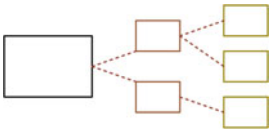
1

User's Manual



The bare essentials, in other words.
—**Ian Rankin**, ***Tooth & Nail***.—

Roadmap

The **Roadmap** is a section that will start each chapter by providing a commented table of contents. It also usually contains indications on the purpose of the chapter.

For instance, in this initial chapter, we explain the typographical notations that we adopted to distinguish between the different semantic levels of the course. We also try to detail how one should work with this book and how one could best benefit from this work. This chapter is to be understood as a user's (or instructor's) manual that details our pedagogical choices. It also seems the right place to introduce the programming language R, which we use to illustrate all the introduced concepts.

In each chapter, both Ian Rankin's quotation and the figure on top of the title page are (at best) vaguely related to the topic of the chapter, and one should not waste too much time pondering their implications and multiple meanings. The similarity with the introductory chapter of *Introducing Monte Carlo Methods with R* is not coincidental, as Robert and Casella (2009) used the same skeleton as in *Bayesian Core* and as we restarted from their version.

J.-M. Marin and C.P. Robert, *Bayesian Essentials with R*, Springer Texts in Statistics, DOI 10.1007/978-1-4614-8687-9_1,

1.1 Expectations

The key word associated with this book is *modeling*, that is, the ability to build up a probabilistic interpretation of an observed phenomenon and the "story" that goes with it. The "grand scheme" is to get anyone involved in analyzing data to process a dataset within this coherent methodology. This means picking a parameterized probability distribution, denoted by f_θ, and extracting information about (shortened in "estimating") the unknown parameter θ of this probability distribution in order to provide a convincing interpretation of the reasons that led to the phenomenon at the basis of the dataset (and/or to be able to draw predictions about upcoming phenomena of the same nature). Before starting the description of the probability distributions, we want to impose on the reader the essential feature that a model is an *interpretation* of a real phenomenon that fits its characteristics up to some degree of approximation rather than an *explanation* that would require the model to be "true". In short, there is no such thing as a "true model", even though some models are more appropriate than others!

In this book, we chose to describe the use of "classical" probability models for several reasons: First, it is often better to start a trip on well-traveled paths because they are less likely to give rise to unexpected surprises and misinterpretations. Second, they can serve as references for more advanced modelings: Quantities that appear in both simple and advanced modelings should get comparable estimators or, if not, the more advanced modeling should account for that difference. At last, the deliberate choice of an artificial model should give a clearer meaning to the motto that *all models are false* in that it illustrates the fact that a model is not necessarily justified by the theory beyond the modeled phenomenon but that its corresponding inference can nonetheless be exploited *as if* it were a true model. By the end of the book, the reader should also be in a position to assess the relevance of a particular model for a given dataset.

Working with this book should not appear as a major endeavor: The datasets are described along with the methods that are relevant for the corresponding model, and the statistical analysis is provided with detailed comments. The R code that backs up this analysis is included and commented throughout the text. If there is a difficulty with this scheme, it actually starts at this point: Once the reader has seen the analysis, it should be possible for her or him to repeat this analysis or a similar analysis with no further assistance. Even better, the reader should try to read as little as possible of the analysis proposed in this book and on the opposite hand should try to conduct the following stage of the analysis *before* reading the proposed (but not unique) solution. The ultimate lesson here is that there are indeed many ways to analyze a dataset and to propose modeling scenarios and inferential schemes. It is beyond the purpose of this book to provide all of those analyses, and the reader (or the instructor) is supposed to look for alternatives on her or his own.

We thus expect readers to place themselves in a realistic situation to conduct this analysis in life-threatening (or job-threatening) situations. As detailed in the preface, the course was originally intended for students in the last year of study toward a professional degree, and it seems quite reasonable to insist that they face similar situations before entering their incoming job!

1.2 Prerequisites and Further Reading

This being a textbook about statistical modeling, the students are supposed to have a background in both probability and statistics, at the level, for instance, of Casella and Berger (2001). In particular, a knowledge of standard sampling distributions and their properties is desirable. Lab work in the spirit of Nolan and Speed (2000) is also a plus. (One should read in particular their Appendix A on "How to write lab reports?") Further knowledge about Bayesian statistics is not a requirement, although using Robert (2007) or Hoff (2009) as further references would bring a better insight into the topics treated here.

Similarly, we expect students to be able to understand the bits of R programs provided in the analysis, mostly because the syntax of R is very simple. We include an introduction to this language in this chapter and we refer to Dalgaard (2002) for a deeper entry and also to Venables and Ripley (2002).

Besides Robert (2007), the philosophy of which is obviously reflected in this book, other reference books pertaining to applied Bayesian statistics include Gelman et al. (2013), Carlin and Louis (1996), and Congdon (2001, 2003). More specific books that cover parts of the topics of a given chapter are mentioned (with moderation) in the corresponding chapter, but we can quote here the relevant books of Holmes et al. (2002), Pole et al. (1994), and Gill (2002). We want to stress that the citations are limited for efficiency purposes: There is no extensive coverage of the literature as in, e.g., Robert (2007) or Gelman et al. (2013), because the prime purpose of the book is to provide a working methodology, for which incremental improvements and historical perspectives are not directly relevant.

While we also cover simulation-based techniques in a self-contained perspective, and thus do not assume prior knowledge of Monte Carlo methods, detailed references are Robert and Casella (2004, 2009) and Chen et al. (2000).

