1 Introduction and preliminaries

1.1 The R environment

R is an integrated suite of software facilities for data manipulation, calculation and graphical display. Among other things it has

- an effective data handling and storage facility,
- a suite of operators for calculations on arrays, in particular matrices,
- a large, coherent, integrated collection of intermediate tools for data analysis,
- graphical facilities for data analysis and display either directly at the computer or on hard-copy, and
- a well developed, simple and effective programming language (called 'S') which includes conditionals, loops, user defined recursive functions and input and output facilities. (Indeed most of the system supplied functions are themselves written in the S language.)

The term "environment" is intended to characterize it as a fully planned and coherent system, rather than an incremental accretion of very specific and inflexible tools, as is frequently the case with other data analysis software.

R is very much a vehicle for newly developing methods of interactive data analysis. It has developed rapidly, and has been extended by a large collection of *packages*. However, most programs written in R are essentially ephemeral, written for a single piece of data analysis.

1.2 Related software and documentation

R can be regarded as an implementation of the S language which was developed at Bell Laboratories by Rick Becker, John Chambers and Allan Wilks, and also forms the basis of the S-PLUS systems.

The evolution of the S language is characterized by four books by John Chambers and coauthors. For R, the basic reference is *The New S Language: A Programming Environment for Data Analysis and Graphics* by Richard A. Becker, John M. Chambers and Allan R. Wilks. The new features of the 1991 release of S are covered in *Statistical Models in S* edited by John M. Chambers and Trevor J. Hastie. The formal methods and classes of the **methods** package are based on those described in *Programming with Data* by John M. Chambers. See Appendix F [References], page 97, for precise references.

There are now a number of books which describe how to use R for data analysis and statistics, and documentation for S/S-PLUS can typically be used with R, keeping the differences between the S implementations in mind. See Section "What documentation exists for R?" in *R FAQ*.

1.3 R and statistics

Our introduction to the R environment did not mention *statistics*, yet many people use R as a statistics system. We prefer to think of it of an environment within which many classical and modern statistical techniques have been implemented. A few of these are built into the base R environment, but many are supplied as *packages*. There are about 25 packages supplied with R (called "standard" and "recommended" packages) and many more are available through the CRAN family of Internet sites (via `https://CRAN.R-project.org`) and elsewhere. More details on packages are given later (see Chapter 13 [Packages], page 75).

Most classical statistics and much of the latest methodology is available for use with R, but users may need to be prepared to do a little work to find it.

There is an important difference in philosophy between S (and hence R) and the other main statistical systems. In S a statistical analysis is normally done as a series of steps, with

intermediate results being stored in objects. Thus whereas SAS and SPSS will give copious output from a regression or discriminant analysis, R will give minimal output and store the results in a fit object for subsequent interrogation by further R functions.

1.4 R and the window system

The most convenient way to use R is at a graphics workstation running a windowing system. This guide is aimed at users who have this facility. In particular we will occasionally refer to the use of R on an X window system although the vast bulk of what is said applies generally to any implementation of the R environment.

Most users will find it necessary to interact directly with the operating system on their computer from time to time. In this guide, we mainly discuss interaction with the operating system on UNIX machines. If you are running R under Windows or macOS you will need to make some small adjustments.

Setting up a workstation to take full advantage of the customizable features of R is a straightforward if somewhat tedious procedure, and will not be considered further here. Users in difficulty should seek local expert help.

1.5 Using R interactively

When you use the R program it issues a prompt when it expects input commands. The default prompt is '`>`', which on UNIX might be the same as the shell prompt, and so it may appear that nothing is happening. However, as we shall see, it is easy to change to a different R prompt if you wish. We will assume that the UNIX shell prompt is '`$`'.

In using R under UNIX the suggested procedure for the first occasion is as follows:

1. Create a separate sub-directory, say `work`, to hold data files on which you will use R for this problem. This will be the working directory whenever you use R for this particular problem.

   ```
   $ mkdir work
   $ cd work
   ```

2. Start the R program with the command

   ```
   $ R
   ```

3. At this point R commands may be issued (see later).
4. To quit the R program the command is

   ```
   > q()
   ```

   At this point you will be asked whether you want to save the data from your R session. On some systems this will bring up a dialog box, and on others you will receive a text prompt to which you can respond *`yes`*, *`no`* or *`cancel`* (a single letter abbreviation will do) to save the data before quitting, quit without saving, or return to the R session. Data which is saved will be available in future R sessions.

Further R sessions are simple.

1. Make `work` the working directory and start the program as before:

   ```
   $ cd work
   $ R
   ```

2. Use the R program, terminating with the `q()` command at the end of the session.

To use R under Windows the procedure to follow is basically the same. Create a folder as the working directory, and set that in the `Start In` field in your R shortcut. Then launch R by double clicking on the icon.

1.6 An introductory session

Readers wishing to get a feel for R at a computer before proceeding are strongly advised to work through the introductory session given in Appendix A [A sample session], page 80.

1.7 Getting help with functions and features

R has an inbuilt help facility similar to the `man` facility of UNIX. To get more information on any specific named function, for example `solve`, the command is

```
> help(solve)
```

An alternative is

```
> ?solve
```

For a feature specified by special characters, the argument must be enclosed in double or single quotes, making it a "character string": This is also necessary for a few words with syntactic meaning including `if`, `for` and `function`.

```
> help("[[")
```

Either form of quote mark may be used to escape the other, as in the string `"It's important"`. Our convention is to use double quote marks for preference.

On most R installations help is available in HTML format by running

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

which will launch a Web browser that allows the help pages to be browsed with hyperlinks. On UNIX, subsequent help requests are sent to the HTML-based help system. The 'Search Engine and Keywords' link in the page loaded by `help.start()` is particularly useful as it is contains a high-level concept list which searches though available functions. It can be a great way to get your bearings quickly and to understand the breadth of what R has to offer.

The `help.search` command (alternatively `??`) allows searching for help in various ways. For example,

```
> ??solve
```

Try `?help.search` for details and more examples.

The examples on a help topic can normally be run by

```
> example(topic)
```

Windows versions of R have other optional help systems: use

```
> ?help
```

for further details.

1.8 R commands, case sensitivity, etc.

Technically R is an *expression language* with a very simple syntax. It is *case sensitive* as are most UNIX based packages, so `A` and `a` are different symbols and would refer to different variables. The set of symbols which can be used in R names depends on the operating system and country within which R is being run (technically on the *locale* in use). Normally all alphanumeric symbols are allowed[1] (and in some countries this includes accented letters) plus '`.`' and '`_`', with the restriction that a name must start with '`.`' or a letter, and if it starts with '`.`' the second character must not be a digit. Names are effectively unlimited in length.

Elementary commands consist of either *expressions* or *assignments*. If an expression is given as a command, it is evaluated, printed (unless specifically made invisible), and the value is lost. An assignment also evaluates an expression and passes the value to a variable but the result is not automatically printed.

[1] For portable R code (including that to be used in R packages) only A–Z, a–z, and 0–9 should be used.

Commands are separated either by a semi-colon (‘`;`’), or by a newline. Elementary commands can be grouped together into one compound expression by braces (‘`{`’ and ‘`}`’). *Comments* can be put almost[2] anywhere, starting with a hash mark (‘`#`’), everything to the end of the line is a comment.

If a command is not complete at the end of a line, R will give a different prompt, by default

```
+
```

on second and subsequent lines and continue to read input until the command is syntactically complete. This prompt may be changed by the user. We will generally omit the continuation prompt and indicate continuation by simple indenting.

Command lines entered at the console are limited[3] to about 4095 bytes (not characters).

1.9 Recall and correction of previous commands

Under many versions of UNIX and on Windows, R provides a mechanism for recalling and re-executing previous commands. The vertical arrow keys on the keyboard can be used to scroll forward and backward through a *command history*. Once a command is located in this way, the cursor can be moved within the command using the horizontal arrow keys, and characters can be removed with the `DEL` key or added with the other keys. More details are provided later: see Appendix C [The command-line editor], page 90.

The recall and editing capabilities under UNIX are highly customizable. You can find out how to do this by reading the manual entry for the **readline** library.

Alternatively, the Emacs text editor provides more general support mechanisms (via ESS, *Emacs Speaks Statistics*) for working interactively with R. See Section “R and Emacs” in *R FAQ*.

