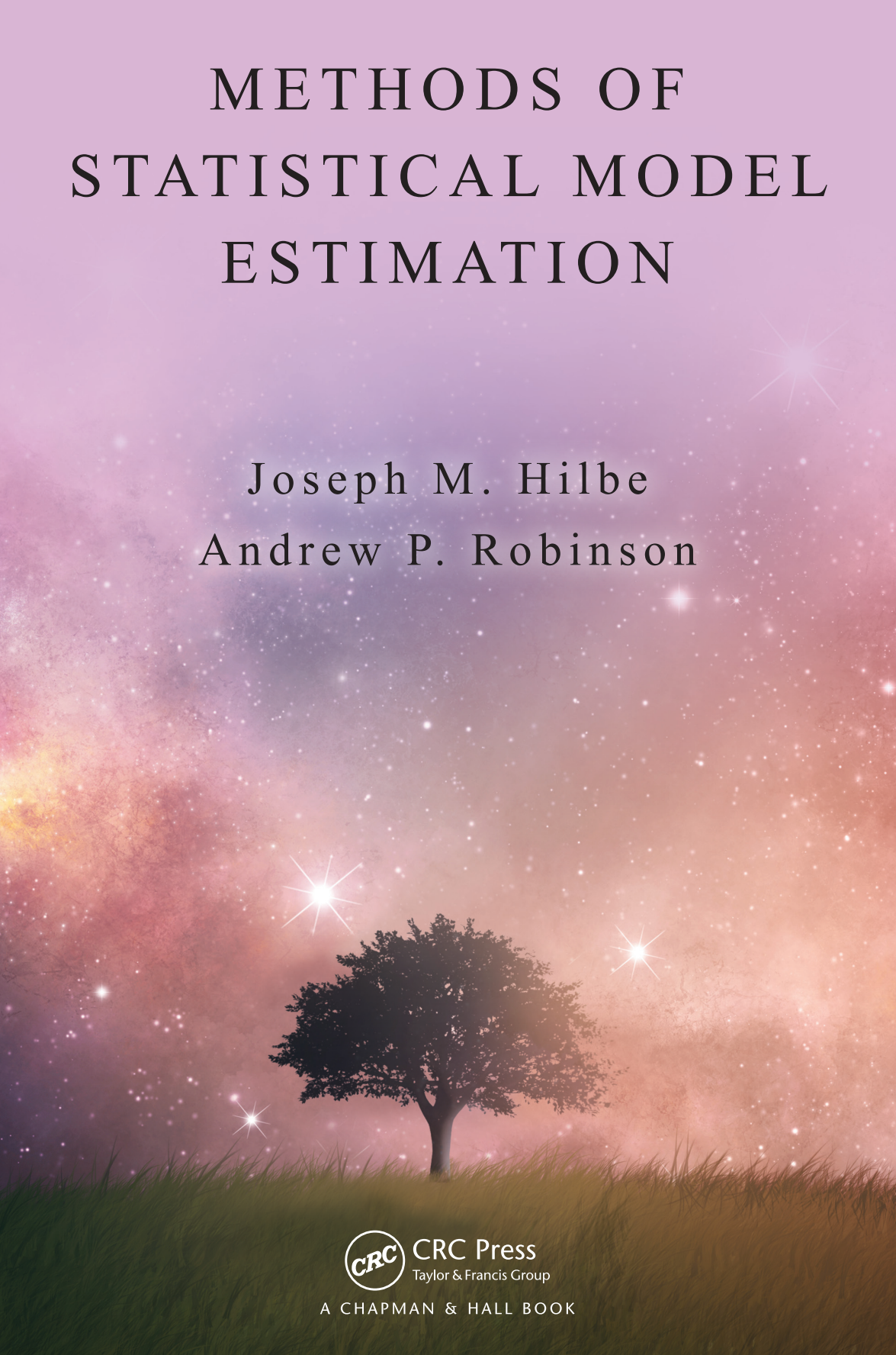
METHODS OF STATISTICAL MODEL ESTIMATION

Joseph M. Hilbe
Andrew P. Robinson



CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

METHODS OF STATISTICAL MODEL ESTIMATION

METHODS OF STATISTICAL MODEL ESTIMATION

Joseph M. Hilbe
Jet Propulsion Laboratory
California Institute of Technology, USA
and
Arizona State Univeristy, USA

Andrew P. Robinson
ACERA & Department of Mathematics and Statistics
The University of Melbourne, Australia

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Version Date: 20130426

International Standard Book Number-13: 978-1-4398-5803-5 (eBook - PDF)

Contents

Preface

Methods of Statistical Model Estimation has been written to develop a particular pragmatic viewpoint of statistical modelling. Our goal has been to try to demonstrate the unity that underpins statistical parameter estimation for a wide range of models. We have sought to represent the techniques and tenets of statistical modelling using executable computer code. Our choice does not preclude the use of explanatory text, equations, or occasional pseudo-code. However, we have written computer code that is motivated by pedagogic considerations first and foremost.

