S-PLUS 5.x Complements to

Modern Applied Statistics with S-Plus

Second edition

by

W. N. Venables and B. D. Ripley

Springer (1997). ISBN 0-387-98214-0

3 May 1998

These complements have been produced to supplement the second edition of MASS. They will be updated from time to time. The definitive source is http://www.stats.ox.ac.uk/pub/MASS2/.

Selectable links are in this colour.
Selectable URLs are in this colour.

Introduction

These complements are made available on-line to supplement the book for users of S-PLUS 5.x, which is based on version 4 of the S language. It is based on release 3 of the Solaris and Linux versions of 5.0 and previews of version 5.1, but will be updated as other versions become available to us.

The general convention is that material here should be thought of as following the material in the chapter in the book, so that new sections are numbered following the last section of the chapter, and figures and equations here are numbered following on from those in the book.

There are separate Complements documents for S programming and for statistical methods available from http://www.stats.ox.ac.uk/pub/MASS2/.

Contents

Chapter 1

Introduction

1.1 A quick overview of S

The symbol `=` may be used (almost everywhere) for assignment in place of `<-`. However, the exceptions[1] are insidious and can cause hard-to-trace errors, so `<-` is still the preferred choice for S-PLUS.[2]

1.2 Using S-PLUS under UNIX

Getting started

The way in which the file system is used to store objects has changed somewhat with S-PLUS 5.x. Objects are still stored in a `.Data` directory, but that now has `__Meta` and `__Help` subdirectories, and perhaps `__Shelp` and `__Hhelp`.

The recommended procedure to create a new project is now (assuming the command to start S-PLUS 5.x on your system is `Splus5`)

1. Create a separate directory, say `SwS`, for this project, which we suppose is 'Statistics with S-PLUS', and make it your working directory.

   ```
   $ mkdir SwS
   $ cd SwS
   ```

   Copy any data files you need to use with S-PLUS to this directory.

2. Within the project directory run

   ```
   $ Splus5 CHAPTER
   ```

   which will create the subdirectories which S-PLUS uses.

3. Start the system with

   ```
   $ Splus5
   ```

[1] when it might be confused with setting the value of a parameter; see page 6.
[2] In *Getting Started with* S-PLUS *5.0* page 9, and also preferred by us.

4. At this point S commands may be issued (see later). The default prompt is `>` unless the command is incomplete, when it is `+`. To use our main software library issue

```
library(MASS, first=T)
```

5. To quit the S program the command is

```
>  q()
$
```

Working directory and initialization in S-PLUS *5.0*

If the current directory has not been set up as an S-PLUS chapter, the following options are tried in turn

(i) If the user's home directory is itself an S-PLUS chapter, it is used.

(ii) A 'temporary' chapter is created (in the users' home directory) for the current S-PLUS session only. However, this is not removed after the session: it has a name of the form `Schapter2497`.

Setting the environment variable `S_WORK` will override the choice process: if `S_WORK` specifies an S-PLUS chapter it will be used, otherwise the directory it specifies is used if writeable otherwise a 'temporary' chapter is used.

Whether an S-PLUS chapter is present is recognized by the presence of a subdirectory named `.Data`, so a directory used as a project with an earlier version of S-PLUS will be recognized. It is not recommended to use such a directory though, and its objects should be converted as described on page 55.

Note that the behaviour here is quite different from that currently documented.

You can ask for some actions to be performed whenever S-PLUS is started.

(i) If a file `.S.init` is present in the current directory it is evaluated as a set of S commands. This is done early in the sequence, before `.First.Sys` or `.First` are called.

(ii) If a function `.First.local` exists on the search path it is called from `.First.Sys`.

(iii) The last start-up action is to execute any commands specified in the environmental variable `S_FIRST` or this does not exist to call a function `.First` if one is found in the search path (usually in the current working directory).

Again, the current behaviour here is quite different from that currently documented.

Working directory and initialization in S-PLUS *5.1*

These actions have been changed in S-PLUS 5.1, but are still different from those described in Chambers (1998).

If the current directory has not been set up as an S-PLUS chapter, the following options are tried in turn

(i) If `~/MySwork` is an S-PLUS chapter, it is used.

(ii) If `~/MySwork` does not exist, it is created as an S-PLUS chapter and used.

(iii) A 'temporary' chapter is created in the users' home directory for the current S-PLUS session only. However, this is not removed after the session: it has a name of the form `Schapter2497`.

Setting the environment variable `S_WORK` will override the choice process: if `S_WORK` specifies an S-PLUS chapter it will be used, otherwise the directory it specifies is used if writeable otherwise a 'temporary' chapter is used.

Whether an S-PLUS chapter is present is recognized by the presence of a subdirectory named `.Data/__Meta`, so a directory used as a project with an earlier version of S-PLUS will *not* be recognized.

You can ask for some actions to be performed whenever S-PLUS is started.

(i) If a file `.S.init` is present in `SHOME` it is evaluated as a set of S commands.

(ii) If a file `.S.chapters` is present in the `SHOME` directory, the chapters it specifies are attached.

(iii) If a file `.S.chapters` is present in the current directory or, failing that, the user's `~/MySwork` directory, the chapters it specifies are attached. These chapters are specified one per line, either as absolute paths or relative to the system library directory.

(iv) If a file `.S.init` is present in the current directory, or failing that, the user's `~/MySwork` directory, it is evaluated as a set of S commands. This is done early in the sequence, before `.First.Sys` or `.First` are called.

(v) If a function `.First.local` exists on the search path it is called from `.First.Sys`.

(vi) The last start-up action is to execute any commands specified in the environmental variable `S_FIRST` or this does not exist to call a function `.First` if one is found in the search path (usually in the current working directory).

Getting help with functions and features — 5.0

The help available in S-PLUS 5.x is a mixture of old-style and new-style. Most of the help files are old-style and appear exactly as they did in S-PLUS 3.x. There is a format for later files that looks like

```
> ?getClass

Title:
        Function getClass

Usage:
        getClass(Class, complete, where)

Arguments:
  Class:  argument, no default.
  complete:  argument, 'default = missing(where).'
  where:  argument, no default.

Description:
        If 'complete' is 'TRUE', get the current definition of
        the class from the session, or an empty definition
        if 'Class' is not defined.


        If 'complete' is 'FALSE', or 'where' is provided, get the
        object from the meta-data defining
        the class (it's an error in this case if there is none).
        See also:
        'findClass', 'existsClass'


[1] "getClass"
```

If you use `?object` or `help(object)` you will get the old-style help if available, with the same interface as before. Offline printing of old-style help works, but nothing is produced from new-style help.

Use `help(object, oldOK=F)` to force the use of new-style documentation (which may well be quite different). In that case using `offline=T` sends the output to a file.

There is no `help.start` system.

Getting help with functions and features – 5.1

S-PLUS 5.1 has a new help system. The help files are stored in SGML files in the `.Data/__Shelp` directory and converted as needed to HTML files in the `.Data/__Hhelp` directory.

Help files can be displayed in one of two ways.

1. Using `help` or `?` will display the file in a modified version of the `lynx` browser in the current terminal. You can follow hyperlinks with the cursor arrow keys. This will find help in any directory on the search path, including the current directory and any packages that have been attached.