1.10 Executing commands from or diverting output to a file

If commands[4] are stored in an external file, say `commands.R` in the working directory `work`, they may be executed at any time in an R session with the command

```
> source("commands.R")
```

For Windows **Source** is also available on the **File** menu. The function `sink`,

```
> sink("record.lis")
```

will divert all subsequent output from the console to an external file, `record.lis`. The command

```
> sink()
```

restores it to the console once again.

1.11 Data permanency and removing objects

The entities that R creates and manipulates are known as *objects*. These may be variables, arrays of numbers, character strings, functions, or more general structures built from such components.

During an R session, objects are created and stored by name (we discuss this process in the next section). The R command

```
> objects()
```

(alternatively, `ls()`) can be used to display the names of (most of) the objects which are currently stored within R. The collection of objects currently stored is called the *workspace*.

---

[2] **not** inside strings, nor within the argument list of a function definition

[3] some of the consoles will not allow you to enter more, and amongst those which do some will silently discard the excess and some will use it as the start of the next line.

[4] of unlimited length.

To remove objects the function `rm` is available:

```
> rm(x, y, z, ink, junk, temp, foo, bar)
```

All objects created during an R session can be stored permanently in a file for use in future R sessions. At the end of each R session you are given the opportunity to save all the currently available objects. If you indicate that you want to do this, the objects are written to a file called `.RData`[5] in the current directory, and the command lines used in the session are saved to a file called `.Rhistory`.

When R is started at later time from the same directory it reloads the workspace from this file. At the same time the associated commands history is reloaded.

It is recommended that you should use separate working directories for analyses conducted with R. It is quite common for objects with names `x` and `y` to be created during an analysis. Names like this are often meaningful in the context of a single analysis, but it can be quite hard to decide what they might be when the several analyses have been conducted in the same directory.

[5] The leading "dot" in this file name makes it *invisible* in normal file listings in UNIX, and in default GUI file listings on macOS and Windows.

2 Simple manipulations; numbers and vectors

2.1 Vectors and assignment

R operates on named *data structures*. The simplest such structure is the numeric *vector*, which is a single entity consisting of an ordered collection of numbers. To set up a vector named `x`, say, consisting of five numbers, namely 10.4, 5.6, 3.1, 6.4 and 21.7, use the R command

```
> x <- c(10.4, 5.6, 3.1, 6.4, 21.7)
```

This is an *assignment* statement using the *function* `c()` which in this context can take an arbitrary number of vector *arguments* and whose value is a vector got by concatenating its arguments end to end.[1]

A number occurring by itself in an expression is taken as a vector of length one.

Notice that the assignment operator (‘`<-`’), which consists of the two characters ‘`<`’ (“less than”) and ‘`-`’ (“minus”) occurring strictly side-by-side and it ‘points’ to the object receiving the value of the expression. In most contexts the ‘`=`’ operator can be used as an alternative.

Assignment can also be made using the function `assign()`. An equivalent way of making the same assignment as above is with:

```
> assign("x", c(10.4, 5.6, 3.1, 6.4, 21.7))
```

The usual operator, `<-`, can be thought of as a syntactic short-cut to this.

Assignments can also be made in the other direction, using the obvious change in the assignment operator. So the same assignment could be made using

```
> c(10.4, 5.6, 3.1, 6.4, 21.7) -> x
```

If an expression is used as a complete command, the value is printed *and lost*[2]. So now if we were to use the command

```
> 1/x
```

the reciprocals of the five values would be printed at the terminal (and the value of `x`, of course, unchanged).

The further assignment

```
> y <- c(x, 0, x)
```

would create a vector `y` with 11 entries consisting of two copies of `x` with a zero in the middle place.

2.2 Vector arithmetic

Vectors can be used in arithmetic expressions, in which case the operations are performed element by element. Vectors occurring in the same expression need not all be of the same length. If they are not, the value of the expression is a vector with the same length as the longest vector which occurs in the expression. Shorter vectors in the expression are *recycled* as often as need be (perhaps fractionally) until they match the length of the longest vector. In particular a constant is simply repeated. So with the above assignments the command

```
> v <- 2*x + y + 1
```

generates a new vector `v` of length 11 constructed by adding together, element by element, `2*x` repeated 2.2 times, `y` repeated just once, and `1` repeated 11 times.

---

[1] With other than vector types of argument, such as `list` mode arguments, the action of `c()` is rather different. See Section 6.2.1 [Concatenating lists], page 27.

[2] Actually, it is still available as `.Last.value` before any other statements are executed.

The elementary arithmetic operators are the usual `+`, `-`, `*`, `/` and `^` for raising to a power. In addition all of the common arithmetic functions are available. `log`, `exp`, `sin`, `cos`, `tan`, `sqrt`, and so on, all have their usual meaning. `max` and `min` select the largest and smallest elements of a vector respectively. `range` is a function whose value is a vector of length two, namely `c(min(x), max(x))`. `length(x)` is the number of elements in `x`, `sum(x)` gives the total of the elements in `x`, and `prod(x)` their product.

Two statistical functions are `mean(x)` which calculates the sample mean, which is the same as `sum(x)/length(x)`, and `var(x)` which gives

```
sum((x-mean(x))^2)/(length(x)-1)
```

or sample variance. If the argument to `var()` is an n-by-p matrix the value is a p-by-p sample covariance matrix got by regarding the rows as independent p-variate sample vectors.

`sort(x)` returns a vector of the same size as `x` with the elements arranged in increasing order; however there are other more flexible sorting facilities available (see `order()` or `sort.list()` which produce a permutation to do the sorting).

Note that `max` and `min` select the largest and smallest values in their arguments, even if they are given several vectors. The *parallel* maximum and minimum functions `pmax` and `pmin` return a vector (of length equal to their longest argument) that contains in each element the largest (smallest) element in that position in any of the input vectors.

For most purposes the user will not be concerned if the "numbers" in a numeric vector are integers, reals or even complex. Internally calculations are done as double precision real numbers, or double precision complex numbers if the input data are complex.

To work with complex numbers, supply an explicit complex part. Thus

```
sqrt(-17)
```

will give `NaN` and a warning, but

```
sqrt(-17+0i)
```

will do the computations as complex numbers.

2.3 Generating regular sequences

R has a number of facilities for generating commonly used sequences of numbers. For example `1:30` is the vector `c(1, 2, ..., 29, 30)`. The colon operator has high priority within an expression, so, for example `2*1:15` is the vector `c(2, 4, ..., 28, 30)`. Put `n <- 10` and compare the sequences `1:n-1` and `1:(n-1)`.

The construction `30:1` may be used to generate a sequence backwards.

The function `seq()` is a more general facility for generating sequences. It has five arguments, only some of which may be specified in any one call. The first two arguments, if given, specify the beginning and end of the sequence, and if these are the only two arguments given the result is the same as the colon operator. That is `seq(2,10)` is the same vector as `2:10`.

Arguments to `seq()`, and to many other R functions, can also be given in named form, in which case the order in which they appear is irrelevant. The first two arguments may be named `from=`*value* and `to=`*value*; thus `seq(1,30)`, `seq(from=1, to=30)` and `seq(to=30, from=1)` are all the same as `1:30`. The next two arguments to `seq()` may be named `by=`*value* and `length=`*value*, which specify a step size and a length for the sequence respectively. If neither of these is given, the default `by=1` is assumed.

For example

```
> seq(-5, 5, by=.2) -> s3
```

generates in `s3` the vector `c(-5.0, -4.8, -4.6, ..., 4.6, 4.8, 5.0)`. Similarly

```
> s4 <- seq(length=51, from=-5, by=.2)
```

generates the same vector in `s4`.

The fifth argument may be named `along=`*`vector`*, which is normally used as the only argument to create the sequence `1, 2, ..., length(`*`vector`*`)`, or the empty sequence if the vector is empty (as it can be).

A related function is `rep()` which can be used for replicating an object in various complicated ways. The simplest form is

```
> s5 <- rep(x, times=5)
```

which will put five copies of `x` end-to-end in `s5`. Another useful version is

```
> s6 <- rep(x, each=5)
```

which repeats each element of `x` five times before moving on to the next.

2.4 Logical vectors

As well as numerical vectors, R allows manipulation of logical quantities. The elements of a logical vector can have the values `TRUE`, `FALSE`, and `NA` (for "not available", see below). The first two are often abbreviated as `T` and `F`, respectively. Note however that `T` and `F` are just variables which are set to `TRUE` and `FALSE` by default, but are not reserved words and hence can be overwritten by the user. Hence, you should always use `TRUE` and `FALSE`.