An example is in the development of a single function to compute deviance residuals in Chapter 4. We defer the details to Section 4.7, but mention here that deviance residuals are an important model diagnostic tool for generalized linear models (GLMs). Each distribution in the exponential family has its own deviance residual, defined by the likelihood. Many statistical books will present tables of equations for computing each of these residuals. Rather than develop a unique function for each distribution, we prefer to present a single function that calls the likelihood appropriately itself. This single function replaces five or six, and in so doing, demonstrates the unity that underpins GLM. Of course, the code is less efficient and less stable than a direct representation of the equations would be, but our goal is clarity rather than speed or stability.

This book also provides guidelines to enable statisticians and researchers from across disciplines to more easily program their own statistical models using R. R, more than any other statistical application, is driven by the contributions of researchers who have developed scripts, functions, and complete packages for the use of others in the general research community. At the time of this writing, more than 4,000 packages have been published on the Comprehensive R Archive Network (CRAN) website.

Our approach in this volume is to discuss how to construct several of the foremost types of estimation methods, which can then enable readers to more easily apply such methods to specific models. After first discussing issues related to programming in R, developing random number generators, numerical optimization, and briefly developing packages for publication on CRAN, we discuss in considerable detail the logic of major estimation methods, including ordinary least squares regression, iteratively re-weighted least squares, maximum likelihood estimation, the EM algorithm, and the estimation of model parameters using simulation. In the process we provide a number of guidelines that can be used by programmers, as well as by statisticians and researchers in general regarding statistical modelling.

Datasets and code related to this volume may be found in the *msme* package on CRAN. We also will have R functions and scripts, as well as data, available for download on Prof. Hilbe's BePress Selected Works website, `http://works.bepress.com/joseph_hilbe/`. The code and data, together with errata and a PDF document named `MSME_Extensions.pdf`, will be in the `msme` folder on the site. The extensions document will have additional code or guidelines that we develop after the book's publication that are relevant to the book. These resources will also be available at the publisher's website, `http://www.crcpress.com/product/ISBN/9781439858028`.

Readers will find that some of the functions in the package have not been exported, that is, made explicitly available when the package is loaded in memory. They can still be called, using the protocol shown in the following example for `Sjll`:

```
msme:::Sjll( ... insert arguments here as usual! ...)
```

That is, prepend the library name and three colons to the function call.

We very much encourage feedback from readers, and would like to have your comments regarding added functions, operations and discussions that you would like to see us write about in a future edition. Our goal is to have this book be of use to you for writing your own code for the estimation of various types of models. Moreover, if readers have written code that they wish to share with us and with other readers, we welcome it and can put it in the Extensions ebook for download. We will always acknowledge the author of any code we use or publish. Annotated, self-contained code is always preferred!

Readers are assumed to have a background in R programming, although the level of programming experience necessary to understand the book is rather minimal. We attempt to explain every R construct and operation; therefore, the text should be successfully used by anyone with an interest in programming. Of course the more background one has in using R, and in programming in general, the easier it will be to implement the code we developed for this volume.

Overview

Chapter 1 is the introductory chapter providing readers with the basics of R programming, including the specifics of R objects, functions, matrices, object-oriented programming, and creating R packages.

Chapter 2 deals with the nature of statistical models and of maximum likelihood estimation. We then introduce random number generators and provide code for constructing them, as well as for writing code for simple simulation activities. We also outline the rationale for using profile likelihood-based standard errors in place of traditional model-based standard errors.

Chapter 3 addresses basic ordinary least squares (OLS) regression. Code structures are developed that will be used throughout the book. Least-squares regression is compared to full maximum likelihood methodology. Also discussed are problems related to missing data, reporting standard errors and confidence intervals, and to understanding S3 class modelling.