Although we had at some stage intended to write a new chapter about hierarchical Bayes analysis, we ended up not including this chapter in the current edition and this for several reasons. First, we were not completely convinced about the relevance of a specific hierarchical chapter, given that the hierarchical theme is somehow transversal to the book and pops in the mixture (Chap. 6), dynamic (Chap. 7) and image (Chap. 8) chapters. Second, the revision took already too long and creating a brand new chapter did not sound a manageable goal. Third, managing realistic hierarchical models meant relying on codes written in JAGS and BUGS, which clashed with the philosophy of backing the whole book on R codes. This was subsumed by the recent and highly relevant publication of *The BUGS Book* (Lunn et al., 2012) and by the incoming new edition of *Bayesian Data Analysis* (Gelman et al., 2013).

1.3 Styles and Fonts

Presentation often matters almost as much as content towards a better understanding, and this is particularly true for data analyzes, since they aim to reproduce a realistic situation of a consultancy job where the consultant must report to a customer the results of an analysis. An equilibrated use of graphics, tables, itemized comments, and short paragraphs is, for instance, quite important for providing an analysis that stresses the different conclusions of the work, as well as the points that are yet unclear and those that could be expanded.

In particular, because this book is doing several things at once (that is, to introduce theoretical and computational concepts and to implement them in realistic situations), it needs to differentiate between the purposes and the levels of the parts of the text so that it is as obvious as possible to the reader. To this effect, we take advantage of the many possibilities of modern computer editing, and in particular of LaTeX, as follows.

First, a minimal amount of theoretical bases is required for dealing with the model introduced in each chapter, either for Bayesian statistics or for Monte Carlo theory. This aspect of the material is necessarily part of the main text, but it is also kept to a minimum—just enough for the book to be self-contained—and therefore occasional references to more detailed books such as Robert (2007) and Robert and Casella (2004) are necessary. These sections need be well-understood before handling the following applications or realistic cases. This book is primarily intended for those without a strong background in the theory of Bayesian statistics or computational methods, and "theoretical" sections are essential for them, hence the need to keep those sections within the main text.

Statistics is as much about data processing as about mathematical and probabilistic modeling. To enforce this principle, we center each chapter around one or two specific realistic datasets that are described early enough in the chapter to be used extensively throughout the chapter. These datasets are available on the book's Website (`http://www.ceremade.dauphine.fr/~xian/BCS/`) and are part of the corresponding R package `bayess`, as **normaldata**, **capturedata**, and so on, the name being chosen in reference to the case/chapter heading. (Some of these datasets are already available as `datasets` in the R language.) In particular, we explain the "how and why" of the corresponding dataset in a separate paragraph in this shaded format. This style is also used for illustrating theoretical developments for the corresponding dataset and for specific computations related to this dataset. For typographical convenience, large graphs and tables may appear outside these sections, in subsequent pages, but are obviously mentioned and identified within them.

Example 1.1. There may also be a need for detailed examples in addition to the main datasets, although we strived to keep them to a minimum and only for very specific issues where the reference dataset was not appropriate. They follow this numbered style, the sideways triangle indicating the end of the example. ◀

ϟ The last style used in the book is the warning, represented by a lightning ϟ symbol in the margin: This entry is intended to signal major warnings about things that can (and do) go wrong "otherwise"; that is, if the warning is not taken into account. Needless to say, these paragraphs must be given the utmost attention!

A diverse collection of exercises is proposed at the end of each chapter, with solutions to all those exercises freely available on Springer-Verlag webpage.

1.4 An Introduction to R

This section attempts at introducing R to newcomers in a few pages and, as such, it should not be considered as a proper introduction to R. Entire volumes, such as the monumental *R Book* by Crawley (2007), and the introduction by Dalgaard (2002), are dedicated to the practice of this language, and therefore additional efforts (besides reading this chapter) will be required from the reader to sufficiently master the language.[1] However, before discouraging anyone, let us comfort you with the fact that:

(a) The syntax of R is simple and logical enough to quickly allow for a basic understanding of simple R programs, as should become obvious in a few paragraphs.
(b) The best, and in a sense the only, way to learn R is through trial-and-error on simple and then more complex examples. Reading the book with a computer available nearby is therefore the best way of implementing this recommendation.

In particular, the embedded help commands `help()` and `help.search()` are very good starting points to gather information about a specific function or a general issue, even though more detailed manuals are available both locally and on-line. Note that `help.start()` opens a Web browser linked to the local manual pages.

One may first wonder why we support using R as the programming interface for this introduction to Monte Carlo methods, since there exist other

[1] If you decide to skip this chapter, be sure to at least print the handy R Reference Card available at http://cran.r-project.org/doc/contrib/Short-refcard.pdf that summarizes, in four pages, the major commands of R.

languages, most (all?) of them faster than R, like Matlab, and some even free, like C or Python. We obviously have no partisan or commercial involvement in this language.[2] Rather, besides the ease of presentation, our main reason for this choice is that the language combines a sufficiently high power (for an interpreted language) with a very clear syntax both for statistical computation and graphics. R is a flexible language that is *object-oriented* and thus allows the manipulation of complex data structures in a condensed and efficient manner. Its graphical abilities are also remarkable. R provides a powerful *interface* that can integrate programs written in other languages such as C, C++, Fortran, Perl, Python, and Java. At last, it is increasingly common to see people who develop new methodology simultaneously producing an R package in support of their approach and to back up introductory statistics courses with illustrations in R.

One choice we have *not* addressed above is "why R and not BUGS?" BUGS (which stands for Bayesian inference Using Gibbs Sampling) is a Bayesian analysis software developed since the early 1990s, mostly by researchers from the Medical Research Council (MRC) at Cambridge University. The most common version is WinBugs, working under Windows, but there also exists an open-source version called OpenBugs. So, to return to the initial question, we are not addressing the possible links and advantages of BUGS simply because the purpose is different. While access to Monte Carlo specifications is possible in BUGS, most computing operations are handled by the software itself, with the possible outcome that the user does not bother about this side of the problem and instead concentrates on Bayesian modeling. Thus, while R can be easily linked with BUGS and simulation can be done via BUGS, we think that a lower-level language such as R is more effective in bringing you in touch. However, more advanced models like the hierarchical models cannot be easily handled by basic R programming and packages are not necessarily available to handle the variety of those models and call for other programming languages like JAGS. (JAGS standing for *Just Another Gibbs Sampler* and being dedicated to the study of Bayesian hierarchical models. This program is also freely available and distributed under the GNU Licence, the current version being JAGS 3.3.0.)

