Chapter 2

Using R

This chapter covers the essentials for using R to explore data interactively. Section 2.1 covers basic access to an R session. Users interact with R through a single language for both data analysis and programming (Section 2.3, page 19). The key concepts are function calls in the language and the objects created and used by those calls (2.4, 24), two concepts that recur throughout the book. The huge body of available software is organized around packages that can be attached to the session, once they are installed (2.5, 25). The system itself can be downloaded and installed from repositories on the Web (2.6, 29); there are also a number of resources on the Web for information about R (2.7, 31).

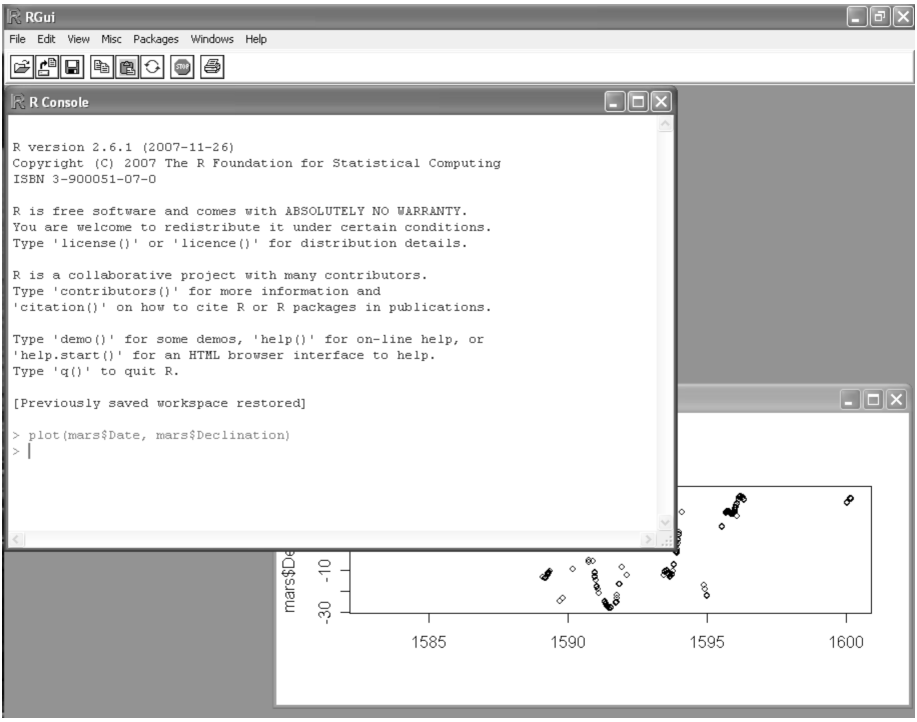
Lastly, we examine aspects of R that may raise difficulties for some new users (2.8, 34).

2.1 Starting R

R runs on the commonly used platforms for personal computing: Windows®, Mac OS X®, Linux, and some versions of UNIX®. In the usual desktop environments for these platforms, users will typically start R as they would most applications, by clicking on the R icon or on the R file in a folder of applications.

An application will then appear looking much like other applications on the platform: for example, a window and associated toolbar. In the

standard version, at least on most platforms, the application is called the `"R Console"`. In Windows recently it looked like this:



The application has a number of drop-down menus; some are typical of most applications (`"File"`, `"Edit"`, and `"Help"`). Others such as `"Packages"` are special to R. The real action in running R, however, is not with the menus but in the console window itself. Here the user is expected to type input to R in the form of expressions; the program underlying the application responds by doing some computation and if appropriate by displaying a version of the results for the user to look at (printed results normally in the same console window, graphics typically in another window).

This interaction between user and system continues, and constitutes an R session. The session is the fundamental user interface to R. The following section describes the logic behind it. A session has a simple model for user interaction, but one that is fundamentally different from users' most common experience with personal computers (in applications such as word processors, Web browsers, or audio/video systems). First-time users may feel abandoned, left to flounder on their own with little guidance about what to do and even less help when they do something wrong. More guidance is available than may be obvious, but such users are not entirely wrong in their

reaction. After intervening sections present the essential concepts involved in using R, Section 2.8, page 34 revisits this question.

2.2 An Interactive Session

Everything that you do interactively with R happens in a *session*. A session starts when you start up R, typically as described above. A session can also be started from other special interfaces or from a command shell (the original design), without changing the fundamental concept and with the basic appearance remaining as shown in this section and in the rest of the book. Some other interfaces arise in customizing the session, on page 17.

During an R session, you (the user) provide expressions for evaluation by R, for the purpose of doing any sort of computation, displaying results, and creating objects for further use. The session ends when you decide to quit from R.

All the expressions evaluated in the session are just that: general *expressions* in R's version of the S language. Documentation may mention "commands" in R, but the term just refers to a complete expression that you type interactively or otherwise hand to R for evaluation. There's only one language, used for either interactive data analysis or for programming, and described in section 2.3. Later sections in the book come back to examine it in more detail, especially in Chapter 3.

The R evaluator displays a prompt, and the user responds by typing a line of text. Printed output from the evaluation and other messages appear following the input line.

Examples in the book will be displayed in this form, with the default prompts preceding the user's input:

```
> quantile(Declination)
    0%    25%    50%    75%   100%
-27.98 -11.25   8.56  17.46  27.30
```

The `"> "` at the beginning of the example is the (default) prompt string. In this example the user responded with

```
quantile(Declination)
```

The evaluator will keep prompting until the input can be interpreted as a complete expression; if the user had left off the closing `")"`, the evaluator would have prompted for more input. Since the input here is a complete expression, the system evaluated it. To be pedantic, it parsed the input text

and evaluated the resulting object. The evaluation in this case amounts to calling a function named `quantile`.