Chapter 4 relates to the theory and logic of programming generalized linear models (GLM). We spend considerable time analyzing the iteratively reweighted least squares (IRLS) algorithm, which has traditionally been used for the estimation of GLMs. We first demonstrate how to code specific GLM models as stand-alone functions, then show how all of them can be incorporated within a GLM covering algorithm. The object is to demonstrate the development of modular programming. A near complete GLM function, called `irls`, is coded and explained, with code for the three major Bernoulli and binomial models, Poisson, negative binomial, gamma, and inverse Gaussian models included. We also provide the function with a wide variety of post estimation statistics and a separate summary function for displaying regression results. Topics such as over-dispersion, offsets, goodness-of-fit, and residual analysis are also examined.

Chapter 5 develops traditional maximum likelihood methodology showing how to write modular code for one- and two-parameter GLMs as full maximum likelihood models. One parameter GLMs (function `ml_glm`) include binomial, Poisson, gamma, and inverse Gaussian families. Our `ml_glm2` function allows modelling of two-parameter Gaussian, gamma, and negative binomial regression models. We also develop a model that was not previously available in R, the heterogeneous negative binomial model, or NB-H. The NB-H model allows parameterization of the scale parameter as well as for standard predictor parameters.

Chapter 6 provides the logic and code for using maximum likelihood for the estimation of basic fixed effects and random effects models. We provide code for a conditional fixed effects negative binomial as well as a Gaussian random intercept model. We also provide an examination of the logic and annotated code for a working EM algorithm.

In the final chapter, Chapter 7, we address simulation as a method for estimating the parameters of regression procedures. We demonstrate how to construct synthetic models, then Monte Carlo simulation and finally how to employ Markov Chain Monte Carlo simulation for the estimation of Poisson regression coefficients, standard errors and associated statistics. In doing so we provide the basis of Bayesian modelling. We do not, however, wish to discuss Bayesian methodology in detail, but only insofar as the basic method can be used in estimating model parameters. Fully working annotated code is provided for the estimation of a Bayesian model with non-informative priors. The code can easily be adapted for the use of other data and models, as well as for the incorporation of priors.

Exercise questions are provided at the end of each chapter. We encourage

the readers to try answering them. We have designed them so that they are answerable given the information provided in the chapter.

Our goal throughout has been to produce a clear and fully understandable volume on writing code for the estimation of statistical models using the foremost estimation techniques that have been developed by statisticians over the last half century. We attempt to use examples that will be of interest to researchers across disciplines.

As a general rule, we will include R code that the reader should be able to run, conditional on the successful execution of earlier code, and signal that code with the usual R prompts '`>`' and '`+`'. We will also include pseudo-code that provides some greater generality but should not be run as is. We omit the prompts for the pseudo-code to distinguish it from the executable code.

Acknowledgments

We wish to thank Rob Calver, statistics editor at Chapman & Hall/CRC (Taylor and Francis), for believing in this project, and for allowing us to write the book as we saw fit. Others whom we wish to thank include, [Hilbe] Alain Zuur, Highland Statistics, James Hardin (University of South Carolina), Robert Muenchen (University of Tennessee), and [Robinson] Mark Burgman (University of Melbourne), Jeff Gove (USDA FS), John Maindonald (ANU), Gordon Smyth (WEHI), and Murray Aitkin (University of Melbourne). We thank the R and LaTeX communities, and the authors and maintainers of Sweave, for these phenomenal resources.

Authoring books such as this one takes a great deal of writing and research time. However, most of our time was taken up in coding, testing, error checking, running models, re-coding, and so forth. This effort takes considerable time and patience, time that would otherwise be spent with our families. We therefore thank our families for not complaining about the times we were physically, as well as mentally, absent while working on this volume. Specifically, JMH wishes to acknowledge the support of his wife Cheryl, daughter Heather, sons Michael and Mitchell, grandsons Austin and Shawn, and Sirr, a white Maltese who keeps him company throughout the day when working on the computer. APR is grateful for the support of Grace, his son Felix, and Henry, a black-and-tan mutt who got walked to the beach far less often than he would like.

Joseph M. Hilbe
Florence, AZ, USA (`hilbe@asu.edu`)

Andrew P. Robinson
Melbourne, Australia (`apro@unimelb.edu.au`)

1

Programming and R

1.1 Introduction

The goal of this chapter is to introduce the reader to the programming tools that will be used in subsequent chapters. It therefore provides a highly selective review of R programming.