Logical vectors are generated by *conditions*. For example

```
> temp <- x > 13
```

sets `temp` as a vector of the same length as `x` with values `FALSE` corresponding to elements of `x` where the condition is *not* met and `TRUE` where it is.

The logical operators are `<`, `<=`, `>`, `>=`, `==` for exact equality and `!=` for inequality. In addition if `c1` and `c2` are logical expressions, then `c1 & c2` is their intersection (*"and"*), `c1 | c2` is their union (*"or"*), and `!c1` is the negation of `c1`.

Logical vectors may be used in ordinary arithmetic, in which case they are *coerced* into numeric vectors, `FALSE` becoming `0` and `TRUE` becoming `1`. However there are situations where logical vectors and their coerced numeric counterparts are not equivalent, for example see the next subsection.

2.5 Missing values

In some cases the components of a vector may not be completely known. When an element or value is "not available" or a "missing value" in the statistical sense, a place within a vector may be reserved for it by assigning it the special value `NA`. In general any operation on an `NA` becomes an `NA`. The motivation for this rule is simply that if the specification of an operation is incomplete, the result cannot be known and hence is not available.

The function `is.na(x)` gives a logical vector of the same size as `x` with value `TRUE` if and only if the corresponding element in `x` is `NA`.

```
> z <- c(1:3,NA);  ind <- is.na(z)
```

Notice that the logical expression `x == NA` is quite different from `is.na(x)` since `NA` is not really a value but a marker for a quantity that is not available. Thus `x == NA` is a vector of the same length as `x` *all* of whose values are `NA` as the logical expression itself is incomplete and hence undecidable.

Note that there is a second kind of "missing" values which are produced by numerical computation, the so-called *Not a Number*, `NaN`, values. Examples are

```
> 0/0
```

or

```
> Inf - Inf
```

which both give `NaN` since the result cannot be defined sensibly.

In summary, `is.na(xx)` is `TRUE` *both* for `NA` and `NaN` values. To differentiate these, `is.nan(xx)` is only `TRUE` for `NaN`s.

Missing values are sometimes printed as `<NA>` when character vectors are printed without quotes.

2.6 Character vectors

Character quantities and character vectors are used frequently in R, for example as plot labels. Where needed they are denoted by a sequence of characters delimited by the double quote character, e.g., `"x-values"`, `"New iteration results"`.

Character strings are entered using either matching double (`"`) or single (`'`) quotes, but are printed using double quotes (or sometimes without quotes). They use C-style escape sequences, using `\` as the escape character, so `\` is entered and printed as `\\`, and inside double quotes `"` is entered as `\"`. Other useful escape sequences are `\n`, newline, `\t`, tab and `\b`, backspace—see `?Quotes` for a full list.

Character vectors may be concatenated into a vector by the `c()` function; examples of their use will emerge frequently.

The `paste()` function takes an arbitrary number of arguments and concatenates them one by one into character strings. Any numbers given among the arguments are coerced into character strings in the evident way, that is, in the same way they would be if they were printed. The arguments are by default separated in the result by a single blank character, but this can be changed by the named argument, `sep=`*`string`*, which changes it to *`string`*, possibly empty.

For example

```
> labs <- paste(c("X","Y"), 1:10, sep="")
```

makes `labs` into the character vector

```
c("X1", "Y2", "X3", "Y4", "X5", "Y6", "X7", "Y8", "X9", "Y10")
```

Note particularly that recycling of short lists takes place here too; thus `c("X", "Y")` is repeated 5 times to match the sequence `1:10`.[3]

2.7 Index vectors; selecting and modifying subsets of a data set

Subsets of the elements of a vector may be selected by appending to the name of the vector an *index vector* in square brackets. More generally any expression that evaluates to a vector may have subsets of its elements similarly selected by appending an index vector in square brackets immediately after the expression.

Such index vectors can be any of four distinct types.

1. **A logical vector**. In this case the index vector is recycled to the same length as the vector from which elements are to be selected. Values corresponding to `TRUE` in the index vector are selected and those corresponding to `FALSE` are omitted. For example

   ```
   > y <- x[!is.na(x)]
   ```

   creates (or re-creates) an object `y` which will contain the non-missing values of `x`, in the same order. Note that if `x` has missing values, `y` will be shorter than `x`. Also

   ```
   > (x+1)[(!is.na(x)) & x>0] -> z
   ```

   creates an object `z` and places in it the values of the vector `x+1` for which the corresponding value in `x` was both non-missing and positive.

[3] `paste(..., collapse=`*`ss`*`)` joins the arguments into a single character string putting *ss* in between, e.g., `ss <- "|"`. There are more tools for character manipulation, see the help for `sub` and `substring`.

2. **A vector of positive integral quantities**. In this case the values in the index vector must lie in the set {1, 2, . . . , `length(x)`}. The corresponding elements of the vector are selected and concatenated, *in that order*, in the result. The index vector can be of any length and the result is of the same length as the index vector. For example `x[6]` is the sixth component of `x` and

   ```
   > x[1:10]
   ```

   selects the first 10 elements of `x` (assuming `length(x)` is not less than 10). Also

   ```
   > c("x","y")[rep(c(1,2,2,1), times=4)]
   ```

   (an admittedly unlikely thing to do) produces a character vector of length 16 consisting of `"x", "y", "y", "x"` repeated four times.
3. **A vector of negative integral quantities**. Such an index vector specifies the values to be *excluded* rather than included. Thus

   ```
   > y <- x[-(1:5)]
   ```

   gives `y` all but the first five elements of `x`.
4. **A vector of character strings**. This possibility only applies where an object has a `names` attribute to identify its components. In this case a sub-vector of the names vector may be used in the same way as the positive integral labels in item 2 further above.

   ```
   > fruit <- c(5, 10, 1, 20)
   > names(fruit) <- c("orange", "banana", "apple", "peach")
   > lunch <- fruit[c("apple","orange")]
   ```

   The advantage is that alphanumeric *names* are often easier to remember than *numeric indices*. This option is particularly useful in connection with data frames, as we shall see later.

An indexed expression can also appear on the receiving end of an assignment, in which case the assignment operation is performed *only on those elements of the vector*. The expression must be of the form `vector[`*`index_vector`*`]` as having an arbitrary expression in place of the vector name does not make much sense here.

For example

```
> x[is.na(x)] <- 0
```

replaces any missing values in `x` by zeros and

```
> y[y < 0] <- -y[y < 0]
```

has the same effect as

```
> y <- abs(y)
```

2.8 Other types of objects

Vectors are the most important type of object in R, but there are several others which we will meet more formally in later sections.

- *matrices* or more generally *arrays* are multi-dimensional generalizations of vectors. In fact, they *are* vectors that can be indexed by two or more indices and will be printed in special ways. See Chapter 5 [Arrays and matrices], page 18.
- *factors* provide compact ways to handle categorical data. See Chapter 4 [Factors], page 16.
- *lists* are a general form of vector in which the various elements need not be of the same type, and are often themselves vectors or lists. Lists provide a convenient way to return the results of a statistical computation. See Section 6.1 [Lists], page 26.
- *data frames* are matrix-like structures, in which the columns can be of different types. Think of data frames as 'data matrices' with one row per observational unit but with (possibly)

both numerical and categorical variables. Many experiments are best described by data frames: the treatments are categorical but the response is numeric. See Section 6.3 [Data frames], page 27.

- *functions* are themselves objects in R which can be stored in the project's workspace. This provides a simple and convenient way to extend R. See Chapter 10 [Writing your own functions], page 41.

3 Objects, their modes and attributes

3.1 Intrinsic attributes: mode and length

The entities R operates on are technically known as *objects*. Examples are vectors of numeric (real) or complex values, vectors of logical values and vectors of character strings. These are known as "atomic" structures since their components are all of the same type, or *mode*, namely *numeric*[1], *complex*, *logical*, *character* and *raw*.

Vectors must have their values *all of the same mode*. Thus any given vector must be unambiguously either *logical*, *numeric*, *complex*, *character* or *raw*. (The only apparent exception to this rule is the special "value" listed as `NA` for quantities not available, but in fact there are several types of `NA`). Note that a vector can be empty and still have a mode. For example the empty character string vector is listed as `character(0)` and the empty numeric vector as `numeric(0)`.