The printed output may suggest a table, and that's intentional. But in fact nothing special happened; the standard action by the evaluator is to print the object that is the value of the expression. All evaluated expressions are objects; the printed output corresponds to the object; specifically, the form of printed output is determined by the kind of object, by its *class* (technically, through a method selected for that class). The call to `quantile()` returned a numeric vector, that is, an object of class `"numeric"`. A method was selected based on this class, and the method was called to print the result shown. The `quantile()` function expects a vector of numbers as its argument; with just this one argument it returns a numeric vector containing the minimum, maximum, median and quartiles.

The method for printing numeric vectors prints the values in the vector, five of them in this case. Numeric objects can optionally have a `names` attribute; if they do, the method prints the names as labels above the numbers. So the `"0%"` and so on are part of the object. The designer of the `quantile()` function helpfully chose a names attribute for the result that makes it easier to interpret when printed.

All these details are unimportant if you're just calling `quantile()` to summarize some data, but the important general concept is this: Objects are the center of computations in R, along with the function calls that create and use those objects. The duality of objects and function calls will recur in many of our discussions.

Computing with existing software hinges largely on using and creating objects, via the large number of available functions. Programming, that is, creating *new* software, starts with the simple creation of function objects. More ambitious projects often use a paradigm of creating new classes of objects, along with new or modified functions and methods that link the functions and classes. In all the details of programming, the fundamental duality of objects and functions remains an underlying concept.

Essentially all expressions are evaluated as function calls, but the language includes some forms that don't look like function calls. Included are the usual operators, such as arithmetic, discussed on page 21. Another useful operator is `` `?` ``, which looks up R help for the topic that follows the question mark. To learn about the function `quantile()`:

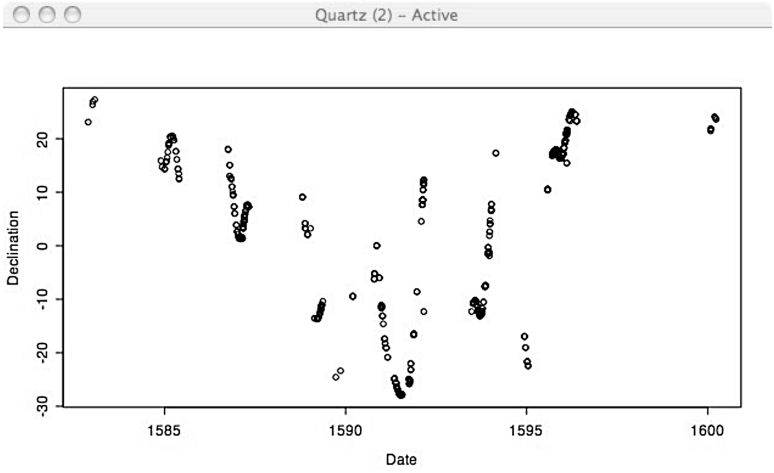
```
> ?quantile
```

In standard GUI interfaces, the documentation will appear in a separate window, and can be generated from a pull-down menu as well as from the

`?` operator.

Graphical displays provide some of the most powerful techniques in data analysis, and functions for data visualization and other graphics are an essential part of R:

```
> plot(Date, Declination)
```



Here the user typed another expression, `plot(Date, Declination)`; in this case producing a scatter plot as a side effect, but no printed output. The graphics during an interactive session typically appear in one or more separate windows created by the GUI, in this example a window using the native `quartz()` graphics device for Mac OS X. Graphic output can also be produced in a form suitable for inclusion in a document, such as output in a general file format (PDF or postscript, for example). Computations for graphics are discussed in more detail in Chapter 7.

The sequence of expression and evaluation shown in the examples is essentially all there is to an interactive session. The user supplies expressions and the system evaluates them, one after another. Expressions that produce simple summaries or plots are usually done to see something, either graphics or printed output. Aside from such immediate gratification, most expressions are there in order to assign objects, which can then be used in later computations:

```
> fitK <- gam(Kyphosis ~ s(Age, 4) + Number, family = binomial)
```

Evaluating this expression calls the function `gam()` and assigns the value of the call, associating that object with the name `fitK`. For the rest of the

session, unless some other assignment to this name is carried out, `fitK` can be used in any expression to refer to that object; for example, `coef(fitK)` would call a function to extract some coefficients from `fitK` (which is in this example a fitted model).

Assignments are a powerful and interesting part of the language. The basic idea is all we need for now, and is in any case the key concept: Assignment associates an object with a name. The term "associates" has a specific meaning here. Whenever any expression is evaluated, the context of the evaluation includes a local *environment*, and it is into this environment that the object is assigned, under the corresponding name. The object and name are associated in the environment, by the assignment operation. From then on, the name can be used as a *reference* to the object in the environment. When the assignment takes place at the "top level" (in an input expression in the session), the environment involved is the *global* environment. The global environment is part of the current session, and all objects assigned there remain available for further computations in the session.

Environments are an important part of programming with R. They are also tricky to deal with, because they behave differently from other objects. Discussion of environments continues in Section 2.4, page 24.

A session ends when the user quits from R, either by evaluating the expression `q()` or by some other mechanism provided by the user interface. Before ending the session, the system offers the user a chance to save all the objects in the global environment at the end of the session:

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

If the user answers yes, then when a new session is started in the same working directory, the global environment will be restored. Technically, the environment is restored, not the session. Some actions you took in the session, such as attaching packages or using `options()`, may not be restored, if they don't correspond to objects in the global environment.