1.4.1 Getting Started

The R language is straightforward to install: it can be downloaded (obviously free) from one of the numerous CRAN (Comprehensive R Archive Network) mirror Websites around the world.[3]

At this stage, we refrain from covering the installation of the R package and thus assume that (a) R is installed on the machine you want to work with and (b) that you have managed to launch it (in most cases, you simply have

[2] Once again, R is a freely distributed and open-source language.

[3] The main CRAN Website is http://cran.r-project.org/.

to click on the proper icon). In the event you use a friendly (GUI) interface like RKWard, the interface opens several windows whose use should be self-explanatory (along with a proper on-line help). Otherwise, you should then obtain a terminal window whose first lines resemble the following, most likely with a more recent version:

```
R version 2.14.1 (2011-12-22)
Copyright (C) 2011 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
Platform: i686-pc-linux-gnu (32-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

Neither this austere beginning nor the prospect of using a line editor should put you off, though, as there are many other ways of inputting and outputting commands and data, as we shall soon see! The final line above with the symbol `>` means that the R software is waiting for a command from the user. This character `>` at the beginning of each line in the executable window is called the *prompt* and precedes the line command, which is terminated by pressing the `RETURN` key. At this early stage, all commands will be passed as line commands, and you should thus spot commands thanks to this symbol.

Commands and programs that need to be stopped during their execution, for instance because they take too long or too much memory to complete or because they involve a programming mistake such as an infinite loop, can be stopped by the Control-C double-key action without exiting the R session.

For memory and efficiency reasons, R does not install all the available functions and programs when launched but only the basic *packages* that it requires to run properly. Additional packages can be loaded via the `library` command, as in

```
> library(mnormt) # Multivariate Normal and t Distributions
```

and the entire list of available packages is provided by `library()`. (The symbol `#` in the prompt lines above indicates a comment: All characters following `#` until the end of the command line are ignored. Comments are recommended to

improve the readability of your programs.) There exist hundreds of packages available on the Web.[4] Installing a new package such as the package `mnormt` is done by downloading the file from the Web depository and calling

```
> install.package("mnormt")
```

For a given package, the `install.package` command obviously needs to be executed only once, while the `library` call is required each time R is launched (as the corresponding package is not kept as part of the `.RData` file). Thus, it is good practice to include calls to required libraries within your R programs in order to avoid error messages when launching them.

1.4.2 R Objects

As with many advanced programming languages, R distinguishes between several types of *objects*. Those types include scalar, vector, matrix, time series, data frames, functions, or graphics. An R object is mostly characterized by a *mode* that describes its contents and a *class* that describes its structure. The R function `str` applied to any R object, including R functions, will show its structure. For instance,

```
> str(log)
function (x, base = exp(1))
```

The different modes are

- `null` (empty object),
- `logical` (`TRUE` or `FALSE`),
- `numeric` (such as `3`, `0.14159`, or `2+sqrt(3)`),
- `complex`, (such as `3-2i` or `complex(1,4,-2)`), and
- `character` (such as ‘‘`Blue`’’, ‘‘`binomial`’’, ‘‘`male`’’, or ‘‘`y=a+bx`’’),

and the main classes are `vector`, `matrix`, `array`, `factor`, `time-series`, `data.frame`, and `list`. Heterogeneous objects such as those of the `list` class can include elements with various modes. Manual entries about those classes can be obtained via the help commands `help(data.frame)` or `?matrix` for instance.

R can operate on most of those types as a regular function would operate on a scalar, which is a feature that should be exploited as much as possible for compact and efficient programming. The fact that R is interpreted rather than compiled involves many subtle differences, but a major issue is that all variables in the system are evaluated and stored at every step of R programs. This means that loops in R are enormously time-consuming and should be avoided at all costs! Therefore, using the shortcuts offered by R in the manipulation of vectors, matrices, and other structures is a must.

[4] Packages that have been validated and tested by the R core team are listed at http://cran.r-project.org/src/contrib/PACKAGES.html.

The vector *class*

As indicated logically by its name, the `vector` object corresponds to a mathematical vector of elements of the same type, such as `(TRUE,TRUE,FALSE)` or `(1,2,3,5,7,11)`. Creating small vectors can be done using the R command `c()` as in

```
> a=c(2,6,-4,9,18)
```

This fundamental function combines or concatenates terms together. For instance,

```
> d=c(a,b)
```

concatenates the two vectors `a` and `b` into a new vector `d`. Note that decimal numbers should be encoded with a dot, character strings in quotes `" "`, and logical values with the character strings `TRUE` and `FALSE` or with their respective abbreviations `T` and `F`. Missing values are encoded with the character string `NA`.

In Fig. 1.1, we give a few illustrations of the use of vectors in R. The character `+` indicates that the console is waiting for a supplementary instruction, which is useful when typing long expressions. The assignment operator is `=`, not to be confused with `==`, which is the Boolean operator for equality. An older assignment operator is `<-`, as in

```
> x <- c(3,6,9)
```

and, at least for compatibility reasons, it still remains functional in current versions of R, but we prefer using the equality sign. (As pointed out by Spector (2009), an exception is when using `system.time`, briefly described in Fig. 1.8, since `=` is then used to identify keywords, although `=` can preserve its initial purpose if curly brackets `{` and `}` delimit the allocation commands.)

A misleading feature of the assignment operator `<-` is found in Boolean expressions such as