Users who have some exposure to data analysis and statistical packages that provide graphical user interfaces may be wary about such a seemingly bare-bones introduction to R. Many other data analysis products provide apparently straightforward importation of data, accompanied by attractive graphics and automated model fitting. Why is it useful to dig about in the tissue of the language? The reason is that, in our experience as statisticians, the provision to the analyst of data that are clean and ready to analyze is the exception rather than the rule. Invariably some pre-analysis processing is required. R provides a very flexible and powerful set of tools for the manipulation of data. Careful use of these tools will both ease the process and improve the transparency of preparing the data for suitable analysis. Therefore, close examination of the data manipulation facilities of R will benefit the analyst.

The definitive reference to R is the R Language Definition, which is freely available in PDF and HTML format on the R website, as well as being provided by default with each R installation. This work is continually updated by the volunteers that support R. It can be accessed via the `The R Language Definition` link on the front page of the html help file that is opened by the `help.start` function.

1.2 R Specifics

Making a definitive description of R is a tricky proposal, because R is multifaceted and evolving. Therefore, we will tackle a simpler problem and describe R just as we will be treating it in this book. R, for the purposes of this book, is an interpreted, impure object-oriented programming language that provides many structures and functions that ease the importation, handling, and analysis of data, as well as reporting the outcome. Also for the purposes

of this book, R is software that the user interacts with via a command-line interface. The label R is commonly used to describe both the language and the software application that interprets it. R is more fully documented in readily available resources (e.g., R Development Core Team, 2012).

According to the R FAQ, the design of R has been heavily influenced by two other languages: it is very similar in appearance to Becker, Chambers & Wilks' S, and its underlying implementation and semantics are derived from Sussman's Scheme (Hornik, 2010). R is not uncommonly described as being "not unalike S."

R is an interpreted, as opposed to a compiled, language. An interpreted language is one for which the most common implementation involves translating the code to machine-executable commands and executing those commands one at a time. Re-running the program requires re-translation. A compiled implementation is one in which programs are written as collections of instructions, then converted to binary objects. Such binary objects then can be run many times without re-translation. We note in passing that R can run programs that have been compiled from other languages, such as C and FORTRAN. Furthermore, as of R version 2.13.0, a byte compiler is available, which provides useful although occasionally modest decreases in execution time.

As far as the user is concerned, the disadvantage of R being an interpreted language rather than a compiled language is that it is slower in execution than it would be if it were compiled. However, in the experience of the authors, the execution time of R is very rarely a bottleneck in analytical exercises.

The R software provides an interpreter, or listener. The listener accepts user input in the form of R code, then the software interprets the input, executes the instructions, and returns the output. In practice, the user types commands at the prompt, and R executes those commands, like this:

```
> 1 + 2

[1] 3
```

R, being somewhat like S, is somewhat object oriented (OO). Describing R's OO nature is complicated because of several factors: first, by the fact that there are different kinds of object orientation, and second, that R itself provides several implementations of OO programming. Indeed it is possible, although may be inefficient, for the user to ignore R's OO nature entirely. Hence R is an impure object-oriented language (according to the definition supplied by Craig, 2007, for example).

In this book we constrain ourselves to describing the implementation of S3 classes, which were introduced to S in version 3 (Chambers, 1992a). At the time of writing, R also provides S4 classes (Chambers, 1998), and the user-contributed packages *R.oo* (Bengtsson, 2003) and *proto* (Kates and Petzoldt, 2007). S3's object orientation is class-based, as opposed to prototype based. We will cover object-oriented programming (OOP) using R in more detail in Section 1.3.3. In the meantime, we need to understand that R allows the user

to create and manipulate objects, and that this creation and manipulation is central to interacting efficiently with R.

1.2.1 Objects

Everything in R is an *object*. Objects are created by the evaluation of R *statements*. If we want to save an object for later manipulation, which we most commonly do, then we choose an appropriate name for the object and assign the name using the left arrow symbol `<-`. It is also possible to use the equals sign `=`; however, in this book we prefer `<-`. So object creation is, broadly, as follows.

```
name <- R statements
```

Valid object names may contain letters, digits and the two characters `.` and `_`, and must start with a letter or `.` (Chambers, 2008).