Unfortunately, your session may end involuntarily: the evaluator may be forced to terminate the session or some outside event may kill the process. R tries to save the workspace even when fatal errors occur in low-level C or Fortran computations, and such disasters should be rare in the core R computations and in well-tested packages. But to be truly safe, you should explicitly back up important results to a file if they will be difficult to recreate. See documentation for functions `save()` and `dump()` for suitable techniques.

Customizing the R session

As you become a more involved user of R, you may want to customize your interaction with it to suit your personal preferences or the goals motivating your applications. The nature of the system lends itself to a great variety of options from the most general to trivial details.

At the most general is the choice of user interface. So far, we have assumed you will start R as you would start other applications on your computer, say by clicking on the R icon.

A second approach, available on any system providing both R and a command shell, is to invoke R as a shell command. In its early history, S in all its forms was typically started as a program from an interactive shell. Before multi-window user interfaces, the shell would be running on an interactive terminal of some sort, or even on the machine's main console. Nowadays, shells or terminal applications run in their own windows, either supported directly by the platform or indirectly through a client window system, such as those based on X11. Invoking R from a shell allows some flexibility that may not be provided directly by the application (such as running with a C-level debugger). Online documentation from a shell command is printed text by default, which is not as convenient as a browser interface. To initiate a browser interface to the help facility, see the documentation for `help.start()`.

A third approach, somewhat in between the first two, is to use a GUI based on another application or language, potentially one that runs on multiple platforms. The most actively supported example of this approach is ESS, a general set of interface tools in the emacs editor. ESS stands for Emacs Speaks Statistics, and the project supports other statistical systems as well as R; see `ess.r-project.org`. For those who love emacs as a general computational environment, ESS provides a variety of GUI-like features, plus a user-interface programmability characteristic of emacs. The use of a GUI based on a platform-independent user interface has advantages for those who need to work regularly on more than one operating system.

Finally, an R session can be run in a non-interactive form, usually invoked in a batch mode from a command shell, with its input taken from a file or other source. R can also be invoked from within another application, as part of an inter-system interface.

In all these situations, the logic of the R session remains essentially the same as shown earlier (the major exception being a few computations in R that behave differently in a non-interactive session).

Encoding of text

A major advance in R's world view came with the adoption of multiple *locales*, using information available to the R session that defines the user's preferred encoding of text and other options related to the human language and geographic location. R follows some evolving standards in this area. Many of those standards apply to C software, and therefore they fit fairly smoothly into R.

Normally, default locales will have been set when R was installed that reflect local language and other conventions in your area. See Section 8.1, page 293, and `?locales` for some concepts and techniques related to locales. The specifications use standard but somewhat unintuitive terminology; unless you have a particular need to alter behavior for parsing text, sorting character data, or other specialized computations, caution suggests sticking with the default behavior.

Options during evaluation

R offers mechanisms to control aspects of evaluation in the session. The function `options()` is used to share general-purpose values among functions. Typical options include the width of printed output, the prompt string shown by the parser, and the default device for graphics. The `options()` mechanism maintains a named list of values that persist through the session; functions use those values, by extracting the relevant option via `getOption()`:

```
> getOption("digits")
[1] 7
```

In this case, the value is meant to be used to control the number of digits in printing numerical data. A user, or in fact any function, can change this value, by using the same name as an argument to `options()`:

```
> 1.234567890
[1] 1.234568
> options(digits = 4)
> 1.234567890
[1] 1.235
```

For the standard options, see `?options`; however, a call to `options()` can be used by any computation to set values that are then used by any other computation. Any argument name is legal and will cause the corresponding option to be communicated among functions.

Options can be set from the beginning of the session; see `?Startup`. However, saving a workspace image does not cause the options in effect to be saved and restored. Although the `options()` mechanism does use an R object, `.Options`, the internal C code implementing `options()` takes the object from the base package, not from the usual way of finding objects. The code also enforces some constraints on what's legal for particular options; for example, `"digits"` is interpreted as a single integer, which is not allowed to be too small or too large, according to values compiled into R.

The use of `options()` is convenient and even necessary for the evaluator to behave intelligently and to allow user customization of a session. Writing functions that depend on options, however, reduces our ability to understand these functions' behavior, because they now depend on external, changeable values. The behavior of code that depends on an option may be altered by any other function called at any earlier time during the session, if the other function calls `options()`. Most R programming should be *functional programming*, in the sense that each function call performs a well-defined computation depending only on the arguments to that call. The `options()` mechanism, and other dependencies on external data that can change during the session, compromise functional programming. It may be worth the danger, but think carefully about it. See page 47 for more on the programming implications, and for an example of the dangers.

2.3 The Language

This section and the next describe the interactive language as you need to use it during a session. But as noted on page 13, there is no interactive language, only the one language used for interaction and for programming. To use R interactively, you basically need to understand two things: functions and objects. That same duality, functions and objects, runs through everything in R from an interactive session to designing large-scale software. For interaction, the key concepts are function calls and assignments of objects, dealt with in this section and in section 2.4 respectively. The language also has facilities for iteration and testing (page 22), but you can often avoid interactive use of these, largely because R function calls operate on, and return, whole objects.