```
> if (x[1]<-2) ...
```

which is supposed to test whether or not `x[1]` is less than -2 but ends up allocating 2 to `x[1]`, erasing its current value! Adding a space in the expression is sufficient to solve the problem: `if (x[1] < -2)`.
Note also that using

```
> if (x[1]=-2) ...
```

mistakenly instead of `(x[1]==-2)` has the same consequence.

New R objects are simply defined by assigning them a value, as in the first line of Fig. 1.1, without a preliminary declaration of type (as in the C language).

`> a=c(5,5.6,1,4,-5)`	build the object `a` containing a numeric vector of dimension 5 with elements 5, 5.6, 1, 4, –5
`> a[1]`	display the first element of `a`
`> b=a[2:4]`	build the numeric vector `b` of dimension 3 with elements 5.6, 1, 4
`> d=a[c(1,3,5)]`	build the numeric vector `d` of dimension 3 with elements 5, 1, –5
`> 2*a`	multiply each element of `a` by 2 and display the result
`> b%%3`	provides each element of `b` modulo 3
`> d%/%2.4`	computes the integer division of each element of `d` by 2.4
`> e=3/d`	build the numeric vector[5] `e` of dimension 3 and elements 3/5, 3, –3/5
`> log(d*e)`	multiply the vectors `d` and `e` term by term and transform each term into its natural logarithm
`> sum(d)`	calculate the sum of `d`
`> length(d)`	display the length of `d`
`> t(d)`	transpose `d`, the result is a row vector
`> t(d)%*%e`	scalar product between the row vector `t(b)` and the column vector `e` with identical length
`> t(d)*e`	element-wise product between two vectors with identical lengths
`> g=c(sqrt(2),log(10))`	build the numeric vector `g` of dimension 2 and elements $\sqrt{2}$, log(10)
`> e[d==5]`	build the subvector of `e` that contains the components `e[i]` such that `d[i]=5`
`> a[-3]`	create the subvector of `a` that contains all components of `a` but the third.
`> is.vector(d)`	display the logical expression `TRUE` if a vector and `FALSE` else

Fig. 1.1. Illustrations of the processing of vectors in R

Note[6] in Table 1.1 the convenient use of Boolean expressions to extract subvectors from a vector without having to resort to a component-by-component test (and hence a loop). The quantity `d==5` is itself a vector of Booleans, while the number of components satisfying the constraint can be computed by `sum(d==5)`. The ability to apply scalar functions to vectors as a whole is also a major advantage of R. In the event the function depends on a parameter or an option, this quantity can be entered as in

[5] The variable `e` is not predefined in R as exp(1).

[6] Positive and negative indices cannot be used simultaneously.

```
> e=lgamma(e^2) #warning: this is not the exponential basis,
  exp(1)
```

which returns the vector with components $\log \Gamma(e_i^2)$. Functions that are specially designed for vectors include, for instance, `sample`, `order`, `sort` and `rank`, which all have to do with manipulating the order in which the components of the vector occur.

Besides their numeric and logical indexes, the components of a vector can also be identified by names. For a given vector `x`, `names(x)` is a vector of characters of the same length as `x`. This additional attribute is most useful when dealing with real data, where the components have a meaning such as `"unemployed"` or `"democrat"`. Those names can also be erased by the command

```
> names(x)=NULL
```

ϟ The `:` operator found in Fig. 1.1 is a very useful device that defines a consecutive sequence, but it is also fragile in that sequences do not always produce what is expected. For instance, `1:2*n` corresponds to `(1:2)*n` rather than `1:(2*n)`.

The matrix, array, *and* factor *classes*

The `matrix` class provides the R representation of matrices. A typical entry is, for instance,

```
> x=matrix(vec,nrow=n,ncol=p)
```

which creates an $n \times p$ matrix whose elements are those of the vector `vec`, assuming this vector is of dimension np. An important feature of this entry is that, in a somewhat unusual way, the components of `vec` are stored by column, which means that `x[1,1]` is equal to `vec[1]`, `x[2,1]` is equal to `vec[2]`, and so on, except if the option `byrow=T` is used in `matrix`. (Because of this choice of storage convention, working on R matrices column-wise is faster then working row-wise.) Note also that, if `vec` is of dimension $n \times p$, it is not necessary to specify both the `nrow=n` and `ncol=p` options in `matrix`. One of those two parameters is sufficient to define the matrix. On the other hand, if `vec` is *not* of dimension $n \times p$, `matrix(vec,nrow=n,ncol=p)` will create an $n \times p$ matrix with the components of `vec` repeated the appropriate number of times. For instance,

