for $x = 0$. The signed versions are in the darker colours.

Table A.1. A hypothetical 4-bit integer coding scheme.

Binary	Hexadecimal	Signed	Unsigned
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	−1	8
1001	9	−2	9
1010	a	−3	10
1011	b	−4	11
1100	c	−5	12
1101	d	−6	13
1110	e	−7	14
1111	f	−8	15

Table A.2. Main integer data types showing the smallest and largest possible values. n: number of bits

Bytes	n	signed	Smallest	Largest
1	8	yes	-128	127
		no	0	255
2	16	yes	$-32\,768$	$32\,767$
		no	0	$65\,535$
4	32	yes	$-2\,147\,483\,648$	$2\,147\,483\,647$
		no	0	$4\,294\,967\,295$
8	64	yes	-2^{63}	$2^{63}-1$[a]
		no	0	$2^{64}-1$[b]

[a] $2^{63}-1 = 9\,223\,372\,036\,854\,775\,807 \approx 9.2 \times 10^{18}$.
[b] $2^{64}-1 = 18\,446\,744\,073\,709\,551\,615 \approx 1.8 \times 10^{19}$.

Table A.3. Main real data types. n: number of bits (fraction+exponent). All values are approximate.

Bytes	n	Smallest difference from one[a]	Smallest number larger than zero	Largest number
2	16 (11+5)	9.8×10^{-4}	6.1×10^{-5}	6.5×10^{5}
4	32 (24+8)	1.2×10^{-7}	1.2×10^{-38}	10^{38}
8	64 (53+11)	2.2×10^{-16}	2.2×10^{-308}	10^{308}
10	80 (64+15)	1.9×10^{-18}	10^{-4951}	10^{4932}
16	128 (113+15)	1.9×10^{-34}	10^{-4932}	10^{4932}
32	256 (237+19)	9.1×10^{-72}	10^{-78912}	$10^{78\,912}$

[a] This column gives the smallest number i so that: $1 + i > 1$ returns `TRUE`.

B

Computing More Precise Sums

Figure B.1 shows the representable numbers for a 64-bit floating point data type (Fig. B.2 shows all these intervals on linear and logarithmic scales). Because this data type uses 53 bits for the fraction, all numbers between 2^{52} and 2^{53} are exactly represented with a resolution of 1. For the numbers between 2^{51} and 2^{52} the resolution is $\frac{1}{2}$, for those between 2^{50} and 2^{51} the resolution is $\frac{1}{4}$, and so on. Furthermore, for the numbers between 2^{53} and 2^{54} the resolution is 2, for those between 2^{54} and 2^{55} the resolution is 4, and so on. By iteration, we can find that between 2^0 and 2^1 (i.e., 1 and 2) the resolution is $1/2^{52} = 2^{-52} \approx 2.2 \times 10^{-16}$.

Consequently, the representable numbers between 1 and 2 can be written exactly with powers of 2 added to one:

$$1 \qquad 1 + 2^{-52} \qquad 1 + 2(2^{-52}) \qquad 1 + 3(2^{-52}) \qquad \dots \qquad 2$$

This has two important consequences. The first one is that numbers such as $1 + \epsilon$ cannot be represented if $\epsilon < 2^{-52}$: this is quite obvious and follows from the size of the resolution within this interval. The second consequence is that $1 + \epsilon$ cannot be represented if ϵ is not a power of 2: this is precisely the case of $\epsilon = 0.2$ but also many others (0.3, 0.7, etc).[1]

To see more clearly the mechanism behind this, consider the interval from 2^{52} to 2^{53} and let us write $\xi = 2^{52}$. This number is obtained by multiplying 2 by itself fifty-one times, so it is obviously an even integer:

$$\xi = 4\,503\,599\,627\,370\,496.$$

ξ is exactly representable as a 64-bit floating point number. The next representable number is $\xi + 1$ which is also an integer, implying that all numbers

[1] Suppose there exists an integer n so that $1+n(2^{-52}) = 1.2$. Then, we find $n = 0.2\times 2^{52} = \frac{2^{52}}{5}$. However, since the numerator of this fraction is a product of 2's, it can be divided only by powers of the same number (2, 4, 8, 16, ...). Therefore n does not exist as an integer.