Function Calls

As noted in Section 2.2, the essential computation in R is the evaluation of a call to a function. Function calls in their ordinary form consist of

the function's name followed by a parenthesized argument list; that is, a sequence of arguments separated by commas.

```
plot(Date, Declination)
glm(Survived ~ .)
```

Arguments in function calls can be any expression. Each function has a set of formal arguments, to which the actual arguments in the call are matched. As far as the language itself is concerned, a call can supply any subset of the complete argument list. For this purpose, argument expressions can optionally be named, to associate them with a particular argument of the function:

```
jitter(y, amount = .1 * rse)
```

The second argument in the call above is explicitly matched to the formal argument named `amount`. To find the argument names and other information about the function, request the online documentation. A user interface to R or a Web browser gives the most convenient access to documentation, with documentation listed by package and within package by topic, including individual functions by name. Documentation can also be requested in the language, for example:

```
> ?jitter
```

This will produce some display of documentation for the topic `"jitter"`, including in the case of a function an outline of the calling sequence and a discussion of individual arguments. If there is no documentation, or you don't quite believe it, you can find the formal argument names from the function object itself:

```
> formalArgs(jitter)
[1] "x"      "factor" "amount"
```

Behind this, and behind most techniques involving functions, is the simple fact that `jitter` and all functions are objects in R. The function name is a reference to the corresponding object. So to see what a function does, just type its name with no argument list following.

```
> jitter
function (x, factor = 1, amount = NULL)
{
    if (length(x) == 0)
        return(x)
    if (!is.numeric(x))
        stop("'x' must be numeric")
            etc.
```

The printed version is another R expression, meaning that you can input such an expression to define a function. At which point, you are programming in R. See Chapter 3. The first section of that chapter should get you started.

In principle, the function preceding the parenthesized arguments can be specified by any expression that returns a function object, but in practice functions are nearly always specified by name.

Operators

Function calls can also appear as operator expressions in the usual scientific notation.

```
y - mean(y)
weight > 0
x < 100 | is.na(date)
```

The usual operators are defined for arithmetic, comparisons, and logical operations (see Chapter 6). But operators in R are not built-in; in fact, they are just special syntax for certain function calls. The first line in the example above computes the same result as:

```
`-`(y, mean(y))
```

The notation `` `-` `` is an example of what are called *backtick quotes* in R. These quotes make the evaluator treat an arbitrary string of characters as if it was a name in the language. The evaluator responds to the names `"y"` or `"mean"` by looking for an object of that name in the current environment. Similarly `` `-` `` causes the evaluator to look for an object named `"-"`. Whenever we refer to operators in the book we use backtick quotes to emphasize that this is the name of a function object, not treated as intrinsically different from the name `mean`.

Functions to extract components or slots from objects are also provided in operator form:

```
mars$Date
classDef@package
```

And the expressions for extracting subsets or elements from objects are also actually just specialized function calls. The expression

```
y[i]
```

is recognized in the language and evaluated as a call to the function `` `[` ``, which extracts a subset of the object in its first argument, with the subset defined by the remaining arguments. The expression `y[i]` is equivalent to:

```
`[`(y, i)
```

You could enter the second form perfectly legally. Similarly, the function `` `[[` `` extracts a single element from an object, and is normally presented as an operator expression:

```
mars[["Date"]]
```

You will encounter a few other operators in the language. Frequently useful for elementary data manipulation is the `` `:` `` operator, which produces a sequence of integers between its two arguments:

```
1:length(x)
```

Other operators include `` `~` ``, used in specifying models, `` `%%` `` for modulus, `` `%*%` `` for matrix multiplication, and a number of others.

New operators can be created and recognized as infix operators by the parser. The last two operators mentioned above are examples of the general convention in the language that interprets

```
%text%
```

as the name of an operator, for any text string. If it suits the style of computation, you can define any function of two arguments and give it, say, the name `` `%d%` ``. Then an expression such as

```
x %d% y
```

will be evaluated as the call:

```
`%d%`(x, y)
```

Iteration: A quick introduction

The language used by R has the iteration and conditional expressions typical of a C-style language, but for the most part you can avoid typing all but the simplest versions interactively. The following is a brief guide to using and avoiding iterative expressions.

The workhorse of iteration is the `for` loop. It has the form:

```
for( var in seq) expr
```

where *var* is a name and *seq* is a vector of values. The loop assigns each element of *seq* to *var* in sequence and then evaluates the arbitrary expression *expr* each time. When you use the loop interactively, you need to either show something each time (printed or graphics) or else assign the result somewhere; otherwise, you won't get any benefit from the computation. For example, the function `plot()` has several "types" of `x-y` plots (points, lines, both, etc.). To repeat a plot with different types, one can use a `for()` loop over the codes for the types:

```
> par(ask=TRUE)
> for(what in c("p","l","b")) plot(Date, Declination, type = what)
```

The call to `par()` caused the graphics to pause between plots, so we get to see each plot, rather then having the first two flash by. The variables `Date` and `Declination` come from some data on the planet Mars, in a data frame object, `mars` (see Section 6.5, page 176). If we wanted to see the class of each of the 17 variables in that data frame, another `for()` loop would do it:

```
for(j in names(mars)) print(class(mars[,j]))
```

But this will just print 17 lines of output, which we'll need to relate to the variable names. Not much use.

Here's where an alternative to iteration is usually better. The workhorse of these is the function `sapply()`. It applies a function to each element of the object it gets as its first argument, so:

```
> sapply(mars,class)
           Year               X          Year.1           Month
      "integer"       "logical"       "integer"       "integer"
            Day       Day..adj.            Hour             Min
  etc.
```

The function tries to simplify the result, and is intelligent enough to include the names as an attribute. See `?sapply` for more details, and the "See Also" section of that documentation for other similar functions.