```
> matrix(1:4,ncol=3)
     [,1] [,2] [,3]
[1,]    1    3    1
[2,]    2    4    2
Warning message:
data length [4] is not a submultiple or multiple of the
  number
of columns [3] in matrix in: matrix(1:4, ncol = 3)
```

produces a 2×3 matrix along with a warning message that something may be missing in the call to `matrix`. Note again that $1, 2, 3, 4$ are entered consecutively when following the column (or *lexicographic*) order. Names can be given to the rows and columns of a matrix using the `rownames` and `colnames` functions.

Note that, in some situations, it is useful to remember that an R matrix can also be used as a vector. If `x` is an $n \times p$ matrix, `x[i+p*(j-1)]` is equal to `x[i,j]`, i.e., `x` can also be manipulated as a vector made of the columns of `vec` piled on top of one another. For instance, `x[x>5]` is a vector, while `x[x>5]=0` modifies the right entries in the matrix `x`. Conversely, vectors can be turned into $p \times 1$ matrices by the command `as.matrix`. Note that `x[1,]` produces the first row of `x` as a vector rather than as a $p \times 1$ matrix.

R allows for a wide range of manipulations on matrices, both termwise and in the classical matrix algebra perspective. For instance, the standard matrix product is denoted by `%*%`, while `*` represents the term-by-term product. (Note that taking the product `a%*%b` when the number of columns of `a` differs from the number of rows of `b` produces an error message.) Figure 1.2 gives a few examples of matrix-related commands. The `apply` function is particularly easy to use for functions operating on matrices by row or column.

`> x1=matrix(1:20,nrow=5)`	build the numeric matrix `x1` of dimension 5×4 with first row 1, 6, 11, 16
`> x2=matrix(1:20,nrow=5,byrow=T)`	build the numeric matrix `x2` of dimension 5×4 with first row 1, 2, 3, 4
`> a=x3%*%x2`	matrix summation of `x2` and `x3`
`> x3=t(x2)`	transpose the matrix `x2`
`> b=x3%*%x2`	matrix product between `x2` and `x3`, with a check of the dimension compatibility
`> c=x1*x2`	term-by-term product between `x1` and `x2`
`> dim(x1)`	display the dimensions of `x1`
`> b[,2]`	select the second column of `b`
`> b[c(3,4),]`	select the third and fourth rows of `b`
`> b[-2,]`	delete the second row of `b`
`> rbind(x1,x2)`	vertical merging of `x1` and `x2`
`> cbind(x1,x2)`	horizontal merging of `x1` and `x2`
`> apply(x1,1,sum)`	calculate the sum of each row of `x1`
`> as.matrix(1:10)`	turn the vector `1:10` into a 10×1 matrix

Fig. 1.2. Illustrations of the processing of matrices in R

The function `diag` can be used to extract the vector of the diagonal elements of a matrix, as in `diag(a)`, or to create a diagonal matrix with a given diagonal, as in `diag(1:10)`. Since matrix algebra is central to good

programming in R, as matrix programming allows for the elimination of time-consuming loops, it is important to be familiar with matrix manipulation. For instance, the function `crossprod` replaces the product `t(x)%*%y` on either vectors or matrices by `crossprod(x,y)` more efficiently:

```
> system.time(crossprod(1:10^6,1:10^6))
   user  system elapsed
  0.016   0.048   0.066
> system.time(t(1:10^6)%*%(1:10^6))
   user  system elapsed
  0.084   0.036   0.121
```

Eigen-analysis of square matrices is also included in the `base` package. For instance, `chol(m)` returns the upper triangular factor of the Choleski decomposition of `m`; that is, the matrix R such that $R^{\mathrm{T}}R$ is equal to `m`. Similarly, `eigen(m)` returns a list that contains the eigenvalues of `m` (some of which can be complex numbers) as well as the corresponding eigenvectors (some of which are complex if there are complex eigenvalues). Related functions are `svd` and `qr`, which provide the singular values and the QR decomposition of their argument, respectively. Note that the inverse $\mathtt{M}^{-1}$ of a matrix `M` can be found either by `solve(M)` (recommended) or `ginv(M)`, which requires downloading the library MASS and also produces generalized inverses (which may be a mixed blessing since the fact that a matrix is not invertible is not signaled by `ginv`). Special versions of `solve` are `backsolve` and `forwardsolve`, which are restricted to upper and lower diagonal triangular systems, respectively. Note also the alternative of using `chol2inv` which returns the inverse of a matrix `m` when provided by the Choleski decomposition `chol(m)`.

Structures with more than two indices are represented by *arrays* and can also be processed by R commands, for instance `x=array(1:50,c(2,5,5))`, which gives a three-entry table of 50 terms. Once again, they can also be interpreted as vectors.

The `apply` function used in Fig. 1.2 is a very powerful device that operates on arrays and, in particular, matrices. Since it can return arrays, it bypasses calls to multiple loops and makes for (sometimes) quicker and (always) cleaner programs. It should not be considered as a panacea, however, as `apply` hides calls to loops inside a single command. For instance, a comparison of `apply(A, 1,mean)` with `rowMeans(A)` shows the second version is about 200 times faster. Using linear algebra whenever possible is therefore a more efficient solution. Spector (2009, Sect. 8.7) gives a detailed analysis of the limitations of `apply` and the advantages of vectorization in R.

A `factor` is a vector of characters or integers used to specify a discrete classification of the components of other vectors with the same length. Its main difference from a standard vector is that it comes with a `level` attribute used to specify the possible values of the factor. This structure is therefore appropriate to represent qualitative variables. R provides both ordered and

unordered factors, whose major appeal lies within model formulas, as illustrated in Fig. 1.3. Note the subtle difference between `apply` and `tapply`.

`> state=c("tas","tas","sa","sa","wa")`	create a vector with five values
`> statef=factor(state)`	distinguish entries by group
`> levels(statef)`	give the groups
`> incomes=c(60,59,40,42,23)`	create a vector of incomes
`> tapply(incomes,statef,mean)`	average the incomes for each group
`> statef=factor(state,`	define a new level with one more
`+ levels=c("tas","sa","wa","yo"))`	group than observed
`> table(statef)`	return statistics for all levels

Fig. 1.3. Illustrations of the factor class

The list *and* data.frame *classes*

A `list` in R is a rather loose object made of a collection of other arbitrary objects known as its *components*.[7] For instance, a list can be derived from n existing objects using the function `list`:

```
a=list(name_1=object_1,...,name_n=object_n)
```

This command creates a list with n arguments using `object_1,...,object_n` for the components, each being associated with the argument's name, `name_i`. For instance, `a$name_1` will be equal to `object_1`. (It can also be represented as `a[[1]]`, but this is less practical, as it requires some bookkeeping of the order of the objects contained in the list.) Lists are very useful in preserving information about the values of variables used within R functions in the sense that all relevant values can be put within a list that is the output of the corresponding function (see Sect. 1.4.5 for details about the construction of functions in R). Most standard functions in R, for instance `eigen` in Fig. 1.4, return a list as their output. Note the use of the abbreviations `vec` and `val` in the last line of Fig. 1.4. Such abbreviations are acceptable as long as they do not induce confusion. (Using `res$v` would not work!)

The local version of `apply` is `lapply`, which computes a function for each argument of the list