1.2.1.1 Vectors

We start with a vector object, of which there are six types: real, string, logical, integer, complex, and raw. Here we will focus on the first three types. We create a vector of three real numeric objects, which we shall call `wavelengths`, as follows:

```
> wavelengths <- c(325.3, 375.6, 411.1)
```

This code used the `c` function to concatenate the three numbers into a vector object, and then assigned the vector object a name: `wavelengths`. This vector object is a container for the three real numbers. We can print the object by just entering its name at the prompt.

```
> wavelengths
```

```
[1] 325.3 375.6 411.1
```

Every object in R has a *class*, which controls how the object can be manipulated. The class of the object can be determined (and set) using the `class` function.

```
> class(wavelengths)
```

```
[1] "numeric"
```

Classes are baked into base R, so knowing what they do and what they are for helps the user understand what R is doing. We will cover classes in greater detail later in this chapter. For the moment, we comment that knowing the class is very helpful.

We can create a vector of character strings in the same way:

```
> sentence <- c("This", "is", "a", "character", "vector")
> class(sentence)

[1] "character"
```

We remark that R has some wonderful character-handling functions such as `paste`, `nchar`, `substr`, `grep`, and `gsub`, but their coverage is beyond the scope of this book.

Many operations are programmed so that if the operation is called on the vector, then it is efficiently carried out on each of its elements. For example,

```
> wavelengths / 1000

[1] 0.3253 0.3756 0.4111
```

This is not true for all operations; some necessarily operate on the entire vector, for example, `mean`,

```
> mean(wavelengths)

[1] 370.6667
```

and `length`.

```
> length(wavelengths)

[1] 3
```

R also provides a special class of integer-like objects called a *factor*. A factor is used to represent a categorical (actually nominal, more precisely) variable, and that is the reason that it is important for this book. The object is displayed using a set of character strings, but is stored as an integer, with a set of character strings that are attached to the integers. Factors differ from character strings in that they are constrained in terms of the values that they can take.

```
> a_factor <- factor(c("A","A","B","B","B","C"))
> a_factor

[1] A A B B B C
Levels: A B C
```

Note that when we printed the object, we were told what the *levels* of the factor were. These are the only values that the object can take. Here, we try to turn the 6th element into a Z.

```
> a_factor[6] <- "Z"
> a_factor
```

```
[1] A    A    B    B    B    <NA>
Levels: A B C
```

We failed: R has made the 6th element *missing* (`NA`) instead. Note that we accessed the individual element using square brackets. We will expand on this topic in Section 1.2.1.2.

The levels, that is the permissible values, of a factor object can be displayed and manipulated using the `levels` function. For example,

```
> levels(a_factor)
```

```
[1] "A" "B" "C"
```

Now we will turn the second element of the levels into `Bee`.

```
> levels(a_factor)[2] <- "Bee"
```

The consequence of this operation is that when we now print the factor, the levels have been changed.

```
> a_factor
```

```
[1] A    A    Bee  Bee  Bee  <NA>
Levels: A Bee C
```

One challenge that new R users often have with factors is that the functions that are used to read data into R will make assumptions about whether the intended class of input is factor, character string, or integer. These assumptions are documented in the appropriate help files, but are not necessarily obvious otherwise. This is especially tricky for some data where numbers are mixed with text, sometimes accidentally. Manipulating factors as though they really were numeric is often perilous. For example, they cannot be added.

```
> factor(1) + factor(2)
```

```
[1] NA
```

The final object class that we will describe is the logical class, which can be thought of as a special kind of factor with only two levels: `TRUE` and `FALSE`. Logical objects differ from factors in that mathematical operators can be used, e.g.,

```
> TRUE + TRUE
```

```
[1] 2
```

It is standard that `TRUE` evaluates to 1 when numerical operations are applied, and `FALSE` evaluates to 0. Logical objects are created, among other ways, by evaluating logical statements, for example

```
> 1:4 < 3
```

```
[1]  TRUE  TRUE FALSE FALSE
```

Logical objects can be manipulated by the usual logical operators, *and* `&`, *or* `|`, and *not* `!`. Here we evaluate `TRUE` (or) `TRUE`.

```
> TRUE | TRUE
```

```
[1] TRUE
```

In evaluating this statement, R will evaluate each logical object and then the logical operator. An alternative is to use `&&` or `||`, for which R will evaluate the first expression and then only evaluate the second if it is needed. This approach can be faster and more stable under some circumstances. However, the latter versions are not vectorized.