The language has other iteration operators (`while()` and `repeat`), and the usual conditional operators (`if ... else`). These are all useful in programming and discussed in Chapter 3. By the time you need to use them in a non-trivial way interactively, in fact, you should consider turning your computation into a function, so Chapter 3 is indeed the place to look; see Section 3.4, page 58, in particular, for more detail about the language.

2.4 Objects and Names

A motto in discussion of the S language has for many years been: everything is an object. You will have a potentially very large number of objects available in your R session, including functions, datasets, and many other classes of objects. In ordinary computations you will create new objects or modify existing ones.

As in any computing language, the ability to construct and modify objects relies on a way to refer to the objects. In R, the fundamental reference to an object is a *name*. This is an essential concept for programming with R that arises throughout the book and in nearly any serious programming project.

The basic concept is once again the key thing to keep in mind: references to objects are a way for different computations in the language to refer to the same object; in particular, to make changes to that object. In the S language, references to ordinary objects are only through names. And not just names in an abstract, global sense. An object reference must be a name in a particular R environment. Typically, the reference is established initially either by an assignment or as an argument in a function call.

Assignment is the obvious case, as in the example on page 15:

```
> fitK <- gam(Kyphosis ~ s(Age, 4) + Number, family = binomial)
```

Assignment creates a reference, the name `"fitK"`, to some object. That reference is in some environment. For now, just think of environments as tables that R maintains, in which objects can be assigned names. When an assignment takes place in the top-level of the R session, the current environment is what's called the *global* environment. That environment is maintained throughout the current session, and optionally can be saved and restored between sessions.

Assignments appear inside function definitions as well. These assignments take place during a call to the function. They do not use the global environment, fortunately. If they did, every assignment to the name `"x"` would overwrite the same reference. Instead, assignments during function calls use an environment specially created for that call. So another reason that functions are so central to programming with R is that they protect users from accidentally overwriting objects in the middle of a computation.

The objects available during an interactive R session depend on what packages are attached; technically, they depend on the nested environments through which the evaluator searches, when given a name, to find a corresponding object. See Section 5.3, page 121, for the details of the search.

2.5 Functions and Packages

In addition to the software that comes with any copy of R, there are many thousands of functions available to be used in an R session, along with a correspondingly large amount of other related software. Nearly all of the important R software comes in the form of packages that make the software easily available and usable. This section discusses the implications of using different packages in your R session. For much more detail, see Chapter 4, but that is written more from the view of writing or extending a package. You will get there, I hope, as your own programming efforts take shape. The topic here, though, is how best to use other people's efforts that have been incorporated in packages.

The process leading from needing some computational tool to having it available in your R session has three stages: *finding* the software, typically in a package; *installing* the package; and *attaching* the package to the session.

The last step is the one you will do most often, so let's begin by assuming that you know which package you need and that the required package has been installed with your local copy of R. See Section 2.5, page 26, for finding and installing the relevant package.

You can tell whether the package is attached by looking for it in the printed result of `search()`; alternatively, you can look for a particular object with the function `find()`, which returns the names of all the attached packages that contain the object. Suppose we want to call the function `dotplot()`, for example.

```
> find("dotplot")
character(0)
```

No attached package has an object of this name. If we happen to know that the function is in the package named `lattice`, we can make that package available for the current session. A call to the function `library()` requests this:

```
library(lattice)
```

The function is `library()` rather than `package()` only because the original S software called them libraries. Notice also that the package name was given without quotes. The `library()` function, and a similar function `require()`, do some nonstandard evaluation that takes unquoted names. That's another historical quirk that saves users from typing a couple of quote characters.

If a package of the name `"lattice"` has been installed for this version of R, the call will attach the package to the session, making its functions and other objects available:

```
> library(lattice)
> find("dotplot")
[1] "package:lattice"
```

By "available", we mean that the evaluator will find an object belonging to the package when an expression uses the corresponding name. If the user types `dotplot(Declination)` now, the evaluator will normally find the appropriate function. To see why the quibbling "normally" was added, we need to say more precisely what happens to find a function object.

The evaluator looks first in the global environment for a function of this name, then in each of the attached packages, in the order shown by `search()`. The evaluator will generally stop searching when it finds an object of the desired name, `dotplot`, `Declination`, or whatever. If two attached packages have functions of the same name, one of them will "mask" the object in the other (the evaluator will warn of such conflicts, usually, when a package is attached with conflicting names). In this case, the result returned by `find()` would show two or more packages.

For example, the function `gam()` exists in two packages, `gam` and `mgcv`. If both were attached:

```
> find("gam")
[1] "package:gam"  "package:mgcv"
```

A simple call to `gam()` will get the version in package `gam`; the version in package `mgcv` is now masked.

R has some mechanisms designed to get around such conflicts, at least as far as possible. The language has an operator, `` `::` ``, to specify that an object should come from a particular package. So `mgcv::gam` and `gam::gam` refer unambiguously to the versions in the two packages. The masked version of `gam()` could be called by:

```
> fitK <- mgcv::gam(Kyphosis ~ s(Age, 4) + etc.
```

Clearly one doesn't want to type such expressions very often, and they only help if one is aware of the ambiguity. For the details and for other approaches, particularly when you're programming your own packages, see Section 5.3, page 121.

Finding and installing packages

Finding the right software is usually the hardest part. There are thousands of packages and smaller collections of R software in the world. Section 2.7, page 31, discusses ways to search for information; as a start, CRAN, the

central repository for R software, has a large collection of packages itself, plus further links to other sources for R software. Extended browsing is recommended, to develop a general feel for what's available. CRAN supports searching with the Google search engine, as do some of the other major collections.