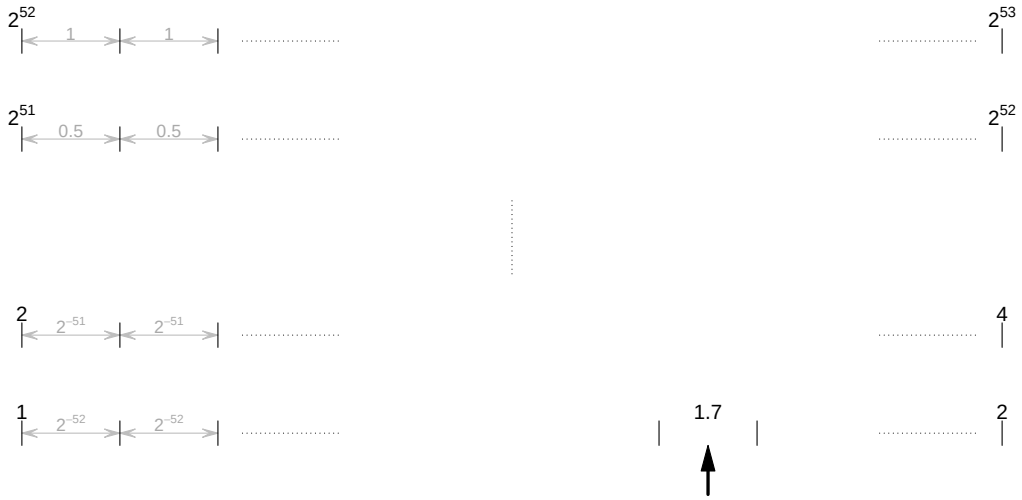


Fig. B.1. Resolution of 64-bit floating-point numbers on different intervals.

between ξ and $\xi + 1$ are not representable. So when one of these are encountered during a calculation (e.g., $\xi + 0.1$), the result will be the closest number between ξ or $\xi + 1$:

```
> 2^52 == 2^52 + 0.1
[1] TRUE
```

And for $\xi + 0.5$, this is done to the lowest number:

```
> 2^52 == 2^52 + 0.5
[1] TRUE
```

The same reasoning can be applied to the interval from 1 to 2 (and of course all the other intervals) except that this time the resolution is 2^{-52}.

This process is likely to affect the computation of sums depending on the order of the data. We can assess this by writing two C functions like these:

```
#include <R.h>

void f(double *x, int *n, double *s)
{
    double S = 0;
    for (int i = 0; i < *n; i++) S += x[i];
    *s = S;
}

void fl(double *x, int *n, double *s)
{
    long double S = 0;
    for (int i = 0; i < *n; i++) S += x[i];
```

```
    *s = (double) S;
}
```

The second version of the summation function (`fl`) has its sum declared as a 128-bit floating point real number (`long double`) which is converted into a 64-bit number (`double`) before returning it. The two R functions are almost identical, only the name of C function being different:

```
f <- function(x)
{
    n <- length(x)
    s <- 0
    .C("f", x, n, s)[[3]]
}

fl <- function(x)
{
    n <- length(x)
    s <- 0
    .C("fl", x, n, s)[[3]]
}
```

We can now compare the precision of both versions and also with R's `sum()`. We generate a vector of random values drawn from the standard normal distribution, reorder them randomly, and compute their sum with these three methods. This is repeated 100 times, all sums are calculated (temporally stored in `S1`, `S2`, and `S3`), and their respective standard-deviation is stored in a matrix (`RES`). To simplify the comparisons, these standard-deviations are log-transformed: if the sums are not affected by the order of the values, then the SD will be zero and $\log(\mathrm{SD}) = -\infty$. The size of the vector varies between 10, 100, ..., 10^7.

```
> RES <- matrix(NA, 7, 3)
> N <- 100
> for (i in 1:7) {
+     n <- 10^i
+     x <- rnorm(n)
+     S1 <- replicate(N, sum(sample(x)))
+     S2 <- replicate(N, f(sample(x)))
+     S3 <- replicate(N, fl(sample(x)))
+     RES[i, 1] <- log10(sd(S1))
+     RES[i, 2] <- log10(sd(S2))
+     RES[i, 3] <- log10(sd(S3))
+ }
> RES
          [,1]       [,2]      [,3]
```

```
[1,]      -Inf -15.611787      -Inf
[2,]      -Inf -14.505427      -Inf
[3,]      -Inf -13.589172      -Inf
[4,]      -Inf -12.598789      -Inf
[5,]      -Inf -11.706848      -Inf
[6,]      -Inf -10.719911 -14.24314
[7,] -12.81935  -9.568902 -12.88850
```

Interestingly, even with only ten values, their order will affect the sum if it is computed with a 64-bit floating point real. R's `sum` actually uses an 80-bit floating point real to store a sum (an "accumulator"), while our version with 128-bit floating point real is as precise.