2. If Netscape 4.x is installed, running `help.start()` will start a new browser window with a JAVA-based search system for help pages. Note that this help system currently only knows about the system help pages.

   After `help.start`, help pages found in the first option are sent to Netscape rather than `lynx`. To stop this, call `help.off()`.

Chapter 2

The S Language

Versions 5.0 and later of S-PLUS are based on version 4 of the S language, which although broadly compatible with version 3[1] introduces a number of new features. Often the old constructs will work, but there are replacement constructs that are more efficient. The definitive source of information on this version of the S language is Chambers (1998), which does not provide a detailed conversion path for those familiar with the earlier language, our aim in these complements. (Note that Chambers describes the version of the language in use at Lucent and not necessarily that implemented in S-PLUS 5.x.)

2.1 A concise description of S objects

The most obvious new feature is the use of `=` rather than `<-` in Chambers (1998). This can be used wherever it cannot be confused with the `=` between a formal argument and the actual argument. The only common examples are like

```
if(!is.null(x <- object$x))
```

where if we had `x = object$x` the assignment would never take place. However, the S-PLUS documentation recommends that `<-` is still used in preference to `=` (let alone _).

Every object has a class: this makes attributes and modes (page 23) much less important. Thus the example on page 23 changes to

```
> mydata <-  c(2.9, 3.4, 3.4, 3.7, 3.7, 2.8, 2.8, 2.5, 2.4, 2.4)
> class(mydata)
[1] "numeric"
> names(mydata) <- letters[seq(along=mydata)]
> mydata
   a   b   c   d   e   f   g   h   i   j
 2.9 3.4 3.4 3.7 3.7 2.8 2.8 2.5 2.4 2.4
> class(mydata)
[1] "named"
> dim(mydata) <- c(2, 5)
```

[1] the basis of S-PLUS 3.x and 4.x

```
> mydata
     [,1] [,2] [,3] [,4] [,5]
[1,]  2.9  3.4  3.7  2.8  2.4
[2,]  3.4  3.7  2.8  2.5  2.4
> class(mydata)
[1] "matrix"
> dim(mydata) <- NULL
> mydata
 [1] 2.9 3.4 3.4 3.7 3.7 2.8 2.8 2.5 2.4 2.4
> class(mydata)
[1] "numeric"
```

Assigning `dim` no longer sets an attribute: it changes `mydata` into an object of class `"matrix"`.

Coercion

[page 28]
There are now generic conversion functions `is` and `as` as well as the older `is.xxx` and `as.xxx` functions. For more details see page 19 of these complements.

2.2 Arithmetical expressions

Fractional re-cycling (page 31) is now an error. An expression that contains a vector of length zero now consistently returns a vector of length zero. Thus we have

```
> x <- c(10.4, 5.6, 3.1, 6.4, 21.7)
>  x + x[x > 27]            # S-PLUS 3.x and 4.x
[1] NA NA NA NA NA
> x + x[x > 27]            # S-PLUS 5.x
numeric(0)
```

There are some restrictions on the language stemming from the restrictions on agreement between function arguments that we shall discuss in Section 4.4. In particular, `log` is now a function of just one argument; use `log10` or `logb` (for arbitrary base) as appropriate.

The preferred way to set a value to missing is now to use `is.na` on the left-hand side of an assignment, as in

```
> is.na(mydata)[6] <- T
> mydata
 [1] 2.9 3.4 3.4 3.7 3.7  NA 2.8 2.5 2.4 2.4
```

2.6 Character vector operations

Chambers (1998) promises classes `"string"` and `"stringFactor"` and `"stringOrdered"` to supersede `"character"`, `"factor"` and `"ordered"`. However, these classes are not currently supported, and their use is not recommended.

Regular expressions are powerful ways to match character strings familiar to users of such tools as `sed`, `grep`, `awk` and `perl`. They had a limited implementation in the S function `grep` and a more powerful one in function `regexpr` in S-PLUS 4.x, which is also available in S-PLUS 5.x. Function `regexpr` matches one regular expression to a character vector.

```
> regexpr("na$", state.name)
 [1] -1 -1  6 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1  6 -1 -1 -1  8 -1
[20] -1 -1 -1 -1 -1 -1  6 -1 -1 -1 -1 -1 -1 13 -1 -1 -1 -1 -1
[39] -1 13 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
attr(, "match.length"):
 [1] -1 -1  2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1  2 -1 -1 -1  2 -1
[20] -1 -1 -1 -1 -1 -1  2 -1 -1 -1 -1 -1 -1  2 -1 -1 -1 -1 -1
[39] -1  2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
>  state.name[regexpr("na$", state.name) > 0]
[1] "Arizona"        "Indiana"        "Louisiana"
[4] "Montana"        "North Carolina" "South Carolina"
```

The functions `regMatch` and `regMatchPos` have a very similar role, but encode the answer somewhat differently.

```
> regMatch(state.name, "na$")
 [1] F F T F F F F F F F F F F T F F F T F F F F F F F T F F
[29] F F F F T F F F F F F T F F F F F F F F F F
> regMatchPos(state.name, "na$")
integer matrix: 50 rows, 2 columns.
      [,1] [,2]
 [1,]   NA   NA
 [2,]   NA   NA
 [3,]    6    7
 [4,]   NA   NA
    ....
```

Both `regMatch` and `regMatchPos` can match multiple regular expressions, recycling arguments as needed.

Regular expressions can also be used in specifying index vectors in extracting or replacing subsets of vector-like objects, for example

```
> state.name[regularExpression("^.*o$")]     # ends in 'o'
[1] "Colorado"   "Idaho"      "New Mexico" "Ohio"
> state.name[regularExpression("^[^ ]+o$")]  # no blanks
[1] "Colorado" "Idaho"    "Ohio"
```

and as the `first` argument to `substring`, as in

```
> newStates <- state.name[regularExpression("New .*$")]
> substring(newStates, "[^ ]+$")
[1] "Hampshire" "Jersey"    "Mexico"    "York"
```

which extracts the last word of the states whose names start with '`New`'.[2]

There are many dialects of regular expressions: no one seems to specify exactly which is required. On many Unix systems `man -s5 regexp` will give a description of one dialect.

2.7 Finding S objects

The result of a call to `search` is somewhat simpler

```
> search()
[1] "."             "documentation" "data"
[4] "trellis"       "splus"         "stat"
[7] "MASS"          "main"
```

in that only the names of the databases are returned. Some more information is available via the function `searchPaths()` or

```
> search("paths")
[1] ".Data"                         "$SHOME/library/splus"
[3] "$SHOME/library/stat"           "$SHOME/library/data"
[5] "$SHOME/library/documentation" "$SHOME/library/trellis"
[7] "soft/MASS/.Data"               "$SHOME/library/main"
```

Note how libraries are by default attached in the penultimate place.

2.10 Input/Output facilities

S-PLUS 5.x has a general notion of a *connection* to extend that of a file. Many of the types of connections will be familiar to Unix devotees, for in Unix they are just types of files, like pipes and fifos[3]. A (read-only) connection can also be an S character vector. All but the simplest uses of connections will be of interest only to system programmers, who are referred to Chambers (1998, Chapter 10) and especially the on-line help for the details.