R also operates on objects called *lists*, which are of mode *list*. These are ordered sequences of objects which individually can be of any mode. *lists* are known as "recursive" rather than atomic structures since their components can themselves be lists in their own right.

The other recursive structures are those of mode *function* and *expression*. Functions are the objects that form part of the R system along with similar user written functions, which we discuss in some detail later. Expressions as objects form an advanced part of R which will not be discussed in this guide, except indirectly when we discuss *formulae* used with modeling in R.

By the *mode* of an object we mean the basic type of its fundamental constituents. This is a special case of a "property" of an object. Another property of every object is its *length*. The functions `mode(`*`object`*`)` and `length(`*`object`*`)` can be used to find out the mode and length of any defined structure[2].

Further properties of an object are usually provided by `attributes(`*`object`*`)`, see Section 3.3 [Getting and setting attributes], page 14. Because of this, *mode* and *length* are also called "intrinsic attributes" of an object.

For example, if `z` is a complex vector of length 100, then in an expression `mode(z)` is the character string `"complex"` and `length(z)` is `100`.

R caters for changes of mode almost anywhere it could be considered sensible to do so, (and a few where it might not be). For example with

```
> z <- 0:9
```

we could put

```
> digits <- as.character(z)
```

after which `digits` is the character vector `c("0", "1", "2", ..., "9")`. A further *coercion*, or change of mode, reconstructs the numerical vector again:

```
> d <- as.integer(digits)
```

Now `d` and `z` are the same.[3] There is a large collection of functions of the form `as.`*`something`*`()` for either coercion from one mode to another, or for investing an object with some other attribute it may not already possess. The reader should consult the different help files to become familiar with them.

---

[1] *numeric* mode is actually an amalgam of two distinct modes, namely *integer* and *double* precision, as explained in the manual.

[2] Note however that `length(`*`object`*`)` does not always contain intrinsic useful information, e.g., when *`object`* is a function.

[3] In general, coercion from numeric to character and back again will not be exactly reversible, because of roundoff errors in the character representation.

3.2 Changing the length of an object

An "empty" object may still have a mode. For example

```
> e <- numeric()
```

makes `e` an empty vector structure of mode numeric. Similarly `character()` is a empty character vector, and so on. Once an object of any size has been created, new components may be added to it simply by giving it an index value outside its previous range. Thus

```
> e[3] <- 17
```

now makes `e` a vector of length 3, (the first two components of which are at this point both `NA`). This applies to any structure at all, provided the mode of the additional component(s) agrees with the mode of the object in the first place.

This automatic adjustment of lengths of an object is used often, for example in the `scan()` function for input. (see Section 7.2 [The scan() function], page 31.)

Conversely to truncate the size of an object requires only an assignment to do so. Hence if `alpha` is an object of length 10, then

```
> alpha <- alpha[2 * 1:5]
```

makes it an object of length 5 consisting of just the former components with even index. (The old indices are not retained, of course.) We can then retain just the first three values by

```
> length(alpha) <- 3
```

and vectors can be extended (by missing values) in the same way.

3.3 Getting and setting attributes

The function `attributes(`*`object`*`)` returns a list of all the non-intrinsic attributes currently defined for that object. The function `attr(`*`object, name`*`)` can be used to select a specific attribute. These functions are rarely used, except in rather special circumstances when some new attribute is being created for some particular purpose, for example to associate a creation date or an operator with an R object. The concept, however, is very important.

Some care should be exercised when assigning or deleting attributes since they are an integral part of the object system used in R.

When it is used on the left hand side of an assignment it can be used either to associate a new attribute with *`object`* or to change an existing one. For example

```
> attr(z, "dim") <- c(10,10)
```

allows R to treat `z` as if it were a 10-by-10 matrix.

3.4 The class of an object

All objects in R have a *class*, reported by the function `class`. For simple vectors this is just the mode, for example `"numeric"`, `"logical"`, `"character"` or `"list"`, but `"matrix"`, `"array"`, `"factor"` and `"data.frame"` are other possible values.

A special attribute known as the *class* of the object is used to allow for an object-oriented style[4] of programming in R. For example if an object has class `"data.frame"`, it will be printed in a certain way, the `plot()` function will display it graphically in a certain way, and other so-called generic functions such as `summary()` will react to it as an argument in a way sensitive to its class.

To remove temporarily the effects of class, use the function `unclass()`. For example if `winter` has the class `"data.frame"` then

```
> winter
```

[4] A different style using 'formal' or 'S4' classes is provided in package `methods`.

will print it in data frame form, which is rather like a matrix, whereas

```
> unclass(winter)
```

will print it as an ordinary list. Only in rather special situations do you need to use this facility, but one is when you are learning to come to terms with the idea of class and generic functions.

Generic functions and classes will be discussed further in Section 10.9 [Object orientation], page 47, but only briefly.

4 Ordered and unordered factors

A *factor* is a vector object used to specify a discrete classification (grouping) of the components of other vectors of the same length. R provides both *ordered* and *unordered* factors. While the "real" application of factors is with model formulae (see Section 11.1.1 [Contrasts], page 52), we here look at a specific example.

4.1 A specific example

Suppose, for example, we have a sample of 30 tax accountants from all the states and territories of Australia[1] and their individual state of origin is specified by a character vector of state mnemonics as

```
> state <- c("tas", "sa",  "qld", "nsw", "nsw", "nt",  "wa",  "wa",
             "qld", "vic", "nsw", "vic", "qld", "qld", "sa",  "tas",
             "sa",  "nt",  "wa",  "vic", "qld", "nsw", "nsw", "wa",
             "sa",  "act", "nsw", "vic", "vic", "act")
```

Notice that in the case of a character vector, "sorted" means sorted in alphabetical order.

A *factor* is similarly created using the `factor()` function:

```
> statef <- factor(state)
```

The `print()` function handles factors slightly differently from other objects:

```
> statef
 [1] tas sa  qld nsw nsw nt  wa  wa  qld vic nsw vic qld qld sa
[16] tas sa  nt  wa  vic qld nsw nsw wa  sa  act nsw vic vic act
Levels:  act nsw nt qld sa tas vic wa
```

To find out the levels of a factor the function `levels()` can be used.

```
> levels(statef)
[1] "act" "nsw" "nt"  "qld" "sa"  "tas" "vic" "wa"
```

4.2 The function `tapply()` and ragged arrays

To continue the previous example, suppose we have the incomes of the same tax accountants in another vector (in suitably large units of money)

```
> incomes <- c(60, 49, 40, 61, 64, 60, 59, 54, 62, 69, 70, 42, 56,
               61, 61, 61, 58, 51, 48, 65, 49, 49, 41, 48, 52, 46,
               59, 46, 58, 43)
```

To calculate the sample mean income for each state we can now use the special function `tapply()`:

```
> incmeans <- tapply(incomes, statef, mean)
```

giving a means vector with the components labelled by the levels

```
   act    nsw     nt    qld     sa    tas    vic     wa
44.500 57.333 55.500 53.600 55.000 60.500 56.000 52.250
```

The function `tapply()` is used to apply a function, here `mean()`, to each group of components of the first argument, here `incomes`, defined by the levels of the second component, here `statef`[2],

[1] Readers should note that there are eight states and territories in Australia, namely the Australian Capital Territory, New South Wales, the Northern Territory, Queensland, South Australia, Tasmania, Victoria and Western Australia.

[2] Note that `tapply()` also works in this case when its second argument is not a factor, e.g., '`tapply(incomes, state)`', and this is true for quite a few other functions, since arguments are *coerced* to factors when necessary (using `as.factor()`).

as if they were separate vector structures. The result is a structure of the same length as the levels attribute of the factor containing the results. The reader should consult the help document for more details.

Suppose further we needed to calculate the standard errors of the state income means. To do this we need to write an R function to calculate the standard error for any given vector. Since there is an builtin function `var()` to calculate the sample variance, such a function is a very simple one liner, specified by the assignment:

```
> stdError <- function(x) sqrt(var(x)/length(x))
```

(Writing functions will be considered later in Chapter 10 [Writing your own functions], page 41. Note that R's a builtin function `sd()` is something different.) After this assignment, the standard errors are calculated by

```
> incster <- tapply(incomes, statef, stdError)
```

and the values calculated are then

```
> incster
act    nsw  nt    qld     sa tas   vic     wa
1.5 4.3102 4.5 4.1061 2.7386 0.5 5.244 2.6575
```

As an exercise you may care to find the usual 95% confidence limits for the state mean incomes. To do this you could use `tapply()` once more with the `length()` function to find the sample sizes, and the `qt()` function to find the percentage points of the appropriate t-distributions. (You could also investigate R's facilities for t-tests.)