```
> x = list(a = 1:10, beta = exp(-3:3),
+ logic = c(TRUE,FALSE,FALSE,TRUE))
> lapply(x,mean) #compute the empirical means
$a
[1] 5.5
```

[7] Lists can contain lists as elements.

`> li=list(num=1:5,y="color",a=T)`	create a list with three arguments
`> a=matrix(c(6,2,0,2,6,0,0,0,36),nrow=3)`	create a (3,3) matrix
`> res=eigen(a,symmetric=T)`	diagonalize `a` and
`> names(res)`	produce a list with two arguments: `vectors and values`
`> res$vectors`	`vectors` arguments of `res`
`> diag(res$values)`	create the diagonal matrix of eigenvalues
`> res$vec%*%diag(res$val)%*%t(res$vec)`	recover `a`

Fig. 1.4. Chosen features of the list class

```
$beta
[1] 4.535125
$logic
[1] 0.5
```

provided each argument is of a mode that is compatible with the function argument (i.e., is numeric in this case). A "user-friendly" version of `lapply` is `sapply`, as in

```
> sapply(x,mean)
       a     beta    logic
5.500000 4.535125 0.500000
```

The last class we briefly mention here is the `data.frame`. A data frame is a list whose elements are possibly made of differing modes and attributes but have the same length, as in the example provided in Fig. 1.5. A data frame can be displayed in matrix form, and its rows and columns can be extracted using matrix indexing conventions. A list whose components satisfy the restrictions imposed on a data frame can be coerced into a data frame using the function `as.data.frame`. The main purpose of this object is to import data from an external file by using the `read.table` function.

1.4.3 Probability Distributions in R

R is primarily a statistical language. It is therefore well-equipped with probability distributions. As described in Table 1.1, all standard distributions are available, with a clever programming shortcut: A "core" name, such as `norm`, is associated with each distribution and the four basic associated functions, namely the cdf, the pdf, the quantile function, and the simulation procedure, are defined by appending the prefixes `d, p, q, r` to the core name, such as `dnorm()`, `pnorm()`, `qnorm()`, and `rnorm()`. Obviously, each function requires

`> v1=sample(1:12,30,rep=T)`	simulate 30 independent uniform random variables on {1, 2, ..., 12}
`> v2=sample(LETTERS[1:10],30,rep=T)`	simulate 30 independent uniform random variables on {a, b, ..., j}
`> v3=runif(30)`	simulate 30 independent uniform random variables on [0, 1]
`> v4=rnorm(30)`	simulate 30 independent realizations from a standard normal distribution
`> xx=data.frame(v1,v2,v3,v4)`	create a data frame

Fig. 1.5. Definition of a data.frame

additional entries, as in `pnorm(1.96)` or `rnorm(10,mean=3,sd=3)`. Recall that `pnorm()` and `qnorm()` are inverses of one another.

Table 1.1. Standard distributions with R core name

Distribution	Core	Parameters	Default values
Beta	`beta`	`shape1, shape2`	
Binomial	`binom`	`size, prob`	
Cauchy	`cauchy`	`location, scale`	0, 1
Chi-square	`chisq`	`df`	
Exponential	`exp`	`1/mean`	1
Fisher	`f`	`df1, df2`	
Gamma	`gamma`	`shape,1/scale`	`NA`, 1
Geometric	`geom`	`prob`	
Hypergeometric	`hyper`	`m, n, k`	
Log-Normal	`lnorm`	`mean, sd`	0, 1
Logistic	`logis`	`location, scale`	0, 1
Normal	`norm`	`mean, sd`	0, 1
Poisson	`pois`	`lambda`	
Student	`t`	`df`	
Uniform	`unif`	`min, max`	0, 1
Weibull	`weibull`	`shape`	

In addition to these probability functions, R also provides a battery of (classical) statistical tools, ranging from descriptive statistics to nonparametric tests and generalized linear models. A description of these abilities is not possible in this section but we refer the reader to, e.g., Dalgaard (2002) or Venables and Ripley (2002) for a complete entry.

1.4.4 Graphical Facilities

Another clear advantage of using the R language is that it allows a very rich range of graphical possibilities. Functions such as `plot` and `image` can be customized to a large extent, as described in Venables and Ripley (2002) or Murrell (2005) (the latter being entirely dedicated to the R graphic abilities). Even though the default output of `plot` as for instance in

```
> plot(faithful)
```

is not highly most enticing, `plot` is incredibly flexible: To see the number of parameters involved, you can type `par()` that delivers the default values of all those parameters.

> ↯ The wealth of graphical possibilities offered by R should be taken advantage of cautiously! That is, good design avoids clutter, small fonts, unreadable scale, etc. The recommendations found in Tufte (2001) are thus worth following to avoid horrid outputs like those often found in some periodicals! In addition, graphs produced by R usually tend to look nicer on the current device than when printed or included in a slide presentation. Colors may change, font sizes may turn awkward, separate curves may end up overlapping, and so on.

Before covering the most standard graphic commands, we start by describing the notion of device that is at the core of those graphic commands. Each graphical operation sends its outcome to a *device*, which can be a graphical window (like the one that automatically appears when calling a graphical command for the first time as in the example above) or a file where the graphical outcome is stored for printing or other uses. Under Unix, Linux and mac OS, launching a new graphical window can be done via `X11()`, with many possibilities for customization (such as size, positions, color, etc.). Once a graphical window is created, it is given a device number and can be managed by functions that start with `dev.`, such as `dev.list`, `dev.set`, and others. An important command is `dev.off`, which closes the current graphical window. When the device is a file, it is created by a function that is named after its driver. There are therefore a `postscript`, a `pdf`, a `jpeg`, and a `png` function. When printing to a file, as in the following example,