Functions such as `scan` and `write` had an argument `"file"` which specified (as a character string) the name of the file to be used; in many cases this can be replaced by a connection. Why would we want to do this? One reason is to keep a file open for further operations. Earlier versions of our function `ppinit` in library `spatial` had

[2] These examples fail on 5.0r3 on Solaris.
[3] also known as named pipes

```
h <- scan(tfile, list(xl = 0, xu = 0, yl = 0, yu = 0, fac = 0),
          n = 5, skip = 2)
pp <- scan(tfile, list(x = 0, y = 0), skip = 3)
```

which opened the file, skipped two lines, read the third, closed the file, opened the file, skipped three lines and read the rest. We can now use

```
tf <- open(tfile)
h <- scan(tf, list(xl = 0, xu = 0, yl = 0, yu = 0, fac = 0),
          n = 5, skip = 2)
pp <- scan(tf, list(x = 0, y = 0))
close(tf)
```

The convention is that if a function finds a connection open it leaves it open, but if it needs to open it, it is also closed.

Connections are explictly opened by `open` and closed by `close`. Their first argument is `connection` and connections can be generated by any of the functions

```
file("path", open, blocking = T)
pipe("shell command", open)
fifo("path", open="", blocking = F)
textConnection("character vector", open, blocking = T)
stdin(); stdout(); stderr()
```

(Specifying `connection` as a character string gives an implicit call to `file`.) These behave as one would expect: in particular a text connection treats each element of a character vector as a line of text (and is read-only). The `open` argument should either be a logical value (by default the connection is *not* opened) or a character string giving the mode[4].

One thing that does *not* behave as one would expect is file positioning: S-PLUS 5.x keeps separate positions for reading and writing, initially at the beginning and end of the file respectively. Function `seek` allows either position to be changed (if allowed), and will also return the current position (so encompasses the '`tell`' of POSIX file calls).

Connections provide a neater way to use temporary files, at least within S-PLUS. The old way was to use `tempfile` (most unfortunately named; page 147)

```
tmpname <- tempfile()
tf <- open(tmpname)
# do something with the file
tf <- close(tmpname); unlink(tmpname)
```

which can be replaced by

```
tf <- file()
# do something with the file
close(tf)  # might not be needed
```

[4] for example, `"r"`, `"ra"` (read/append) or `"*"` (anything allowed by the OS).

There is a potential disadvantage with this approach: if the 'do something' involves operating system commands the file name is not available (nor is the file itself).

There are additional input/output functions that are particularly useful with open connections. Functions `readLines` and `writeLines` read and write a specified number of text lines; `parseSome` reads and parse up to one S expression and `dataGet` and `dataPut` reads or writes one object in `data.dump` format.

The function `showConnections` shows all the currently open connections, except the standard ones unless its argument `all = T`;

```
> showConnections(all=T)
            Class Mode   State      Description
 stdin "terminal" "r"  "Read"   "/dev/tty"
stdout "terminal" "w"  "Write"  "/dev/tty"
stderr "terminal" "w"  "Write"  "/dev/tty"
 Audit "file"     "a"  "Write"  "./.Data/.Audit"
```

Functions `source` *and* `sink`

The function `sink` has been enhanced, but is not as described in Chambers (1998). Sink files can be nested: see the help page for details.

When `source` is used, the file is read in as a single expression and then evaluated, so any errors in the file result in nothing being committed (which can result in large memory usage whilst deferring commitment), and auto-printing will not occur. There is now another possibility, `setReader(file("myfile.q"))`, that starts another S evaluator and so reads the file asynchronously. Unlike `source`, commitments are made immediately and auto-printing does occur, but the input is not echoed by default.

Binary and `raw` data

There is a new class of data, `raw`, that is considered here as very little can be done with it in S except via compiled code that is linked in. It is a stream of bytes, best read by `readRaw` and written by `writeRaw`, or for large files (but not other connections) by `readMapped`. There are functions `rawFromHex`, `rawFromAscii`, `rawToHex`, and `rawToAscii` to convert to and from character vectors.

These functions can also handle other elementary types such as numeric or integer vectors, and so read or write binary files for use with C programs.

Import and export

S-PLUS 5.x can import and export data frames from/to files in a variety of formats through the functions `importData` and `exportData`. The supported formats are those of a number of spreadsheets[5], databases, statistical (SAS, SPSS, Stata, Systat) and matrix-language (GAUSS, MATLAB) systems, as well as ASCII files (although there are many other ways to read and write these).

[5] but not Excel 95 nor Excel 97

HTML output

The function `html.table` is similar to `write.table`, but formats the result as an HTML table. There are variations on the HTML for tables accepted by browsers, and this one seems to be in Netscape style.

2.11 Customizing your S environment

There are some further options that can be useful. By default, S-PLUS 5.x warns of all conflicts when libraries are attached: suppress whese warnings by `options(conflicts.ok=T)`. The indentation of function listings can be changed by option `indentation` which defaults to a tab, but one can use, for example,

```
options(indentation="  ")
```

to indent each level by two spaces.

2.13 BATCH operation

In batch mode the command prompts will not appear in the output file: the help page for `parse` says this can be altered by setting the environment variable `ALWAYS_PROMPT`, but this is not currently the case.

2.15 New language features

Function `el` extracts a single element of a list: for `i` a single positive integer `el(x, i)` is equivalent to `x[[i]]`, although implemented at a lower level. As a result, end users will probably never use `el`, but they may see it in functions.

In addition to `c` there is a new function `concat` which does not name the result and works for a larger set of classes of objects. Thus

```
> c(a=1, b=2:3)
 a b1 b2
 1  2  3
> concat(a=1, b=2:3)
[1] 1 2 3
```

Large objects in S-PLUS 5.x are memory-mapped, that is stored in virtual memory. This is for efficiency, but only 50 such objects can be memory-mapped at one time. The default threshold for memory mapping is 50Kb, but this can be altered by a call to `mmap.control`, which with no argument returns the current threshold[6].

[6] although this is not documented.

S-PLUS 5.x has more elaborate facilities for testing results. The function `identical` asserts that two S objects are identical in every respect (or not).

There has long been a function `on.exit` to set up actions on leaving a function, including after an error. The function `.onError` provides a means to write more sophisticated error handlers: see Chambers (1998, pp. 270–2) for further details.

Chapter 4

Programming in S

4.2 Vectorized calculations and loop avoidance functions

Chambers (1998, pp. 173–5) suggests that the inefficiency gains by replacing `for` loops by the `apply` family have been over-emphasised in the past: our experience with S-PLUS differs from his with S, and this is an area where the two systems have differed. The 'vectorization' or 'whole-object'view of the problem is however a very helpful one in writing clear code.

There is a new member of the `apply` family, `rapply`, which is like `lapply` but applied recursively, that is to all lists contained within the supplied list, and to lists within those lists and so on. It has a `classes` argument to restrict the types of terminal components to which the function is applied, and a `replace=T` argument to allow all other terminal components to be carried over unchanged. (This name could be misleading: think of a copy of the list being made and then each component that the function is applied to is replaced by the new value.) If `replace=F`, the default, terminal components which do not match any component of the `classes` argument are effectively removed from the result rather than carried forward into it.