The function `tapply()` can also be used to handle more complicated indexing of a vector by multiple categories. For example, we might wish to split the tax accountants by both state and sex. However in this simple instance (just one factor) what happens can be thought of as follows. The values in the vector are collected into groups corresponding to the distinct entries in the factor. The function is then applied to each of these groups individually. The value is a vector of function results, labelled by the `levels` attribute of the factor.

The combination of a vector and a labelling factor is an example of what is sometimes called a *ragged array*, since the subclass sizes are possibly irregular. When the subclass sizes are all the same the indexing may be done implicitly and much more efficiently, as we see in the next section.

4.3 Ordered factors

The levels of factors are stored in alphabetical order, or in the order they were specified to `factor` if they were specified explicitly.

Sometimes the levels will have a natural ordering that we want to record and want our statistical analysis to make use of. The `ordered()` function creates such ordered factors but is otherwise identical to `factor`. For most purposes the only difference between ordered and unordered factors is that the former are printed showing the ordering of the levels, but the contrasts generated for them in fitting linear models are different.

5 Arrays and matrices

5.1 Arrays

An array can be considered as a multiply subscripted collection of data entries, for example numeric. R allows simple facilities for creating and handling arrays, and in particular the special case of matrices.

A dimension vector is a vector of non-negative integers. If its length is k then the array is k-dimensional, e.g. a matrix is a 2-dimensional array. The dimensions are indexed from one up to the values given in the dimension vector.

A vector can be used by R as an array only if it has a dimension vector as its *dim* attribute. Suppose, for example, `z` is a vector of 1500 elements. The assignment

```
> dim(z) <- c(3,5,100)
```

gives it the *dim* attribute that allows it to be treated as a 3 by 5 by 100 array.

Other functions such as `matrix()` and `array()` are available for simpler and more natural looking assignments, as we shall see in Section 5.4 [The array() function], page 20.

The values in the data vector give the values in the array in the same order as they would occur in FORTRAN, that is "column major order," with the first subscript moving fastest and the last subscript slowest.

For example if the dimension vector for an array, say `a`, is `c(3,4,2)` then there are $3 \times 4 \times 2 = 24$ entries in `a` and the data vector holds them in the order `a[1,1,1], a[2,1,1], ..., a[2,4,2], a[3,4,2]`.

Arrays can be one-dimensional: such arrays are usually treated in the same way as vectors (including when printing), but the exceptions can cause confusion.

5.2 Array indexing. Subsections of an array

Individual elements of an array may be referenced by giving the name of the array followed by the subscripts in square brackets, separated by commas.

More generally, subsections of an array may be specified by giving a sequence of *index vectors* in place of subscripts; however *if any index position is given an empty index vector, then the full range of that subscript is taken.*

Continuing the previous example, `a[2,,]` is a 4×2 array with dimension vector `c(4,2)` and data vector containing the values

```
c(a[2,1,1], a[2,2,1], a[2,3,1], a[2,4,1],
  a[2,1,2], a[2,2,2], a[2,3,2], a[2,4,2])
```

in that order. `a[,,]` stands for the entire array, which is the same as omitting the subscripts entirely and using `a` alone.

For any array, say `Z`, the dimension vector may be referenced explicitly as `dim(Z)` (on either side of an assignment).

Also, if an array name is given with just *one subscript or index vector*, then the corresponding values of the data vector only are used; in this case the dimension vector is ignored. This is not the case, however, if the single index is not a vector but itself an array, as we next discuss.

5.3 Index matrices

As well as an index vector in any subscript position, a matrix may be used with a single *index matrix* in order either to assign a vector of quantities to an irregular collection of elements in the array, or to extract an irregular collection as a vector.

A matrix example makes the process clear. In the case of a doubly indexed array, an index matrix may be given consisting of two columns and as many rows as desired. The entries in the index matrix are the row and column indices for the doubly indexed array. Suppose for example we have a 4 by 5 array `X` and we wish to do the following:

- Extract elements `X[1,3]`, `X[2,2]` and `X[3,1]` as a vector structure, and
- Replace these entries in the array `X` by zeroes.

In this case we need a 3 by 2 subscript array, as in the following example.

```
> x <- array(1:20, dim=c(4,5))    # Generate a 4 by 5 array.
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
> i <- array(c(1:3,3:1), dim=c(3,2))
> i                               # i is a 3 by 2 index array.
     [,1] [,2]
[1,]    1    3
[2,]    2    2
[3,]    3    1
> x[i]                            # Extract those elements
[1] 9 6 3
> x[i] <- 0                       # Replace those elements by zeros.
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    0   13   17
[2,]    2    0   10   14   18
[3,]    0    7   11   15   19
[4,]    4    8   12   16   20
>
```

Negative indices are not allowed in index matrices. `NA` and zero values are allowed: rows in the index matrix containing a zero are ignored, and rows containing an `NA` produce an `NA` in the result.

As a less trivial example, suppose we wish to generate an (unreduced) design matrix for a block design defined by factors `blocks` (`b` levels) and `varieties` (`v` levels). Further suppose there are `n` plots in the experiment. We could proceed as follows:

```
> Xb <- matrix(0, n, b)
> Xv <- matrix(0, n, v)
> ib <- cbind(1:n, blocks)
> iv <- cbind(1:n, varieties)
> Xb[ib] <- 1
> Xv[iv] <- 1
> X <- cbind(Xb, Xv)
```

To construct the incidence matrix, `N` say, we could use

```
> N <- crossprod(Xb, Xv)
```

However a simpler direct way of producing this matrix is to use `table()`:

```
> N <- table(blocks, varieties)
```

Index matrices must be numerical: any other form of matrix (e.g. a logical or character matrix) supplied as a matrix is treated as an indexing vector.

5.4 The `array()` function

As well as giving a vector structure a `dim` attribute, arrays can be constructed from vectors by the `array` function, which has the form

```
> Z <- array(data_vector, dim_vector)
```

For example, if the vector `h` contains 24 or fewer, numbers then the command

```
> Z <- array(h, dim=c(3,4,2))
```

would use `h` to set up 3 by 4 by 2 array in `Z`. If the size of `h` is exactly 24 the result is the same as

```
> Z <- h ; dim(Z) <- c(3,4,2)
```

However if `h` is shorter than 24, its values are recycled from the beginning again to make it up to size 24 (see Section 5.4.1 [The recycling rule], page 20) but `dim(h) <- c(3,4,2)` would signal an error about mismatching length. As an extreme but common example

```
> Z <- array(0, c(3,4,2))
```

makes `Z` an array of all zeros.

At this point `dim(Z)` stands for the dimension vector `c(3,4,2)`, and `Z[1:24]` stands for the data vector as it was in `h`, and `Z[]` with an empty subscript or `Z` with no subscript stands for the entire array as an array.

Arrays may be used in arithmetic expressions and the result is an array formed by element-by-element operations on the data vector. The `dim` attributes of operands generally need to be the same, and this becomes the dimension vector of the result. So if `A`, `B` and `C` are all similar arrays, then

```
> D <- 2*A*B + C + 1
```

makes `D` a similar array with its data vector being the result of the given element-by-element operations. However the precise rule concerning mixed array and vector calculations has to be considered a little more carefully.

5.4.1 Mixed vector and array arithmetic. The recycling rule

The precise rule affecting element by element mixed calculations with vectors and arrays is somewhat quirky and hard to find in the references. From experience we have found the following to be a reliable guide.

- The expression is scanned from left to right.
- Any short vector operands are extended by recycling their values until they match the size of any other operands.
- As long as short vectors and arrays *only* are encountered, the arrays must all have the same `dim` attribute or an error results.
- Any vector operand longer than a matrix or array operand generates an error.
- If array structures are present and no error or coercion to vector has been precipitated, the result is an array structure with the common `dim` attribute of its array operands.