Use the search engine on the Web site to look for relevant terms. This may take some iteration, particularly if you don't have a good guess for the actual name of the function. Browse through the search output, looking for a relevant entry, and figure out the name of the package that contains the relevant function or other software.

Finding something which is not in these collections may take more ingenuity. General Web search techniques often help: combine the term `"R"` with whatever words describe your needs in a search query. The e-mail lists associated with R will usually show up in such a search, but you can also browse or search explicitly in the archives of the lists. Start from the R home page, `r-project.org`, and follow the link for `"Mailing Lists"`.

On page 15, we showed a computation using the function `gam()`, which fits a generalized additive model to data. This function is not part of the basic R software. Before being able to do this computation, we need to find and install some software. The search engine at the CRAN site will help out, if given either the function name `"gam"` or the term `"generalized additive models"`. The search engine on the site tends to give either many hits or no relevant hits; in this case, it turns out there are many hits and in fact two packages with a `gam()` function. As an example, suppose we decide to install the `gam` package.

There are two choices at this point, in order to get and install the package(s) in question: a binary or a source copy of the package. Usually, installing from binary is the easy approach, assuming a binary version is available from the repository. Binary versions are currently available from CRAN only for Windows and Mac OS X platforms, and may or may not be available from other sources. Otherwise, or if you prefer to install from source, the procedure is to download a copy of the source archive for the package and apply the `"INSTALL"` command. From an R session, the function `install.packages()` can do part or all of the process, again depending on the package, the repository, and your particular platform. The R GUI may also have a menu-driven equivalent for these procedures: Look for an item in the tool bar about installing packages.

First, here is the function `install.packages()`, as applied on a Mac OS X platform. To obtain the `gam` package, for example:

```
install.packages("gam")
```

The function will then invoke software to access a CRAN site, download the packages requested, and attempt to install them on the same R system you are currently using. The actual download is an archive file whose name concatenates the name of the package and its current version; in our example, `"gam_0.98.tgz"`.

Installing from inside a session has the advantage of implicitly specifying some of the information that you might otherwise need to provide, such as the version of R and the platform. Optional arguments control where to put the installed packages, whether to use source or binary and other details.

As another alternative, you can obtain the download file from a Web browser, and run the installation process from the command shell. If you aren't already at the CRAN Web site, select that item in the navigation frame, choose a mirror site near you, and go there.

Select `"Packages"` from the CRAN Web page, and scroll or search in the list of packages to reach a package you want (it's a very long list, so searching for the exact name of the package may be required). Selecting the relevant package takes you to a page with a brief description of the package. For the package `gam` at the time this is written:

gam: Generalized Additive Models

Functions for fitting and working with generalized additive models, as described in chapter 7 of "Statistical Models in S" (Chambers and Hastie (eds), 1991), and "Generalized Additive Models" (Hastie and Tibshirani, 1990).

Version: 0.98
Depends: R (>= 2.0), stats, splines
Suggests: akima
Date: 2006-07-11
Author: Trevor Hastie
Maintainer: Trevor Hastie
License: GPL2.0

Downloads:

Package source: gam_0.98.tar.gz
MacOS X binary: gam_0.98.tgz
Windows binary: gam_0.98.zip
Index of contents: gam.INDEX
Reference manual: gam.pdf

At this stage, you can access the documentation or download one of the proffered versions of the package. Or, after studying the information, you could revert to the previous approach and use `install.packages()`. If you do work from one of the source or binary archives, you need to apply the shell-style command to install the package. Having downloaded the source archive for package `gam`, the command would be:

```
R CMD INSTALL gam_0.98.tar.gz
```

The INSTALL utility is used to install packages that we write ourselves as well, so detailed discussion appears in Chapter 4.

The package for this book

In order to follow the examples and suggested computations in the book, you should install the SoDA package. It is available from CRAN by any of the mechanisms shown above. In addition to the many references to this package in the book itself, it will be a likely source for new ideas, enhancements, and corrections related to the book.

2.6 Getting R

R is an open-source system, in particular a system licensed under the *GNU Public license*. That license requires that the source code for the system be freely available. The current source implementing R can be obtained over the Web. This open definition of the system is a key support when we are concerned with trustworthy software, as is the case with all similar open-source systems.

Relatively simple use of R, and first steps in programming with R, on the other hand, don't require all the resources that would be needed to create your local version of the system starting from the source. You may already have a version of R on your computer or network. If not, or if you want a more recent version, binary copies of R can be obtained for the commonly used platforms, from the same repository. It's easier to start with binary, although as your own programming becomes more advanced you may need more of the source-related resources anyway.

The starting point for obtaining the software is the central R Web site, `r-project.org`. You can go there to get the essential information about R. Treat that as the up-to-date authority, not only for the software itself but also for detailed information about R (more on that on page 31).

The main Web site points you to a variety of pages and other sites for various purposes. To obtain R, one goes to the CRAN repository, and from there to either `"R Binaries"` or `"R Sources"`. Downloading software may involve large transfers over the Web, so you are encouraged to spread the load. In particular, you should select from a list of mirror sites, preferably picking one geographically near your own location. When we talk about the

CRAN site from now on, we mean whichever one of the mirror sites you have chosen.

R is actively maintained for three platforms: Windows, Mac OS X, and Linux. For these platforms, current versions of the system can be obtained from CRAN in a form that can be directly installed, usually by a standard installation process for that platform. For Windows, one obtains an executable setup program (a ".exe" file); for Mac OS X, a disk image (a ".dmg" file) containing the installer for the application. The Linux situation is a little less straightforward, because the different flavors of Linux differ in details when installing R. The Linux branch of "R Binaries" branches again according to the flavors of Linux supported, and sometimes again within these branches according to the version of this flavor. The strategy is to keep drilling down through the directories, selecting at each stage the directory that corresponds to your setup, until you finally arrive at a directory that contains appropriate files (usually ".rpm" files) for the supported versions of R.