4.4 Introduction to object orientation

Version 4 of the S language that underlies S-PLUS 5.x has introduced a radically different approach to object orientation, although backwards compatibility is provided for the object-orientation system described in Section 4.5. The system relies very heavily on this backwards compatibility; at present new-style classes are used only at a very low level and in the new time-series software. The definitive reference for the new-style object orientation is Chambers (1998).

Every object in S-PLUS 5.x has a class, and only one class. For example, vectors are of class `numeric`, `integer`, `character`, ... and matrices are of class `matrix`.

All objects in a class must have the same structure. This is not true of many old-style classes; for example objects of class `lm` are lists which may or may not have a component named `x`, and the attributes mechanism is often used to append

varying amounts of information. This *can* be accommodated in the new-style classes by specifying class as composed of class `"list"` or `"structure"`, but the advantages of the new-style classes accrue from a tighter specification.

All method for a new-style generic function must have exactly the same formal arguments. Again, flexibility is possible, via a `...` formal argument, but it is harder to achieve.

Having drawn to the reader's attention that new-style classes really are different and have to be used in a less casual way, we should immediately point out that they do have corresponding advantages. By having a fixed structure fewer calculations are needed at run-time, and it is possible to check the validity of objects against the class structure and so avoid storing incorrectly-structured objects. A substantial amount of information about classes is stored in *metadata*, objects stored in a special area of the S database (usually directory `.Data/__Meta`).

New-style classes are made up of *slots*. These are similar to but distinct from components of a list, in that the number of slots and their names and classes are specified when a class is created: objects are extracted from slots by the `@` operator.

New-style classes are optional: old-style classes still exist and are still widely used in S-PLUS 5.x. We hope though that the new mechanisms *will* be used for new projects.

Creating a class

A class is created by the function `setClass`. Its first argument is the name of the class, and its `representation` argument specifies the slots. For example, a class to represent the locations of fungi might have

```
setClass("fungi", representation(x="numeric", y="numeric",
                                 species="character"))
```

Note how the class is made up by combining other classes. Sometimes we might wish to do this recursively, so we might instead have

```
setClass("xyloc", representation(x="numeric", y="numeric"))
setClass("fungi", representation("xyloc", species="character"))
```

This gives the same structure, as giving an *unnamed* argument to `representation` requests that the slots from that class be included.

Once a class has been created, it can be examined by `getClass`.

```
> getClass("fungi")

Slots:
         x         y     species
 "numeric" "numeric" "character"

Extends:
Class "xyloc" by direct inclusion
```

The last statement means that wherever an object of class `"xyloc"` is needed, one of class `"fungi"` can be used and the corresponding slots will be extracted and used.

To create an object from the class, use `new`, for example

```
field1 <- new("fungi", x=runif(25), y=runif(25),
              species=sample(letters[1:5], 25, rep=T))
```

We specify the contents of each slot; note that although the `species` argument is not a character, it will be coerced to a character for internal storage.

To examine the new object, just type its name:

```
> field1
An object of class "fungi"

Slot "x":
 [1] 0.960659 0.937460 0.044102 0.764619 0.705858 0.503551
 [7] 0.928648 0.840273 0.547102 0.487805 0.398985 0.263520
[13] 0.925925 0.428515 0.960061 0.298338 0.577216 0.488645
[19] 0.159737 0.182527 0.213183 0.265986 0.732724 0.643868
[25] 0.897490

Slot "y":
 [1] 0.499815 0.576944 0.905814 0.014425 0.746566 0.421574
 [7] 0.494986 0.725713 0.071926 0.279652 0.975037 0.509670
[13] 0.772228 0.980534 0.714942 0.471561 0.119848 0.716741
[19] 0.303823 0.775617 0.136685 0.017758 0.791863 0.723880
[25] 0.880876

Slot "species":
 [1] "b" "a" "d" "c" "b" "c" "d" "a" "a" "d" "a" "c" "b" "b" "c"
[16] "d" "d" "d" "a" "d" "c" "a" "d" "d" "a"
```

This calls `show` (not `print`); we will do better later.

Users will rarely see a call to `new`, as it is usually used only within functions that create objects. We might for example use

```
fungi <- function(x,y,species) new("fungi",x=x,y=y,species=species)
```

It is tempting to check here that the arguments are sensible, for example of the same length. However, that is a sanity check we should apply to all objects of the class.

A class can be removed by a call to `removeClass`.

Validity checking

Our specification of the class `"fungi"` is incomplete: we want all the slots to be of the same length. We could do this by

```
validFungi <- function(object)
{
  len <- length(object@x)
  if(length(object@y) != len || length(object@species) != len)
    return("mismatch in lengths of slots")
  else return(TRUE)
}
```

and either of

```
setClass("fungi", representation("xyloc", species="character"),
                  validity = validFungi)
setValidity("fungi", validFungi)
```

Be careful here: this sets the checking function to the *current* version of the function named. If you change the function, use `setValidity` again.

Objects are checked for validity whenever they are assigned on a permanent database.

Looking at classes

We have already seen that `getClass` will give the current definition, and this includes a validity function if one is defined. A more complete picture can be obtained by `dumpClass`, which writes all the information on a file:

```
> dumpClass("fungi")
[1] "fungi.class.q"
> !cat fungi.class.q
setClass("
fungi
",
  representation=
representation("xyloc", species = "character")
,
    validity = function(object)
{
        len <- length(object@x)
        if(length(object@y) != len ||
             length(object@species) != len)
                return("mismatch in lengths of slots")
        else return(TRUE)
}
)
```

This gives all the information needed to re-create the class.

Information on the slots can be obtained by

```
> getSlots("fungi")
         x         y     species
 "numeric" "numeric" "character"
> slotNames("fungi")
```

```
[1] "x"       "y"       "species"
> hasSlot(field1, "x")
[1] T
```

The first two can also be applied to an object of the class.

Prototypes

All classes have a prototype stored with their definitions; the default prototype is to take for each slot a new object of the appropriate class. This is given by `new` if no other arguments are supplied:

```
> new("fungi")
An object of class "fungi"
Slot "x":
numeric(0)
Slot "y":
numeric(0)
Slot "species":
character(0)
```

Sometimes another prototype would be more appropriate, and this can be specified by the `prototype` formal argument to `setClass` or at a later stage by a call to `setPrototype`.

More usefully, classes can have a prototype but not a representation. One such class is `"sequence"`,

```
> getClass("sequence")

No slots; prototype of class "integer"
new("sequence")
An object of class "sequence"

numeric(0)
```

This is a class like `"integer"` but with no implied promise that it would do where an integer vector was required.