5.5 The outer product of two arrays

An important operation on arrays is the *outer product*. If `a` and `b` are two numeric arrays, their outer product is an array whose dimension vector is obtained by concatenating their two dimension vectors (order is important), and whose data vector is got by forming all possible products of elements of the data vector of `a` with those of `b`. The outer product is formed by the special operator `%o%`:

```
> ab <- a %o% b
```

An alternative is

```
> ab <- outer(a, b, "*")
```

The multiplication function can be replaced by an arbitrary function of two variables. For example if we wished to evaluate the function $f(x; y) = \cos(y)/(1 + x^2)$ over a regular grid of values with x- and y-coordinates defined by the R vectors `x` and `y` respectively, we could proceed as follows:

```
> f <- function(x, y) cos(y)/(1 + x^2)
> z <- outer(x, y, f)
```

In particular the outer product of two ordinary vectors is a doubly subscripted array (that is a matrix, of rank at most 1). Notice that the outer product operator is of course non-commutative. Defining your own R functions will be considered further in Chapter 10 [Writing your own functions], page 41.

An example: Determinants of 2 by 2 single-digit matrices

As an artificial but cute example, consider the determinants of 2 by 2 matrices $[a, b; c, d]$ where each entry is a non-negative integer in the range $0, 1, \ldots, 9$, that is a digit.

The problem is to find the determinants, $ad - bc$, of all possible matrices of this form and represent the frequency with which each value occurs as a *high density* plot. This amounts to finding the probability distribution of the determinant if each digit is chosen independently and uniformly at random.

A neat way of doing this uses the `outer()` function twice:

```
> d <- outer(0:9, 0:9)
> fr <- table(outer(d, d, "-"))
> plot(fr, xlab="Determinant", ylab="Frequency")
```

Notice that `plot()` here uses a histogram like plot method, because it "sees" that `fr` is of class `"table"`. The "obvious" way of doing this problem with `for` loops, to be discussed in Chapter 9 [Loops and conditional execution], page 39, is so inefficient as to be impractical.

It is also perhaps surprising that about 1 in 20 such matrices is singular.

5.6 Generalized transpose of an array

The function `aperm(a, perm)` may be used to permute an array, `a`. The argument `perm` must be a permutation of the integers $\{1, \ldots, k\}$, where k is the number of subscripts in `a`. The result of the function is an array of the same size as `a` but with old dimension given by `perm[j]` becoming the new `j`-th dimension. The easiest way to think of this operation is as a generalization of transposition for matrices. Indeed if `A` is a matrix, (that is, a doubly subscripted array) then `B` given by

```
> B <- aperm(A, c(2,1))
```

is just the transpose of `A`. For this special case a simpler function `t()` is available, so we could have used `B <- t(A)`.

5.7 Matrix facilities

As noted above, a matrix is just an array with two subscripts. However it is such an important special case it needs a separate discussion. R contains many operators and functions that are available only for matrices. For example `t(X)` is the matrix transpose function, as noted above. The functions `nrow(A)` and `ncol(A)` give the number of rows and columns in the matrix `A` respectively.

5.7.1 Matrix multiplication

The operator `%*%` is used for matrix multiplication. An n by 1 or 1 by n matrix may of course be used as an n-vector if in the context such is appropriate. Conversely, vectors which occur in matrix multiplication expressions are automatically promoted either to row or column vectors, whichever is multiplicatively coherent, if possible, (although this is not always unambiguously possible, as we see later).

If, for example, `A` and `B` are square matrices of the same size, then

```
> A * B
```

is the matrix of element by element products and

```
> A %*% B
```

is the matrix product. If `x` is a vector, then

```
> x %*% A %*% x
```

is a quadratic form.[1]

The function `crossprod()` forms "cross products", meaning that `crossprod(X, y)` is the same as `t(X) %*% y` but the operation is more efficient. If the second argument to `crossprod()` is omitted it is taken to be the same as the first.

The meaning of `diag()` depends on its argument. `diag(v)`, where `v` is a vector, gives a diagonal matrix with elements of the vector as the diagonal entries. On the other hand `diag(M)`, where `M` is a matrix, gives the vector of main diagonal entries of `M`. This is the same convention as that used for `diag()` in MATLAB. Also, somewhat confusingly, if `k` is a single numeric value then `diag(k)` is the `k` by `k` identity matrix!

5.7.2 Linear equations and inversion

Solving linear equations is the inverse of matrix multiplication. When after

```
> b <- A %*% x
```

only `A` and `b` are given, the vector `x` is the solution of that linear equation system. In R,

```
> solve(A,b)
```

solves the system, returning `x` (up to some accuracy loss). Note that in linear algebra, formally $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ where $\mathbf{A}^{-1}$ denotes the *inverse* of $\mathbf{A}$, which can be computed by

```
solve(A)
```

but rarely is needed. Numerically, it is both inefficient and potentially unstable to compute `x <- solve(A) %*% b` instead of `solve(A,b)`.

The quadratic form $\mathbf{x}^{\mathbf{T}}\mathbf{A}^{-1}\mathbf{x}$ which is used in multivariate computations, should be computed by something like[2] `x %*% solve(A,x)`, rather than computing the inverse of `A`.

---

[1] Note that `x %*% x` is ambiguous, as it could mean either $\mathbf{x}^T\mathbf{x}$ or $\mathbf{x}\mathbf{x}^T$, where `x` is the column form. In such cases the smaller matrix seems implicitly to be the interpretation adopted, so the scalar $\mathbf{x}^T\mathbf{x}$ is in this case the result. The matrix $\mathbf{x}\mathbf{x}^T$ may be calculated either by `cbind(x) %*% x` or `x %*% rbind(x)` since the result of `rbind()` or `cbind()` is always a matrix. However, the best way to compute $\mathbf{x}^T\mathbf{x}$ or $\mathbf{x}\mathbf{x}^T$ is `crossprod(x)` or `x %o% x` respectively.

[2] Even better would be to form a matrix square root B with $A = BB^T$ and find the squared length of the solution of $By = x$, perhaps using the Cholesky or eigendecomposition of A.

5.7.3 Eigenvalues and eigenvectors

The function `eigen(Sm)` calculates the eigenvalues and eigenvectors of a symmetric matrix `Sm`. The result of this function is a list of two components named `values` and `vectors`. The assignment

```
> ev <- eigen(Sm)
```

will assign this list to `ev`. Then `ev$val` is the vector of eigenvalues of `Sm` and `ev$vec` is the matrix of corresponding eigenvectors. Had we only needed the eigenvalues we could have used the assignment:

```
> evals <- eigen(Sm)$values
```

`evals` now holds the vector of eigenvalues and the second component is discarded. If the expression

```
> eigen(Sm)
```

is used by itself as a command the two components are printed, with their names. For large matrices it is better to avoid computing the eigenvectors if they are not needed by using the expression

```
> evals <- eigen(Sm, only.values = TRUE)$values
```

5.7.4 Singular value decomposition and determinants

The function `svd(M)` takes an arbitrary matrix argument, `M`, and calculates the singular value decomposition of `M`. This consists of a matrix of orthonormal columns `U` with the same column space as `M`, a second matrix of orthonormal columns `V` whose column space is the row space of `M` and a diagonal matrix of positive entries `D` such that `M = U %*% D %*% t(V)`. `D` is actually returned as a vector of the diagonal elements. The result of `svd(M)` is actually a list of three components named `d`, `u` and `v`, with evident meanings.

If `M` is in fact square, then, it is not hard to see that

```
> absdetM <- prod(svd(M)$d)
```

calculates the absolute value of the determinant of `M`. If this calculation were needed often with a variety of matrices it could be defined as an R function

```
> absdet <- function(M) prod(svd(M)$d)
```

after which we could use `absdet()` as just another R function. As a further trivial but potentially useful example, you might like to consider writing a function, say `tr()`, to calculate the trace of a square matrix. [Hint: You will not need to use an explicit loop. Look again at the `diag()` function.]

R has a builtin function `det` to calculate a determinant, including the sign, and another, `determinant`, to give the sign and modulus (optionally on log scale),

5.7.5 Least squares fitting and the QR decomposition

The function `lsfit()` returns a list giving results of a least squares fitting procedure. An assignment such as

```
> ans <- lsfit(X, y)
```

gives the results of a least squares fit where `y` is the vector of observations and `X` is the design matrix. See the help facility for more details, and also for the follow-up function `ls.diag()` for, among other things, regression diagnostics. Note that a grand mean term is automatically included and need not be included explicitly as a column of `X`. Further note that you almost always will prefer using `lm(.)` (see Section 11.2 [Linear models], page 53) to `lsfit()` for regression modelling.