Note that for at least one flavor of Linux (Debian), R has been made a part of the platform. You can obtain R directly from the Debian Web site. Look for Debian packages named "r-base", and other names starting with "r-". If you're adept at loading packages into Debian, working from this direction may be the simplest approach. However, if the version of Debian is older than the latest stable version of R, you may miss out on some later improvements and bug fixes unless you get R from CRAN.

For any platform, you will eventually download a file (".exe", "dmg", ".rpm", or other), and then install that file according to the suitable ritual for this platform. Installation may require you to have some administration privileges on the machine, as would be true for most software installations. (If installing software at all is a new experience for you, it may be time to seek out a more experienced friend.) Depending on the platform, you may have a choice of versions of R, but it's unlikely you want anything other than the most recent stable version, the one with the highest version number. The platform's operating system will also have versions, and you generally need to download a file asserted to work with the version of the operating system you are running. (There may not be any such file if you have an old version of the operating system, or else you may have to settle for a comparably ancient version of R.) And just to add further choices, on some platforms you need to choose from different hardware (for example, 32-bit versus 64-bit architecture). If you don't know which choice applies, that may be another indication that you should seek expert advice.

Once the binary distribution has been downloaded and installed, you should have direct access to R in the appropriate mechanism for your plat-

form.

Installing from source

Should you? For most users of R, not if they can avoid it, because they will likely learn more about programming than they need to or want to. For readers of this book, on the other hand, many of these details will be relevant when you start to seriously create or modify software. Getting the source, even if you choose not to install it, may help you to study and understand key computations.

The instructions for getting and for installing R from source are contained in the online manual, *R Installation and Administration*, available from the `Documentation` link at the `r-project.org` Web site.

2.7 Online Information About R

Information for users is in various ways both a strength and a problem with open-source, cooperative enterprises like R. At the bottom, there is always the source, the software itself. By definition, no software that is not open to study of all the source code can be as available for deep study. In this sense, only open-source software can hope to fully satisfy the *Prime Directive* by offering unlimited examination of what is actually being computed.

But on a more mundane level, some open-source systems have a reputation for favoring technical discussions aimed at the insider over user-oriented documentation. Fortunately, as the R community has grown, an increasing effort has gone into producing and organizing information. Users who have puzzled out answers to practical questions have increasingly fed back the results into publicly available information sources.

Most of the important information sources can be tracked down starting at the main R Web page, `r-project.org`. Go there for the latest pointers. Here is a list of some of the key resources, followed by some comments about them.

Manuals: The R distribution comes with a set of manuals, also available at the Web site. There are currently six manuals: *An Introduction to R*, *Writing R Extensions*, *R Data Import/Export*, *The R Language Definition*, *R Installation and Administration*, and *R Internals*. Each is available in several formats, notably as Web-browsable HTML documents.

Help files: R itself comes with files that document all the functions and other objects intended for public use, as well as documentation files on other topics (for example, `?Startup`, discussing how an R session starts).

All contributed packages should likewise come with files documenting their publicly usable functions. The quality control tools in R largely enforce this for packages on CRAN.

Help files form the database used to respond to the help requests from an R session, either in response to the `Help` menu item or through the `` `?` `` operator or `help()` function typed by the user.

The direct requests in these forms only access terms explicitly labeling the help files; typically, the names of the functions and a few other general terms for documentation (these are called *aliases* in discussions of R documentation). For example, to get help on a function in this way, you must know the name of the function exactly. See the next item for alternatives.

Searching: R has a search mechanism for its help files that generalizes the terms available beyond the aliases somewhat and introduces some additional searching flexibility. See `?help.search` for details.

The `r-project.org` site has a pointer to a general search of the files on the central site, currently using the Google search engine. This produces much more general searches. Documentation files are typically displayed in their raw, LaTeX-like form, but once you learn a bit about this, you can usually figure out which topic in which package you need to look at.

And, beyond the official site itself, you can always apply your favorite Web search to files generally. Using `"R"` as a term in the search pattern will usually generate appropriate entries, but it may be difficult to avoid plenty of inappropriate ones as well.

The Wiki: Another potentially useful source of information about R is the site `wiki.r-project.org`, where users can contribute documentation. As with other open Wiki sites, this comes with no guarantee of accuracy and is only as good as the contributions the community provides. But it has the key advantage of openness, meaning that in some "statistical" sense it reflects what R users understand, or at least that subset of the users sufficiently vocal and opinionated to submit to the Wiki.

The strength of this information source is that it may include material that users find relevant but that developers ignore for whatever reason (too trivial, something users would never do, etc.). Some Wiki sites have sufficient support from their user community that they can function as the main information source on their topic. As of this writing, the R Wiki has not reached that stage, so it should be used as a supplement to other information sources, and not the primary source, but it's a valuable resource nevertheless.

The mailing lists: There are a number of e-mail lists associated officially with the R project (officially in the sense of having a pointer from the R Web page, `r-project.org`, and being monitored by members of R core). The two most frequently relevant lists for programming with R are `r-help`, which deals with general user questions, and `r-devel`, which deals generally with more "advanced" questions, including future directions for R and programming issues.