Virtual classes

Virtual classes usually have neither representation nor prototype, and normally there are no objects of that class. They sound useless, but are made useful by the inheritance between classes. We have already pointed out the class `named` for vectors with names.

```
> getClass("named")

Slots:
    .Data      .Names
 "vector" "character"
```

```
Extends:
Class "structure" by direct inclusion
Class "vector" indirectly through class "structure"
Class "logical" indirectly through class "vector"
Class "single" indirectly through class "vector"
Class "integer" indirectly through class "vector"
Class "numeric" indirectly through class "vector"
Class "complex" indirectly through class "vector"
Class "string" indirectly through class "vector"
Class "character" indirectly through class "vector"
    ....
```

This is able to extend all the basic vector classes by including a slot of class `"vector"`, which is a virtual class.

```
> getClass("vector")
Virtual Class

No slots; prototype of class "NULL"

Extends:
Class "complex" as a possible instance of the virtual class
Class "string" as a possible instance of the virtual class
Class "character" as a possible instance of the virtual class
Class "logical" as a possible instance of the virtual class
Class "list" as a possible instance of the virtual class
Class "single" as a possible instance of the virtual class
    ....
```

Having `"vector"` as a virtual class allows methods to be written that just assume that a class has a sequential index of 'elements'. (This abstraction can make coding very economical.) Note that lists are a vector class, as indeed is class `"directory"`.

Creating virtual classes is easy: either give no representation or include the special class `"VIRTUAL"` in the representation.

The one exception to the rule that virtual classes do not have any direct members is in fact a very common one: old-style classes are implemented as virtual classes, for example

```
> getClass("lm")
Virtual Class

No slots; prototype of class "NULL"
```

Inheritance

The power of an object-oriented language stems from the relationships between classes. In S there are 'is' and 'as' relationships. Class A 'is' (extends, inherits from) class B if `is(x, "B")` is true for all objects `x` of class A; in essence this means that whenever an object of class B is required, `x` is acceptable. This is reflected in

```
> extends("fungi", "xyloc")
[1] T
```

This relationship can also be conditional: only some objects of class `"vector"` are of class `"integer"`. However

```
> extends("vector", "integer")
[1] T
```

How can this be? In fact `extends` is being generous, and

```
> extends("vector", "integer", maybe=NA)
[1] NA
```

shows a more complete picture. Of course in the other direction inclusion is always true:

```
> extends("integer", "vector", maybe = NA)
[1] T
```

S gets its information on 'is' relationships from the class definitions, specifically the inclusion of one class in the definition of another, and from `setIs` relationships. We can use `setIs` to set conditional inheritance, for example

```
setIs("vector", "integer",
      test = function(object) class(object)=="integer")
```

Inheritance is transitive, and most of the relationships listed by `getClass` come through transitivity.

It looks as if class `"integer"` should extend class `"numeric"`, and it does. This relies on some coercion, as the internal representation of the classes is different. If necessary such coercion can be set by the `coerce` argument to `setIs`.

Sometimes transitivity gives inheritance that we do not want, or conditional inheritance via a series of tests which we know cannot all be passed. We can deny inheritance by using `test = FALSE` in a call to `setIs`.

Coercion

On the other hand, no objects of class `"numeric"` are regarded as of class `"integer"`

```
> extends("numeric", "integer", maybe=NA)
[1] F
```

although some could be. What we do have is a way to *coerce* objects of class `"numeric"` to class `"integer"`, and this is the function `as`

```
> as(c(1.,2.,3.,4.4),  "integer")
[1] 1 2 3 4
```

(Careful: `1:3` was integer even in earlier versions.) Function `as` derives its information from `setAs` declarations, and tends to be much more demanding than `as.xxx` functions. Function `as` is implemented via `coerce`, so we can find 'as' relations as methods for `coerce`:

```
> showMethods("coerce")
      Database            object                to
 [1,] "main"   "ANY"             "ANY"
 [2,] "main"   "character"       "string"
 [3,] "main"   "numeric"         "integer"
    ....
> selectMethod("coerce", signature("numeric", "integer"))
function(object, to)
as.integer(object)
```

Generic and method functions

We wanted a better printout for objects of class `"fungi"`. Automatic printing is done by function `show`, a function with a single argument, `object`. We can write a suitable method function and declare it by

```
functionArgNames("show")
[1] "object"
show.fungi <- function(object) {
  tmp <- rbind(x = format(round(object@x, 2)),
               y = format(round(object@y, 2)),
               species = object@species)
  dimnames(tmp)[[2]] <- rep("", length(object@x))
  print(tmp, quote=F)
  invisible(object)
}
setMethod("show", "fungi", show.fungi)
```

Note that this declares the current version of `show.fungi` as the method: changes to `show.fungi` would not be reflected in the methods used.

Further thought suggests that this is not a good idea, and anyway we need a method for `print`, which is an old-style function. So we may prefer to use

```
print.fungi <- function(x, digits=2) {
  tmp <- rbind(x = format(round(x@x, digits)),
               y = format(round(x@y, digits)),
               species = x@species)
  if(!is.null(tmp)) {
    dimnames(tmp)[[2]] = rep("", length(x@x))
    print(tmp, quote=F)
  } else cat("empty object of class fungi\n")
  invisible(x)
}
setMethod("show", "fungi", function(object) print.fungi(object))
```

Then we can say `field1` or `show(field1)` or `print(field1)` to equal effect. The advantage of `print` is that it can be more flexible by not being bound to a single argument.

We would also like a `plot` method for this class. However, `plot` is a confusing function:

```
> functionArgNames("plot")
[1] "x"   "y"   "..."
> plot
function(x, ...) UseMethod("plot")
> selectMethod("plot", "fungi")
function(x, y, ...) {
  xyCall(x, y, function(x, y, xlab, ylab, ...)
    .Internal(plot("zplot", x = x, y = y, xlab = xlab,
                   ylab = ylab, ...), "call_S_Version2"),
              xexpr = substitute(x), yexpr = substitute(y), ...)
}
```

`selectMethod` correctly says what will be called. We need to write and declare a method by, for instance

```
plot.fungi <- function(x, y, ...) {
  oldpar <- par(pty="s")
  on.exit(par(oldpar))
  plot(x@x, x@y, type="n", xlab="x", ylab="y")
  text(x@x, x@y, labels=x@species)
  title(deparse(substitute(x)))
}
setMethod("plot", "fungi", plot.fungi)
```

Note that we do need exactly this set of arguments, even if we do not use `y`. Trying to use a different set of arguments (even different names) will give warnings and sometimes unpredictable behaviour: for example `match.call` is unlikely to work. Since we only specified the class for one argument of `plot`, our method will work for any (including missing) argument `y`. Perhaps a safer approach is

```
setMethod("plot", signature(x="fungi", y="missing"), plot.fungi)
```

that insists that `y` really is missing.

Methods for replacement functions are set by `setReplaceMethod` rather than `setMethod`.

Method dispatch

In S-PLUS 3.x methods were selected on the first argument. In S-PLUS 5.x, two or more arguments can be used to select the method. The second argument to `setMethod` is a *signature* specifying the classes of the named or unnamed arguments to be used in method dispatch. An explicit class refers to the first argument, but otherwise the signature should be constructed via a call to `signature`.

The function `selectMethod` shows the method that would be selected for a particular signature, and `showMethods` will list all the methods for the function (which can be many). Function `hasMethod` will say if a non-default method will be found for that signature, and `existsMethod`, `getMethod` and `findMethod` look for a method which is defined for the specified signature (rather than being found by inheritance). Function `dumpMethod` will write the method to a file as a call to `setMethod`.

A method can be removed by a call to `removeMethod`.

Generic functions

Normally there is no need to declare a generic function: setting a method for a function automatically makes it generic with the old definition as the default method and the arguments of the old method defining the (fixed) set of arguments of the generic function. The function `isGeneric` will reveal if this has happened.