The last kind, but not class, of object that we want to touch upon is the missing value, `NA`. This is a placeholder that R uses to represent known unknown data. Such data cannot be lightly ignored, and in fact R will often retain missing values throughout operations to emphasize that the outcome of the operation depends on the missing value(s). For example,

```
> missing.bits <- c(1, NA, 2)
> mean(missing.bits)
```

```
[1] NA
```

If we wish to compute the mean of just the non-missing elements, then we need to provide an argument to that effect, as follows (see Section 1.2.3.1, below).

```
> mean(missing.bits, na.rm = TRUE)
```

```
[1] 1.5
```

Note that the missing element is counted in the length, even though it is missing.

```
> length(missing.bits)
```

```
[1] 3
```

We can assess and set the missing status using the `is.NA` function.

1.2.1.2 Subsetting

In the previous section we extracted elements from vectors. Subsets can easily be extracted from many types of objects, using the square brackets operator or the `subset` function. Here we demonstrate only the former. The square brackets take, as an argument, an expression that can be evaluated to either an integer object or a logical object. For example, using an integer, the second item in our sentence is

```
> sentence[2]

[1] "is"
```

and the first three are

```
> sentence[1:3]

[1] "This" "is"   "a"
```

Note that R has interpreted `1:3` as the sequence of integers starting at 1 and concluding at 3. We can also exclude elements using the negative sign:

```
> sentence[-4]

[1] "This"   "is"     "a"      "vector"

> sentence[-(2:4)]

[1] "This"   "vector"
```

An example of the use of a logical expression for subsetting is

```
> wavelengths > 400

[1] FALSE FALSE  TRUE

> wavelengths[wavelengths > 400]

[1] 411.1
```

These subsetting operations can be nested. Alternatively, the intermediate results can be stored as named objects for subsequent manipulation. The choice between the two approaches comes down to readability against efficiency; creating interim variables slows the execution of the code but allows commands to be split into more easily readable code chunks.

1.2.2 Container Objects

Other object classes are containers for objects. We cover two particularly useful container classes in the section: lists and dataframes.

1.2.2.1 Lists

A list is an object that can contain other objects of arbitrary and varying classes. A list is created as follows.

```
> my.list <- list(number = 1, text = "alphanumeric")
```

Elements can be manipulated or extracted from the list using the double square bracket symbol, or the `$` symbol, as follows.

```
> my.list[[1]]

[1] 1

> my.list$text

[1] "alphanumeric"

> my.list[[1]] <- 2
> my.list

$number
[1] 2

$text
[1] "alphanumeric"
```

Empty lists can be conveniently created using the `vector` function, as follows.

```
> capacious.empty.list <- vector(1000, mode = "list")
```

This list can then be used as a container for the outcome of a loop. Functions like `lapply` and `sapply` allow for elegant, element-wise evaluation of functions upon lists. For example, to determine the class of each element in a list, we can use the following code:

```
> sapply(my.list, class)

     number        text
  "numeric" "character"
```

Lists are very useful devices when programming functions, because functions in R are only allowed to return one object. Hence, if we want a function to return more than one object, then we have it return a list that contains all the needed objects.

1.2.2.2 Dataframes

Dataframes are special kinds of lists that are designed for storing and manipulating datasets as they are commonly found in statistical analysis. Dataframes typically comprise a number of vector objects that have the same length; these objects correspond conceptually to columns in a spreadsheet. Importantly, the objects need not be of the same class. Under this setup, the i-th unit in each column can be thought of as belonging to the i-th observation in the dataset.

There are numerous ways to construct dataframes within an R session. We find the most convenient way to be the `data.frame` function, which takes objects of equal length as arguments and constructs a dataframe from them. If the arguments are named, then the names are used for the corresponding variables. For example,

```
> example <- data.frame(var.a = 1:3,
+                       var.b = c("a","b","c"))
> str(example)

'data.frame':        3 obs. of  2 variables:
 $ var.a: int  1 2 3
 $ var.b: Factor w/ 3 levels "a","b","c": 1 2 3
```

If the arguments are of different lengths, then R will repeat the shorter ones to match the dimension of the longest ones, and report a warning if the shorter are not a factor (in the mathematical sense) of the longest.

```
> data.frame(var.a = 1,
+            var.b = c("a","b","c"))

  var.a var.b
1     1     a
2     1     b
3     1     c
```

Furthermore, as is more commonly used in our experience, when data are read into the R session using one of the `read.xxx` family of functions, the created object is a dataframe.