As well as a way to ask specific questions, the mailing lists are valuable archives for past discussions. See the various search mechanisms pointed to from the mailing list Web page, itself accessible as the `Mailing lists` pointer on the `r-project.org` site. As usual with technical mailing lists, you may need patience to wade through some long tirades and you should also be careful not to believe all the assertions made by contributors, but often the lists will provide a variety of views and possible approaches.

Journals: The electronic journal *R News* is the newsletter of the R Foundation, and a good source for specific tutorial help on topics related to R, among other R-related information. See the `Newsletter` pointer on the `cran.r-project.org` Web site.

The *Journal of Statistical Software* is also an electronic journal; its coverage is more general as its name suggests, but many of the articles are relevant to programming with R. See the Web site `jstatsoft.org`.

A number of print journals also have occasional articles of direct or indirect relevance, for example, *Journal of Computational and Graphical Statistics* and *Computational Statistics and Data Analysis*.

2.8 What's Hard About Using R?

This chapter has outlined the computations involved in using R. An R session consists of expressions provided by the user, typically typed into an R console window. The system evaluates these expressions, usually either showing the user results (printed or graphic output) or assigning the result as an object. Most expressions take the form of calls to functions, of which there are many thousands available, most of them in R packages available on the Web.

This style of computing combines features found in various other languages and systems, including command shells and programming languages. The combination of a functional style with user-level interaction—expecting the user to supply functional expressions interactively—is less common. Beginning users react in many ways, influenced by their previous experience, their expectations, and the tasks they need to carry out. Most readers of this book have selected themselves for more than a first encounter with the software, and so will mostly not have had an extremely negative reaction. Examining some of the complaints may be useful, however, to understand how the software we create might respond (and the extent to which we can respond). Our mission of supporting effective exploration of data obliges us to try.

The computational style of an R session is extremely general, and other aspects of the system reinforce that generality, as illustrated by many of the topics in this book (the general treatment of objects and the facilities for interacting with other systems, for example). In response to this generality, thousands of functions have been written for many techniques. This diversity has been cited as a strength of the system, as indeed it is. But for some users exactly this computational style and diversity present barriers to using the system.

Requiring the user to compose expressions is very different from the mode of interaction users have with typical applications in current computing. Applications such as searching the Web, viewing documents, or playing audio and video files all present interfaces emphasizing *selection-and-response* rather than composing by the user. The user *selects* each step in the computation, usually from a menu, and then *responds* to the options presented by the software as a result. When the user does have to compose (that is, to type) it is typically to fill in specific information such as a Web site, file or optional feature desired. The eventual action taken, which might be operationally equivalent to evaluating an expression in R, is effectively defined by the user's interactive path through menus, forms and other specialized tools in the interface. Based on the principles espoused

in this book, particularly the need for trustworthy software, we might object to a selection-and-response approach to serious analysis, because the ability to justify or reproduce the analysis is much reduced. However, most non-technical computing is done by selection and response.

Even for more technical applications, such as producing documents or using a database system, the user's input tends to be relatively free form. Modern document-generating systems typically format text according to selected styles chosen by the user, rather than requiring the user to express controls explicitly. These differences are accentuated when the expressions required of the R user take the form of a functional, algebraic language rather than free-form input.

This mismatch between requirements for using R and the user's experience with other systems contributes to some common complaints. How does one start, with only a general feeling of the statistical goals or the "results" wanted? The system itself seems quite unhelpful at this stage. Failures are likely, and the response to them also seems unhelpful (being told of a syntax error or some detailed error in a specific function doesn't suggest what to do next). Worse yet, computations that don't fail may not produce any directly useful results, and how can one decide whether this was the "right" computation?

Such disjunctions between user expectations and the way R works become more likely as the use of R spreads. From the most general view, there is no "solution". Computing is being viewed differently by two groups of people, prospective users on one hand, and the people who created the S language, R and the statistical software extending R on the other hand.

The S language was designed by research statisticians, initially to be used primarily by themselves and their colleagues for statistical research and data analysis. (See the Appendix, page 475.) A language suited for this group to communicate their ideas (that is, to "program") is certain to be pitched at a level of abstraction and generality that omits much detail necessary for users with less mathematical backgrounds. The increased use of R and the growth in software written using it bring it to the notice of such potential users far more than was the case in the early history of S.

In addition to questions of expressing the analysis, simply choosing an analysis is often part of the difficulty. Statistical data analysis is far from a routine exercise, and software still does not encapsulate all the expertise needed to choose an appropriate analysis. Creating such expert software has been a recurring goal, pursued most actively perhaps in the 1980s, but it must be said that the goal remains far off.

So to a considerable extent the response to such user difficulties must

include the admission that the software implemented in R is not directly suited to all possible users. That said, information resources such as those described earlier in this chapter are making much progress in easing the user's path. And, those who have come far enough into the R world to be reading this book can make substantial contributions to bringing good data analysis tools to such users.

1. Specialized selection-and-response interfaces can be designed when the data analysis techniques can be captured with the limited input provided by menus and forms.

2. Interfaces to R from a system already supporting the application is another way to provide a limited access expressed in a form familiar to the user of that system. We don't describe such interfaces explicitly in this book, but see Chapter 12 for some related discussion.

3. Both educational efforts and better software tools can make the use of R seem more friendly. More assistance is available than users may realize; see for example the suggestions in Section 3.5. And there is room for improvement: providing more information in a readable format for the beginning user would be a valuable contribution.

4. Last but far from least in potential value, those who have reached a certain level of skill in applying data analysis to particular application areas can ease their colleagues' task by documentation and by providing specialized software, usually in the form of an R package. Reading a description in familiar terminology and organized in a natural structure for the application greatly eases the first steps. A number of such packages exist on CRAN and elsewhere.