It is also possible to declare a generic function directly via `setGeneric`. An example might be

```
setGeneric("lda", function(x, y, ...) {
             res = standardGeneric("lda")
             res@call = match.call()
             res
           })
```

which (unlike `UseMethod`) allows some post-processing of the calls to each method.

Old-style classes

We have seen that old-style classes are virtual classes in the new style. Set them via a call to `oldClass` rather than by calling `class` or setting attribute `"class"` (perhaps by returning a `structure`, a favourite coding idiom of one of us). Thus `lm.fit.qr` now ends

```
  oldClass(fit) <- cl
  fit
}
```

Use `oldUnclass` rather than `unclass` for old-style classes.

Objects can now have only one old-style class, whereas `glm` objects, for example, used to have class `c("glm", "lm")`. This can be emulated by setting explicit inheritance, for example by

```
setOldClass(c("glm", "lm"))
```

but that will apply to all objects of the class.

4.5 Editing, correcting and documenting functions

Locating and correcting errors

The interactive debugger `inspect` is not available in S-PLUS 5.x, so debugging uses `traceback`, `browser` and `debugger`.

Our working example on page 141 is no longer an error as `var` has been re-written. Let us consider another example.

```
> ir.species <- factor(c(rep("s",50), rep("c", 50),
     rep("v", 50)))
> ird <- data.frame(rbind(iris[,,1], iris[,,2], iris[,,3]),
     Species=ir.species)
> lda(species ~ ., ird)
Problem: Object "species" not found
Use traceback() to see the call stack
> traceback()
8: eval(action, sys.parent())
7: doErrorAction("Problem: Object \"species\" not found", 1000)
6:
5: model.frame.default(formula = species ~ ., data = ird)
4: model.frame(formula = species ~ ., data = ird)
3: eval(m, sys.parent())
2: lda(species ~ ., ird)
1:
Message: Problem: Object "species" not found
```

This may be clear enough, but we can investigate further by changing the `options` setting `error` either to `dump.frames` and use the `debugger` or to use the new function `recover`. We will try the latter.

```
options(error=recover)
> lda(species ~ ., ird)
Problem: Object "species" not found
Debug ?  ( y|n ): y
Browsing in frame 6
Local Variables: Petal.L., Petal.W., Sepal.L., Sepal.W., Species

R>
```

The prompt indicate that we are in a modified browser. Use `c` or `q` to quit from the browser (and also when in the browser called by `browser` or `debugger`). Often the most useful things to do are to find out where we are and remind ourselves what we can do.

```
R> where
*6:  from 1
5: model.frame.default(formul.... from 1
4: model.frame(formula = spec.... from 1
3: eval(m, sys.parent()) from 2
```

```
2: lda(species ~ ., ird) from 1
1:  from 1
R> ?
Type any expression. Special commands:
‘up’, ‘down’ for navigation between frames.
‘where’ # where are we in the function calls?
‘dump’  # dump frames, end this task
‘q’     # end this task, no dump
‘go’    # retry the expression, with corrections made
Browsing in frame 6
Local Variables: Petal.L., Petal.W., Sepal.L., Sepal.W., Species
```

The answer is obvious from the list of local variables, but if it were not, we would need to look at a higher frame.

```
R> up
Browsing in frame 1
Local Variables: .Last.expr, .Options, down, q, stop, up, where

NULL
```

This comes as a surprise; the use of `eval` seems to have confused the hierarchy of frames. Further, if we reach frame 1 we need to use `where()`, `down()` and so on. Normally using `dump` and then calling `debugger` will enable the hierarchy to be navigated. Let us try that

```
> options(error=dump.frames)
> lda(species ~ ., ird)
Problem: Object "species" not found
Evaluation frames saved in object "last.dump", use debugger()
  to examine them
> debugger()
Message: Problem: Object "species" not found
browser: Frame 20
b()> where
*20: list() from 17
17: model.frame.default(formul.... from 14
14: model.frame(formula = spec.... from 11
11: eval(m, sys.parent()) from 8
8: lda(species ~ ., ird) from 5
5: list() from 1
2: debugger() from 1
1:  from 1
```

This does enable all the hierarchy to be explored.

Creating a help document – 5.1

The `prompt` function still creates a skeleton help file, now in the SGML dialect used by S-PLUS. So

```
> prompt(Ttest)
created file named  Ttest.sgm  in the current directory
 edit the file and move it to the appropriate __Shelp directory.
> !vi Ttest.sgm
> !mkdir -p .Data/__Shelp  .Data/__Hhelp  # may not be necessary
> !cp Ttest.sgm .Data/__Shelp
> ?Ttest
```

The skeleton file will look like

```
<s-function-doc>
<s-topics>
<s-topic> Ttest </s-topic>
</s-topics>
<s-title>
<!--function to do???-->
</s-title>
<s-description>
<!--brief description-->
</s-description>
<s-usage>
<s-old-style-usage>
Ttest(z, ...)
</s-old-style-usage>
</s-usage>
<s-args-required>
</s-args-required>
<s-args-optional>
<!--move the above two lines to just above the first optional argument-->
<s-arg name= z >
<!--Describe z here-->
</s-arg>
<s-arg name= ... >
<!--Describe ... here-->
</s-arg>
</s-args-optional>
<s-value>
<!--Describe the value returned-->
</s-value>
<s-side-effects>
<!--describe any side effects if they exist-->
</s-side-effects>
<s-details>
<!--explain details here-->
</s-details>
<s-section name = "REFERENCES">
<!--put references here, make other sections like NOTE and WARNING
with s-section-->
</s-section>
<s-see>
<!--put functions to SEE ALSO here-->
```

```
</s-see>
<s-examples>
<s-example>
<!-- Put 1 or more s-example tags here -->
</s-example>
</s-examples>
<s-keywords>
<s-keyword>
<!-- Put one or more s-keyword tags here -->
</s-keyword>
</s-keywords>
<s-docclass>
function
</s-docclass>
</s-function-doc>
```

A line such as `<!--function to do???-->` is an SGML comment. The layout will look similar to HTML, as HTML is a different SGML dialect.

4.6 Calling the operating system

There is a new function to call the operating system with call

```
shell(command, input, output=T, mustWork=F)
```

which is similar to `unix`, but the error action can be controlled by argument `mustWork`: if this is true any error in the command generates an error rather than a warning or (if `output=F`) return an error status.

One can also communicate with the operating system via pipes and fifos: see page 9 of these complements.

4.8 Frames

There are functions `getFunction` and `existsFunction` which look only for functions, and can be asked to ignore generic functions; we have already seen that `getMethod` and `existsMethod` search for methods for a specified generic function and specified signature.

We have given only a very brief account of expressions and evaluation. There is a new function `Quote` that given an expression as an argument returns the expression unevaluated, and so can be useful when manipulating expressions. It is closely related to `substitute`, which also returns an unevaluated expression, but after substituting names from its second argument, by default the local frame, and to `expression` which makes its argument into an expression object.

4.9 Using C and FORTRAN routines

The S-PLUS program does much of its computation through routines written in either FORTRAN or C. User-written routines can be called from the S language using the `.Fortran` or `.C` functions with a protocol that follows. (The function `polynom` on page 29 provides an example.)