There are also numerous ways to extract information from a dataframe, of which we will present only two: the square bracket `[` operator and `subset`.

The square bracket operator works similarly as presented in Section 1.2.1.2, except instead of one index it now requires two: the first for the rows, and the second for the columns.

```
> example[2, 1:2]

  var.a var.b
2     2     b
```

A blank is taken to mean that all should be included.

```
> example[2,]

  var.a var.b
2     2     b
```

Note that this operation is calling a function that takes an object and indices as its arguments. As before, the arguments can be positive or negative integers, or a logical object that identifies the rows to be included by `TRUE`.

Extracting data using the `subset` function proceeds similarly, with one exception: the index must be logical; it cannot be integer.

```
> subset(example, subset = var.a > 1, select = "var.b")

  var.b
2     b
3     c
```

It is worth noting that storing data in a dataframe is less efficient than using a matrix, and manipulating dataframes is, in general, slower than manipulating matrices. However, matrices may only contain data that are all the same class. In cases where data requirements are extreme, it may be worth trying to use matrices instead of dataframes.

1.2.3 Functions

We write functions to enable the convenient evaluation of sets of expressions. Functions serve many purposes, including, but not limited to, improving the readability of code, permitted the convenient re-use of code, and simplifying the process of handling intermediate objects.

In R, functions are objects, and can be manipulated as objects. A function comprises three elements, namely: a list of arguments, a body, and a reference to an environment. We now briefly describe each of these using an example. This trivial function sums its arguments, and if the second argument is omitted, it is set to 1.

```
> example.ok <- function (a, b = 1) {
+   return(a + b)
+ }
> example.ok(2,2)

[1] 4

> example.ok(2)

[1] 3
```

We can now examine the pieces of the function by calling the following functions. The formals are the arguments,

```
> formals(example.ok)

$a


$b
[1] 1
```

the body is the R code to be executed,

```
> body(example.ok)

{
    return(a + b)
}
```

and the environment is the parent environment.

```
> environment(example.ok)

<environment: R_GlobalEnv>
```

We describe each of these elements in greater detail below.

1.2.3.1 Arguments

The arguments of a function are a list (actually a special kind of list, called a *pairlist*, which we do not describe further) of names and, optionally, expressions that can be used as default values. So, the arguments might be for example `x`, or `x = 1`, providing the default value 1, or indeed `x =` *some expression*, which will be evaluated if needed as the default expression.

The function's arguments must be valid object names, that is, they may contain letters, digits and the two characters `.` and `_`, and must start with either a letter or the period `.` (Chambers, 2008). If an expression is provided as part of the function definition, then that expression is evaluated and used as the default value, which is used if the argument is not named in the function call.

```
> example <- function(a = 1) a
> example()

[1] 1
```

Note that the argument expressions are not evaluated until they are needed — this is lazy evaluation. We can demonstrate this by passing an expression that will result in a warning when evaluated.

```
> example <- function(a, b) a
> example(a = 1)

[1] 1

> example(a = 1, b = log(-1))

[1] 1
```

The absence of warning shows that the expression has not been evaluated.

It is important to differentiate between writing and calling the function when thinking about arguments. When we write a function, any arguments that we need to use in that function must be named in the argument list. If we omit them, then R will look outside the function to find them. More details are provided in Section 1.2.3.3.

```
> example <- function(a) a + b

> example(a = 1)

Error in example(a = 1) : object 'b' not found
```

Now if we define `b` in the environment in which the function was created, the parent environment, then the code runs.

```
> b <- 1
> example(a = 1)

[1] 2
```

Note that R searched the parent environment for `b`.

An exception is that if we want to pass optional arbitrary arguments to a function that is called within our function, then we use the `...` argument. Below, note how `example` requires arguments `a` and `b`, but when we call it within our `contains` function we need to provide only `a` and the dots.

```
> example <- function(a, b) a + b
> contains <- function(a, ...) example(a, ...)
> contains(a = 1, b = 2)

[1] 3
```

Our `contains` function was able to handle the argument by passing it to `example`, even without advance warning.

In calling a function, R will match the arguments by name, position, or both name and position. For example,

```
> example(1, 2)
```