```
> pdf(file="faith.pdf")
> par(mfrow=c(1,2),mar=c(4,2,2,1))
> hist(faithful[,1],nclass=21,col="grey",main="",
+ xlab=names(faithful)[1])
> hist(faithful[,2],nclass=21,col="wheat",main="",
+ xlab=names(faithful)[2])
> dev.off()
```

closing the sequence with `dev.off()` is compulsory since it completes the file, which is then saved. If the command `pdf(file="faith.pdf")` is repeated, the earlier version of the `pdf` file is erased.

Of course, using a line command interface for controlling graphics may seem antiquated, but this is the consequence of the R object-oriented philosophy. In addition, current graphs can be saved to a postscript file using the `dev.copy` and `dev.print` functions. Note that R-produced graphs tend to be large objects, in part because the graphs are not pictures of the current state but instead preserve every action ever taken.

As already stressed above, `plot` is a highly versatile tool that can be used to represent functional curves and two-dimensional datasets. Colors (chosen by `colors()` or `colours()` out of 650 hues), widths, and types can be calibrated at will and LaTeX-like formulas can be included within the graphs using `expression`. Text and legends can be included at a specific point with `locator` (see also `identify`) and `legend`. An example of (relatively simple) output is

```
> plot(as.vector(time(mdeaths)),as.vector(mdeaths),cex=.6,
+ pch=19,xlab="",ylab="Monthly deaths from bronchitis")
> lines(spline(mdeaths),lwd=2,col="red",lty=3)
> ar=arima(mdeaths,order=c(1,0,0))$coef
> lines(as.vector(time(mdeaths))[-1], ar[2]+ar[1]*
+ (mdeaths[-length(mdeaths)]-ar[2]),col="blue",lwd=2,lty=2)
> title("Splines versus AR(1) predictor")
> legend(1974,2800,legend=c("spline","AR(1)"),col=c("red",
+ "blue"),lty=c(3,2),lwd=c(2,2),cex=.5)
```

represented in Fig. 1.6, which compares spline fitting to an AR(1) predictor and to an SAR(1,12) predictor. Note that the seasonal model is doing worse.

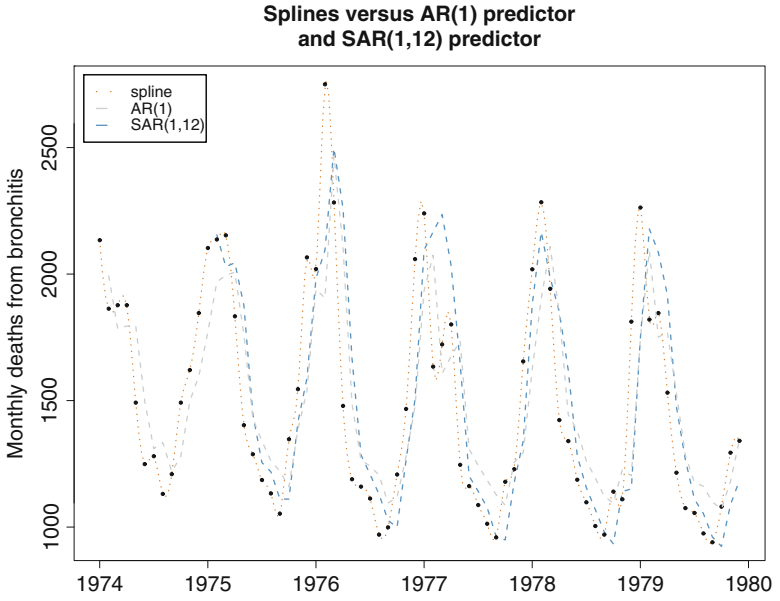


Fig. 1.6. Monthly deaths from bronchitis in the UK over the period 1974–1980 and fits by a spline approximation and an AR predictor

Useful graphical functions include

- `hist` for constructing and optionally plotting histograms of datasets;
- `points` for adding points on an existing graph;
- `lines` for linking points together on an existing graph;
- `polygon` for filling the area between two sets of points;
- `barplot` for creating barplots;
- `boxplot` for creating boxplots.

The two-dimensional representations offered by `image` and `contour` are quite handy when providing likelihood or posterior surfaces. Figure 1.7 gives some of the most usual graphical commands.

`> x=rnorm(100)`	
`> hist(x,nclass=10, prob=T)`	compute and plot an histogram of `x`
`> curve(dnorm(x),add=T)`	draw the normal density on top
`> y=2*x+rnorm(100,0,2)`	
`> plot(x,y,xlim=c(-5,5),ylim=c(-10,10))`	draw a scatterplot of `x` against `y`
`> lines(c(0,0),c(1,2),col="sienna3")`	
`> boxplot(x)`	compute and plot a box-and-whiskers plot of `x`
`> state=c("tas","tas","sa","sa","wa","sa")`	
`> statef=factor(state)`	
`> barplot(table(statef))`	draw a bar diagram of `x`

Fig. 1.7. Some standard plotting commands

1.4.5 Writing New R Functions

One of the strengths of R is that new functions and libraries can be created by anyone and then added to Web depositories to continuously enrich the language. These new functions are not distinguishable from the core functions of R, such as `median()` or `var()`, because those are also written in R. This means their code can be accessed and potentially modified, although it is safer to define new functions. (A few functions are written in C, though, for efficiency.) Learning how to write functions designed for one's own problems is paramount for their resolution, even though the huge collection of available R functions may often contain a function already written for that purpose. A function is defined in R by an assignment of the form