- The first argument to `.C` or `.Fortran` is a character string giving the name[1] of the routine.
- Each further argument must match the argument of the routine. In particular the data passed through to the routine must have the correct `storage.mode` and must match the argument in length. Unlike S, neither FORTRAN nor C can deduce the length or mode of arguments.
- The arguments may be given name fields. These do not match anything in the routine itself, but will be retained as name fields in the result.
- The value returned by the call to `.C` or `.Fortran` is a list containing all the arguments passed to the routine. The components of the list will reflect any changes made by the routine. Any attributes of the arguments will be retained so that arrays will return as arrays, for example.

The storage modes for arguments and their C and FORTRAN counterparts are given in Table 4.4. Note carefully that the `integer` and `logical` modes correspond to `long` in C, not `int`, and that these do differ on the DEC Alpha platform. (A substantial proportion of user-contributed software needs correction.)

Table 4.4: Argument storage modes in S and corresponding data types for C and FORTRAN routines where applicable. (From Table 11.1 on page 412 of Chambers, 1998.) The C type `complex` is defined via `S.h` as a structure with `double` components `re` and `im`.

S class	C	FORTRAN
`"numeric"`	`double *`	`DOUBLE PRECISION`
`"integer"`	`long *`	`INTEGER`
`"single"`	`float *`	`REAL`
`"logical"`	`long *`	`LOGICAL`
`"complex"`	`complex *`	`COMPLEX`
`"character"`	`char **`	`CHARACTER (*)`
`"raw"`	`char *`	
`"list"`	`s_object **`	

Notice from Table 4.4 that the allowable argument types in C routines are all *pointers*. This is because the quantities manipulated are S vectors and so must be accessed by C indirectly. The case of character objects needs a little care. Recall that a character object is a *vector* of character strings; each string is an array of

[1] in lower case for `.Fortran` on Unix.

```
static double horner(double x, double *b, int n)
{
    int i;
    double p = b[n];
    for(i = n-1; i >= 0; i--)
        p = b[i] + x * p;
    return p;
}

void
poly(long int *m, double *p, double *x, long int *n, double *b)
{
    long i;
    for (i = 0; i < *m; i++)
        p[i] = horner(x[i], b, (int)*n);
}
```

Figure 4.2: File `horner.c`: C functions to evaluate a polynomial using Horner's scheme.

characters terminated by the ASCII character `NUL` (which has numeric code `0`). This maps naturally to type `char **` in C, but in FORTRAN character strings are stored as fixed-length one-dimensional character arrays, and the `.Fortran` interface allows only a single character string to be passed. (The length should be passed as a separate argument.)

To illustrate the process consider a simple example. Figure 4.2 shows a pair of C routines that together evaluate a polynomial using a Horner scheme. Notice that the C function `horner` cannot be called directly from S since its arguments are not all pointers, but `poly` does conform. (The function `horner` also has `int` not `long`, and returns a result, which cannot be used by `.C`.) Notice also that `poly` evaluates the polynomial for a vector of `x` values and returns via `p` a vector of results. This is in line with the preferred style of S functions. To avoid inadvertently overwriting existing symbols it is a good idea to declare as `static` all internal functions in C code.

Supposing these functions have been loaded into our copy of S-PLUS (see below) we can write a function to use them. The argument `b` is the vector of polynomial coefficients (including the constant).

```
polynom = function(x, b)
{
  m = as.integer(length(x))
  n = as.integer(length(b)-1)
  p = x
  .C("poly", m, val = p, x, n, b,
     CLASSES = c("integer", rep("numeric", 2), "integer",
                 "numeric"),
     COPY = rep(F, 5))$val
```

```
}
```

Since only the result component is needed, we only return that component. Note that by returning the result 'in place' we retain the attributes of `x`, such as its dimensions. Note that we specify the classes of each argument (to ensure that S does any needed coercions) and that the C code is not going to change any of the arguments that we are not using as named components, so no copies needed to be made of the other components.

To evaluate the polynomial $x^2 - 1$ for each element of a 3×3 matrix we can call `polynom`:

```
> mat = matrix(1:9,3,3)
> polynom(mat, -1:1)
     [,1] [,2] [,3]
[1,]    0   15   48
[2,]    3   24   63
[3,]    8   35   80
```

It is important to note that matrices and arrays in S will be passed to the C function as vectors, not as multiply subscripted arrays, although as we have noted the class will be preserved in the result.

Our libraries provide several useful examples of calls to C routines, for example in the functions `sammon` and `nnet`.

Unlike all previous versions of S-PLUS, it is now claimed that FORTRAN I/O can be used in compiled FORTRAN code to be loaded into S-PLUS.

There is a new interface `.Call` that allows C programmers to manipulate S objects. This is for experts only: see Chambers (1998, Appendix A) for details.

Dynamic loading

S-PLUS already has many compiled FORTRAN and C functions loaded for use by functions such as `svd`, `qr` and `eigen`. We can include other functions, such as those from our `horner.c`. This is much simpler in S-PLUS 5.x, which has only one method[2] equivalent to the dynamic loading of shared libraries on earlier Unix systems and now called .

To include compiled code, the easiest way is to include the files in a call to `Splus5 CHAPTER`, for example

```
Splus5 CHAPTER horner.c
Splus5 make
```

This will create a shared library called `S.so` in the chapter. Then the next time S-PLUS is started in that chapter, `S.so` will be loaded.

It is possible to use `dyn.open` and `dyn.close` to load or unload a shared library, but this is not normally necessary. Sometimes it easiest to use `dyn.open` to re-load the routines after re-compiling them.

The function `is.loaded` can still be used to check if a routine is available, as can `dyn.exists`

[2] in particular, static loading is not available.

```
> is.loaded(symbol.C("poly"))
[1] T
> dyn.exists(symbol.C("poly")) != 0
[1] T
```

The rest of the information in Section 4.11 of the on-line programming complements applies equally to S-PLUS 5.x. Some of the calls have changed slightly, to

`char *Salloc(long n, type)`	allocate `n` items of requested type and set them to zero.
`char *Srealloc(char *p, long new, long old, type)`	reallocate `new` items of requested size, for pointer `p` to a block of size `old`. Allocated memory is zeroed.
`type *Calloc(int n, type)`	`calloc` `n` items of type `type`.
`type *Realloc(char * p, int n, type)`	`realloc` pointer `p` to `n` items of type `type`
`Free(char *p)`	'free' memory pointed to by `p`.
`double unif_rand(S_evaluator)`	one uniform random number.
`double norm_rand(S_evaluator)`	one standard normal variate.
`setseed(long *seed)`	as in `set.seed`.
`seedin(long *iseed, S_evaluator)`	set the seed. `iseed` should be either a pointer to a vector of 12 integers between 0 and 63, or `NULL`, when the seed is read in from `.Random.seed`.
`seedout((long*)NULL, S_evaluator)`	write the seed back out to `.Random.seed` after the random variates have been generated.

Functions which call `S_evaluator` *may* need to include the macro `S_EVALUATOR` (note, not followed by `;`) at the top of the function body.

There are new calls for handling NAs in C code given in Chambers (1998, p. 425).