Another closely related function is `qr()` and its allies. Consider the following assignments

```
> Xplus <- qr(X)
```

```
> b <- qr.coef(Xplus, y)
> fit <- qr.fitted(Xplus, y)
> res <- qr.resid(Xplus, y)
```

These compute the orthogonal projection of `y` onto the range of `X` in `fit`, the projection onto the orthogonal complement in `res` and the coefficient vector for the projection in `b`, that is, `b` is essentially the result of the MATLAB 'backslash' operator.

It is not assumed that `X` has full column rank. Redundancies will be discovered and removed as they are found.

This alternative is the older, low-level way to perform least squares calculations. Although still useful in some contexts, it would now generally be replaced by the statistical models features, as will be discussed in Chapter 11 [Statistical models in R], page 50.

5.8 Forming partitioned matrices, `cbind()` and `rbind()`

As we have already seen informally, matrices can be built up from other vectors and matrices by the functions `cbind()` and `rbind()`. Roughly `cbind()` forms matrices by binding together matrices horizontally, or column-wise, and `rbind()` vertically, or row-wise.

In the assignment

```
> X <- cbind(arg_1, arg_2, arg_3, ...)
```

the arguments to `cbind()` must be either vectors of any length, or matrices with the same column size, that is the same number of rows. The result is a matrix with the concatenated arguments *arg_1*, *arg_2*, ... forming the columns.

If some of the arguments to `cbind()` are vectors they may be shorter than the column size of any matrices present, in which case they are cyclically extended to match the matrix column size (or the length of the longest vector if no matrices are given).

The function `rbind()` does the corresponding operation for rows. In this case any vector argument, possibly cyclically extended, are of course taken as row vectors.

Suppose `X1` and `X2` have the same number of rows. To combine these by columns into a matrix `X`, together with an initial column of `1`s we can use

```
> X <- cbind(1, X1, X2)
```

The result of `rbind()` or `cbind()` always has matrix status. Hence `cbind(x)` and `rbind(x)` are possibly the simplest ways explicitly to allow the vector `x` to be treated as a column or row matrix respectively.

5.9 The concatenation function, `c()`, with arrays

It should be noted that whereas `cbind()` and `rbind()` are concatenation functions that respect `dim` attributes, the basic `c()` function does not, but rather clears numeric objects of all `dim` and `dimnames` attributes. This is occasionally useful in its own right.

The official way to coerce an array back to a simple vector object is to use `as.vector()`

```
> vec <- as.vector(X)
```

However a similar result can be achieved by using `c()` with just one argument, simply for this side-effect:

```
> vec <- c(X)
```

There are slight differences between the two, but ultimately the choice between them is largely a matter of style (with the former being preferable).

5.10 Frequency tables from factors

Recall that a factor defines a partition into groups. Similarly a pair of factors defines a two way cross classification, and so on. The function `table()` allows frequency tables to be calculated from equal length factors. If there are k factor arguments, the result is a k-way array of frequencies.

Suppose, for example, that `statef` is a factor giving the state code for each entry in a data vector. The assignment

```
> statefr <- table(statef)
```

gives in `statefr` a table of frequencies of each state in the sample. The frequencies are ordered and labelled by the `levels` attribute of the factor. This simple case is equivalent to, but more convenient than,

```
> statefr <- tapply(statef, statef, length)
```

Further suppose that `incomef` is a factor giving a suitably defined "income class" for each entry in the data vector, for example with the `cut()` function:

```
> factor(cut(incomes, breaks = 35+10*(0:7))) -> incomef
```

Then to calculate a two-way table of frequencies:

```
> table(incomef,statef)
          statef
incomef   act nsw nt qld sa tas vic wa
  (35,45]   1   1  0   1  0   0   1  0
  (45,55]   1   1  1   1  2   0   1  3
  (55,65]   0   3  1   3  2   2   2  1
  (65,75]   0   1  0   0  0   0   1  0
```

Extension to higher-way frequency tables is immediate.

6 Lists and data frames

6.1 Lists

An R *list* is an object consisting of an ordered collection of objects known as its *components*.

There is no particular need for the components to be of the same mode or type, and, for example, a list could consist of a numeric vector, a logical value, a matrix, a complex vector, a character array, a function, and so on. Here is a simple example of how to make a list:

```
> Lst <- list(name="Fred", wife="Mary", no.children=3,
              child.ages=c(4,7,9))
```

Components are always *numbered* and may always be referred to as such. Thus if `Lst` is the name of a list with four components, these may be individually referred to as `Lst[[1]]`, `Lst[[2]]`, `Lst[[3]]` and `Lst[[4]]`. If, further, `Lst[[4]]` is a vector subscripted array then `Lst[[4]][1]` is its first entry.

If `Lst` is a list, then the function `length(Lst)` gives the number of (top level) components it has.

Components of lists may also be *named*, and in this case the component may be referred to either by giving the component name as a character string in place of the number in double square brackets, or, more conveniently, by giving an expression of the form

```
> name$component_name
```

for the same thing.

This is a very useful convention as it makes it easier to get the right component if you forget the number.

So in the simple example given above:

`Lst$name` is the same as `Lst[[1]]` and is the string `"Fred"`,

`Lst$wife` is the same as `Lst[[2]]` and is the string `"Mary"`,

`Lst$child.ages[1]` is the same as `Lst[[4]][1]` and is the number `4`.

Additionally, one can also use the names of the list components in double square brackets, i.e., `Lst[["name"]]` is the same as `Lst$name`. This is especially useful, when the name of the component to be extracted is stored in another variable as in

```
> x <- "name"; Lst[[x]]
```

It is very important to distinguish `Lst[[1]]` from `Lst[1]`. '`[[...]]`' is the operator used to select a single element, whereas '`[...]`' is a general subscripting operator. Thus the former is the *first object in the list* `Lst`, and if it is a named list the name is *not* included. The latter is a *sublist of the list* `Lst` *consisting of the first entry only. If it is a named list, the names are transferred to the sublist.*

The names of components may be abbreviated down to the minimum number of letters needed to identify them uniquely. Thus `Lst$coefficients` may be minimally specified as `Lst$coe` and `Lst$covariance` as `Lst$cov`.

The vector of names is in fact simply an attribute of the list like any other and may be handled as such. Other structures besides lists may, of course, similarly be given a *names* attribute also.

6.2 Constructing and modifying lists

New lists may be formed from existing objects by the function `list()`. An assignment of the form

```
> Lst <- list(name_1=object_1, ..., name_m=object_m)
```

sets up a list `Lst` of m components using *object_1*, ..., *object_m* for the components and giving them names as specified by the argument names, (which can be freely chosen). If these names are omitted, the components are numbered only. The components used to form the list are *copied* when forming the new list and the originals are not affected.

Lists, like any subscripted object, can be extended by specifying additional components. For example

```
> Lst[5] <- list(matrix=Mat)
```

6.2.1 Concatenating lists

When the concatenation function `c()` is given list arguments, the result is an object of mode list also, whose components are those of the argument lists joined together in sequence.

```
> list.ABC <- c(list.A, list.B, list.C)
```

Recall that with vector objects as arguments the concatenation function similarly joined together all arguments into a single vector structure. In this case all other attributes, such as `dim` attributes, are discarded.

6.3 Data frames

A *data frame* is a list with class `"data.frame"`. There are restrictions on lists that may be made into data frames, namely

- The components must be vectors (numeric, character, or logical), factors, numeric matrices, lists, or other data frames.
- Matrices, lists, and data frames provide as many variables to the new data frame as they have columns, elements, or variables, respectively.
- Vector structures appearing as variables of the data frame must all have the *same length*, and matrix structures must all have the *same number of rows*.

A data frame may for many purposes be regarded as a matrix with columns possibly of differing modes and attributes. It may be displayed in matrix form, and its rows and columns extracted using matrix indexing conventions.

6.3.1 Making data frames

Objects satisfying the restrictions placed on the columns (components) of a data frame may be used to form one using the function `data.frame`:

```
> accountants <- data.frame(home=statef, loot=incomes, shot=incomef)
```

A list whose components conform to the restrictions of a data frame may be *coerced* into a data frame using the function `as.data.frame()`

The simplest way to construct a data frame from scratch is to use the `read.table()` function to read an entire data frame from an external file. This is discussed further in Chapter 7 [Reading data from files], page 30.