```
name=function(arg1[=expr1],arg2[=expr2],...) {
expression
...
expression
value
}
```

where `expression` denotes an R command that uses some of the arguments `arg1, arg2, ...` to calculate a value, `value`, that is the outcome of the function. The braces indicate the beginning and the end of the function and the brackets some possible default values for the arguments. Note that producing a value at the end of a function is essential because anything done within a function is local and temporary, and therefore lost once the function has been exited, unless saved in `value` (hence, again, the appeal of `list()`). For instance, the following function, named `sqrnt`, implements a version of Newton's method for calculating the square root of `y`:

```
sqrnt=function(y) {
x=y/2
while (abs(x*x-y) > 1e-10) x=(x+y/x)/2
x
}
```

When designing a new R function, it is more convenient to use an external text editor and to store the function under development in an external file, say `myfunction.R`, which can be executed in R as `source("myfunction.R")`. Note also that some external commands can be launched within an R function via the very handy command `system()`. This is, for instance, the easiest way to incorporate programs written in other languages (e.g., Fortran, C, Matlab) within R programs.

Without getting deeply into R programming, let us note a distinction between global and local variables: the former are defined in the core of the R code and are recognized everywhere, while the later are only defined within a specific function. This means in particular that a local variable, `locax` say, initialized within a function, `myfunc` say, will not be recognized outside `myfunc`. (It will not even be recognized in a function defined within `myfunc`.)

The expressions used in a function rely on a syntax that is quite similar to those of other programming languages, with conditional statements such as

```
if (expres1) expres2 else expres3
```

where `expres1` is a logical value, and loops such as

```
for (name in expres1) expres2
```

and

```
while (name in expres1) expres2
```

where `expres1` is a collection of values, as illustrated in Fig. 1.8. In particular, Boolean operators can be used within those expressions, including `==` for testing equality, `!=` for testing inequality, `&` for the logical and, `|` for the logical or, and `!` for the logical contradiction.

Since R is an interpreted language, avoiding loops is generally a good idea, but this may render programs much harder to read. It is therefore extremely useful to include comments within the programs by using the symbol `#`.

```
> bool=T;i=0                                separate commands by semicolons
> while(bool==T) {i=i+1; bool=(i<10)}       stop at i = 11
> s=0;x=rnorm(10000)
> system.time(for (i in 1:length(x)){       output sum(x) and
+ s=s+x[i]})[3]                            provide computing time
> system.time(t(rep(1,10000))%*%x)[3]       compare with vector product
> system.time(sum(x))[3]                    compare with sum() efficiency
```

Fig. 1.8. Some artificial loops in R

1.4.6 Input and Output in R

Large data objects need to be read as values from external files rather than entered during an R session at the keyboard (or by cut-and-paste). Input facilities are simple, but their requirements are fairly strict. In fact, there is a clear presumption that it is possible to modify input files using other tools outside R.

An entire data frame can be read directly with the `read.table()` function. Plain files containing rows of values with a single mode can be downloaded using the `scan()` function, as in

```
> a=matrix(scan("myfile"),nrow=5,byrow=T)
```

When data frames have been produced by another statistical software, the library `foreign` can be used to input those frames in R. For example, the function `read.spss()` allows ones to read SPSS data frames.

Conversely, the generic function `save()` can be used to store all R objects in a given file, either in binary or ASCII format. (The alternative function `dump()` is more rudimentary but also useful.) The function `write.table()` is used to export R data frames as ASCII files.

1.4.7 Administration of R Objects

During an R session, objects are created and stored by name. The command `objects()` (or, alternatively, `ls()`) can be used to display, within a directory called the *workspace*, the names of the objects that are currently stored. Individual objects can be deleted with the function `rm()`. Removing all objects created so far is done by `rm(list=ls())`.

All objects created during an R session (including functions) can be stored permanently in a file in provision of future R sessions. At the end of each R session, obtained by the command `quit()` (which can be abbreviated as `q()`), the user is given the opportunity to save all the currently available objects, as in

```
>q()
Save workspace image? [y/n/c]:
```

If the user answers `y`, the object created during the current session and those saved from earlier sessions are saved in a file called `.RData` and located in the working directory. When R is called again, it reloads the workspace from this file, which means that the user starts the new session exactly where the old one had stopped. In addition, the entire past command history is stored in the file `.Rhistory` and can be used in the current or in later sessions by using the command `history()`.

1.5 The bayess Package

Since this is originally a paper book, copying by hand the R code represented on the following pages to your computer terminal would be both tedious and time-wasting. We have therefore gathered all the programs and codes of this book within an R package called `bayess` that you should download from CRAN before proceeding to the next chapter. Once downloaded on your computer following the instructions provided on the CRAN Webpage, the package `bayess` is loaded into your current R session by `library(bayess)`. All the functions defined inside the package are then available, and so is a step-by-step reproduction of the examples provided in the book, using the `demo` command:

```
> demo(Chapter.1)

        demo(Chapter.1)
        ---- ~~~~~~~~~

Type  <Return>   to start :

> # Chapter 1 R commands
>
> # Section 1.4.2
>
> str(log)
function (x, base = exp(1))

> a=c(2,6,-4,9,18)

> x <- c(3,6,9)

> d=a[c(1,3,5)]

> e=3/d

> e=lgamma(e^2)
```

```
> S=readline(prompt="Type  <Return>   to continue : ")
Type  <Return>   to continue :
```

and similarly for the following chapters. Obviously, all commands contained in the demonstrations and all functions defined in the package can be accessed and modified.

Although most steps of the demonstrations are short, some may require longer execution times. If you need to interrupt the demonstration, recall that `Ctrl-C` is an interruption command.