Using information from Table A.3, we can find that for a 128-bit number the resolution between 10^{112} and 10^{113} is one, between 10^{111} and 10^{112} it is $\frac{1}{2}$, between 10^{110} and 10^{111} it is $\frac{1}{4}$, and so on. So, between 2^{52} and 2^{53} the resolution is $2^{-60} \approx 8.7 \times 10^{-19}$. There are therefore $2^{60} - 1$ ($\approx 1.2 \times 10^{18}$) representable numbers in this interval—whereas there is none for 64-bit numbers.

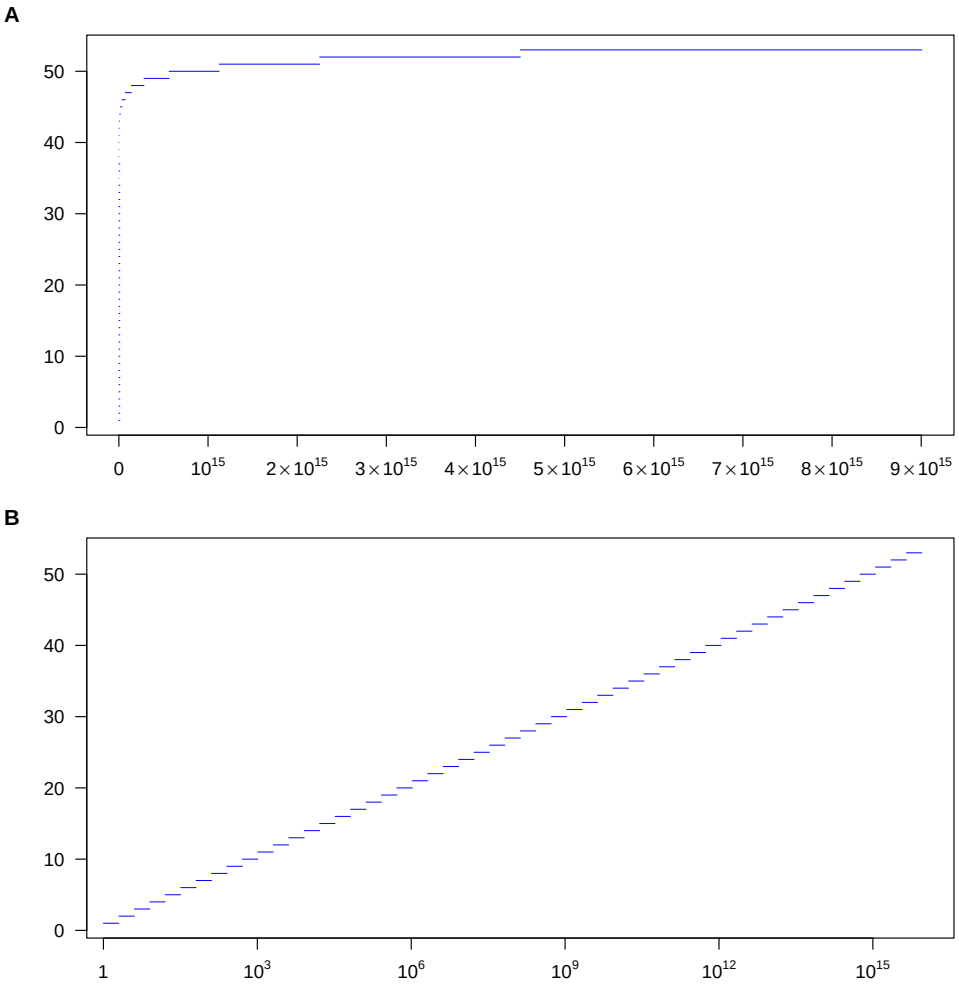


Fig. B.2. The 53 intervals represented on Fig. B.1 on (A) linear scale, and (B) log-scale.

References

[1] Chambers J. M. 2008. *Software for Data Analysis: Programming with R.* Springer, New York.

[2] Ihaka R. & Gentleman R. 1996. R: a language for data analysis and graphics. *Journal of Computational and Graphical Statistics* **5**: 299–314.

[3] Paradis E. 2022. Probabilistic unsupervised classification for large-scale analysis of spectral imaging data. *International Journal of Applied Earth Observations and Geoinformation* **107**: 102675.

Index