6.3.2 `attach()` and `detach()`

The `$` notation, such as `accountants$home`, for list components is not always very convenient. A useful facility would be somehow to make the components of a list or data frame temporarily visible as variables under their component name, without the need to quote the list name explicitly each time.

The `attach()` function takes a 'database' such as a list or data frame as its argument. Thus suppose `lentils` is a data frame with three variables `lentils$u`, `lentils$v`, `lentils$w`. The attach

```
> attach(lentils)
```

places the data frame in the search path at position 2, and provided there are no variables `u`, `v` or `w` in position 1, `u`, `v` and `w` are available as variables from the data frame in their own right. At this point an assignment such as

```
> u <- v+w
```

does not replace the component `u` of the data frame, but rather masks it with another variable `u` in the workspace at position 1 on the search path. To make a permanent change to the data frame itself, the simplest way is to resort once again to the `$` notation:

```
> lentils$u <- v+w
```

However the new value of component `u` is not visible until the data frame is detached and attached again.

To detach a data frame, use the function

```
> detach()
```

More precisely, this statement detaches from the search path the entity currently at position 2. Thus in the present context the variables `u`, `v` and `w` would be no longer visible, except under the list notation as `lentils$u` and so on. Entities at positions greater than 2 on the search path can be detached by giving their number to `detach`, but it is much safer to always use a name, for example by `detach(lentils)` or `detach("lentils")`

> **Note:** In R lists and data frames can only be attached at position 2 or above, and what is attached is a *copy* of the original object. You can alter the attached values *via* `assign`, but the original list or data frame is unchanged.

6.3.3 Working with data frames

A useful convention that allows you to work with many different problems comfortably together in the same workspace is

- gather together all variables for any well defined and separate problem in a data frame under a suitably informative name;
- when working with a problem attach the appropriate data frame at position 2, and use the workspace at level 1 for operational quantities and temporary variables;
- before leaving a problem, add any variables you wish to keep for future reference to the data frame using the `$` form of assignment, and then `detach()`;
- finally remove all unwanted variables from the workspace and keep it as clean of left-over temporary variables as possible.

In this way it is quite simple to work with many problems in the same directory, all of which have variables named `x`, `y` and `z`, for example.

6.3.4 Attaching arbitrary lists

`attach()` is a generic function that allows not only directories and data frames to be attached to the search path, but other classes of object as well. In particular any object of mode `"list"` may be attached in the same way:

```
> attach(any.old.list)
```

Anything that has been attached can be detached by `detach`, by position number or, preferably, by name.

6.3.5 Managing the search path

The function `search` shows the current search path and so is a very useful way to keep track of which data frames and lists (and packages) have been attached and detached. Initially it gives

```
> search()
[1] ".GlobalEnv"   "Autoloads"    "package:base"
```

where `.GlobalEnv` is the workspace.[1]

After `lentils` is attached we have

```
> search()
[1] ".GlobalEnv"   "lentils"      "Autoloads"    "package:base"
> ls(2)
[1] "u" "v" "w"
```

and as we see `ls` (or `objects`) can be used to examine the contents of any position on the search path.

Finally, we detach the data frame and confirm it has been removed from the search path.

```
> detach("lentils")
> search()
[1] ".GlobalEnv"   "Autoloads"    "package:base"
```

[1] See the on-line help for `autoload` for the meaning of the second term.

7 Reading data from files

Large data objects will usually be read as values from external files rather than entered during an R session at the keyboard. R input facilities are simple and their requirements are fairly strict and even rather inflexible. There is a clear presumption by the designers of R that you will be able to modify your input files using other tools, such as file editors or Perl[1] to fit in with the requirements of R. Generally this is very simple.

If variables are to be held mainly in data frames, as we strongly suggest they should be, an entire data frame can be read directly with the `read.table()` function. There is also a more primitive input function, `scan()`, that can be called directly.

For more details on importing data into R and also exporting data, see the *R Data Import/Export* manual.

7.1 The `read.table()` function

To read an entire data frame directly, the external file will normally have a special form.

- The first line of the file should have a *name* for each variable in the data frame.
- Each additional line of the file has as its first item a *row label* and the values for each variable.

If the file has one fewer item in its first line than in its second, this arrangement is presumed to be in force. So the first few lines of a file to be read as a data frame might look as follows.

Input file form with names and row labels:

```
     Price    Floor     Area   Rooms     Age  Cent.heat
01   52.00    111.0      830     5       6.2      no
02   54.75    128.0      710     5       7.5      no
03   57.50    101.0     1000     5       4.2      no
04   57.50    131.0      690     6       8.8      no
05   59.75     93.0      900     5       1.9     yes
...
```

By default numeric items (except row labels) are read as numeric variables and non-numeric variables, such as `Cent.heat` in the example, as character variables. This can be changed if necessary.

The function `read.table()` can then be used to read the data frame directly

```
> HousePrice <- read.table("houses.data")
```

Often you will want to omit including the row labels directly and use the default labels. In this case the file may omit the row label column as in the following.

[1] Under UNIX, the utilities `sed` or `awk` can be used.

Input file form without row labels:

```
Price    Floor     Area   Rooms     Age  Cent.heat
52.00    111.0      830       5     6.2       no
54.75    128.0      710       5     7.5       no
57.50    101.0     1000       5     4.2       no
57.50    131.0      690       6     8.8       no
59.75     93.0      900       5     1.9      yes
...
```

The data frame may then be read as

```
> HousePrice <- read.table("houses.data", header=TRUE)
```

where the `header=TRUE` option specifies that the first line is a line of headings, and hence, by implication from the form of the file, that no explicit row labels are given.

7.2 The `scan()` function

Suppose the data vectors are of equal length and are to be read in parallel. Further suppose that there are three vectors, the first of mode character and the remaining two of mode numeric, and the file is `input.dat`. The first step is to use `scan()` to read in the three vectors as a list, as follows

```
> inp <- scan("input.dat", list("",0,0))
```

The second argument is a dummy list structure that establishes the mode of the three vectors to be read. The result, held in `inp`, is a list whose components are the three vectors read in. To separate the data items into three separate vectors, use assignments like

```
> label <- inp[[1]]; x <- inp[[2]]; y <- inp[[3]]
```

More conveniently, the dummy list can have named components, in which case the names can be used to access the vectors read in. For example

```
> inp <- scan("input.dat", list(id="", x=0, y=0))
```

If you wish to access the variables separately they may either be re-assigned to variables in the working frame:

```
> label <- inp$id; x <- inp$x; y <- inp$y
```

or the list may be attached at position 2 of the search path (see Section 6.3.4 [Attaching arbitrary lists], page 28).

If the second argument is a single value and not a list, a single vector is read in, all components of which must be of the same mode as the dummy value.

```
> X <- matrix(scan("light.dat", 0), ncol=5, byrow=TRUE)
```

There are more elaborate input facilities available and these are detailed in the manuals.

7.3 Accessing builtin datasets

Around 100 datasets are supplied with R (in package **datasets**), and others are available in packages (including the recommended packages supplied with R). To see the list of datasets currently available use

```
data()
```

All the datasets supplied with R are available directly by name. However, many packages still use the obsolete convention in which `data` was also used to load datasets into R, for example

```
data(infert)
```

and this can still be used with the standard packages (as in this example). In most cases this will load an R object of the same name. However, in a few cases it loads several objects, so see the on-line help for the object to see what to expect.

7.3.1 Loading data from other R packages

To access data from a particular package, use the `package` argument, for example

```
data(package="rpart")
data(Puromycin, package="datasets")
```

If a package has been attached by `library`, its datasets are automatically included in the search.

User-contributed packages can be a rich source of datasets.

7.4 Editing data

When invoked on a data frame or matrix, `edit` brings up a separate spreadsheet-like environment for editing. This is useful for making small changes once a data set has been read. The command

```
> xnew <- edit(xold)
```

will allow you to edit your data set `xold`, and on completion the changed object is assigned to `xnew`. If you want to alter the original dataset `xold`, the simplest way is to use `fix(xold)`, which is equivalent to `xold <- edit(xold)`.

Use

```
> xnew <- edit(data.frame())
```

to enter new data via the spreadsheet interface.