4.12 Object orientation: an extended statistical example

The corresponding section in the on-line programming complements discusses how we converted the function `lda` from a single function to a generic for several different types of argument, and added suitable method functions. Here we discuss how to take that work and to convert it to the new-style class structure of S-PLUS 5.x.

Some fundamental design (in our case re-design) decisions have to made early, for new-style classes are rigorously defined, and all methods for a new-style generic must have identical call sequences. First the class. Objects of the old class `"lda"` are lists with components

```
prior, counts, means, scaling, lev, svd, N, call
```

and perhaps `terms` (only present in objects created by `lda.formula`). We can convert this to a new-style class by giving the objects the following slots

```
setClass("lda", representation(prior =   "named",
                               counts =  "named",
                               means =   "matrix",
                               scaling = "matrix",
                               lev =     "character",
                               svd =     "numeric",
                               N =       "integer",
                               call =    "call")
         )
```

Note that we have to specify the classes of the slots very carefully: numeric and integer slots do not have names attributes, but slots of class `"named"` do. We do not include a slot for `terms`, as `"terms"` is an old-style class with no fixed representation.

Our next decision is the set of arguments for all the methods. We had some choice here, and experimented with naming all the possible arguments for each method or making use of ‘ . . . ’. We settled on

```
lda = function(x, y, ...)
  stop("lda not implemented for class ", class(x))
```

as a function that will become the default method. The idea is that `x` and `y` will represent either the matrix `x` and `grouping` or `formula` and `data`, and all other arguments must be named exactly. Since the previous `lda.default` is not going to be one of the methods, we rename it to have call

```
lda1 = function(x, grouping, prior = NULL, tol = 1.0e-4,
                method = c("moment", "mle", "mve", "t"), CV=F,
                nu = 5, ...)
{
    ....
}
```

We can now begin to set up methods. Every object in S-PLUS 5.x has a class, so in that sense the process is easier here. The methods for matrices, Matrices and data frames are easy:

```
lda.data.frame <- function(x, y, ...) {
  x <- as.matrix(x)
  callGeneric()
}
```

```
lda.Matrix <- function(x, y, ...) {
  x <- data.matrix(x)
  callGeneric()
}

lda.matrix <- function(x, y, ...) {
  x <- as(x, "matrix")
  dots <- list(...)
  if(hasArg(subset)) {
    subset <- dots$subset
    x <- x[subset, , drop = F]
    y <- y[subset]
  }
  if(hasArg(na.action)) {
    na.action <- dots$na.action
    dfr <- na.action(data.frame(g = y, x = x))
    y <- dfr$g
    x <- dfr$x
  }
  res <- lda1(x = x, grouping = y, ...)
  res@call <- match.call()
  res
}

setMethod("lda", "matrix", lda.matrix)
setMethod("lda", "Matrix", lda.Matrix)
setMethod("lda", "data.frame", lda.data.frame)
```

We could have used inheritance for Matrices and data frames, but when this was implemented that was not complete, so we choose to be explicit. Note that rather than call `NextMethod`, we call the generic again with `callGeneric`. The generic is no longer the function `lda` that we defined, which has become the default method (and listing functions can be misleading).

The other features to note are that `call` is now a slot set by the `@` operator, and the way missing arguments are handled. They are definitely treated differently: explicit arguments of generic functions which are missing are of class `"missing"` and `missing()` does not work for them. Arguments in `...` can be checked for by `hasArg` provided they are named in the call. That was our design decision for `subset` and `na.action` when they followed `...`, so is safe here. Note that these two arguments (if present) are passed down to `lda1` and swallowed by its unused `...` argument.

We chose to make more extensive changes for a `formula` method.

```
lda.formula <- function(x, y, ...)
{
  data <- as.data.frame(y)
  m <- match.call(expand.dots = F)
  m$... <- NULL
```

```
  dots <- list(...)
  names(m)[2:3] <- c("formula", "data")
  if(hasArg(subset)) m$subset <- dots$subset
  if(hasArg(na.action)) m$na.action <- dots$na.action
  m[[1]] <- as.name("model.frame")
  m <- eval(m, sys.parent())
  Terms <- attr(m, "terms")
  y <- model.extract(m, "response")
  x <- model.matrix(Terms, m)
  xint <- match("(Intercept)", dimnames(x)[[2]], nomatch=0)
  if(xint > 0) x <- x[, -xint, drop=F]
  res <- lda1(x <- x, grouping = y, ...)
  Call <- match.call()
  Call$x <- as.call(attr(Terms, "formula"))
  res@call <- Call
  res
}
setMethod("lda", "formula", lda.formula)
```

First, the argument names have changed (they have to be the same as the other methods) and so the lazy way of calling `model.frame` has to be replaced by an explicitly constructed call. Second, we cannot add the `terms` component, as it is not in the design of the class object. So we note that we will have to cope with this in methods for the new class. It fact there was a lazier way to do this, by not calling `lda.formula` directly.

```
setMethod("lda", "formula",
          function (x, y, ...)  oldlda.formula(x, y, ...))

oldlda.formula <- function(formula, data = NULL, ...,
                           subset, na.action = na.fail)
{
  m <- match.call(expand.dots = F)
  if(is.matrix(eval(m$data, sys.parent())))
    m$data <- as.data.frame(data)
  m$... <- NULL
  m[[1]] <- as.name("model.frame")
  m <- eval(m, sys.parent())
  Terms <- attr(m, "terms")
  grouping <- model.extract(m, "response")
  x <- model.matrix(Terms, m)
  xint <- match("(Intercept)", dimnames(x)[[2]], nomatch=0)
  if(xint > 0) x <- x[, -xint, drop=F]
  res <- lda1(x, grouping, ...)
  Call <- match.call()
  Call$x <- as.call(attr(Terms, "formula"))
  Call[[1]] <- as.name("lda")
  res@call <- Call
  res
}
```

The snag with this approach is that the call needs to be manipulated. Note that we manipulate it anyway, to expand out any `.` on the right-hand side of the formula for future use.

The changes to `lda1` are mainly in constructing the returned object. We have also to ensure that `counts` has the right class (`table` returns a one-dimensional array).

```
lda1 <- function(x, grouping, prior = NULL, tol = 1.0e-4,
                 method = c("moment", "mle", "mve", "t"), CV=F,
                 nu = 5, ...)
{
    ....
  lev <- lev1 <- levels(g)
  counts <- as.vector(table(g))
  names(counts) <- lev
    ....
  res <- new("lda")
  res@prior <- prior
  res@counts <- counts
  res@means <- group.means
  res@scaling <- scaling
  res@lev <- lev
  res@svd <- X.s$d[1:rank]
  res@N <- n
  res
}
```

We need some changes in the methods, apart from replacing `$` by `@` to reflect the change from a list to a representation with slots. We also need to declare a means of displaying the object, as `show` will not automatically call `print.lda` but rather would list the slots.

```
setMethod("show", "lda", function(object) print.lda(object))

print.lda <- function(x, ...)
{
  if(!is.null(cl <- x@call)) {
    cat("Call:\n")
    dput(cl)
  }
  cat("\nPrior probabilities of groups:\n")
  print(x@prior, ...)
  cat("\nGroup means:\n")
  print(x@means, ...)
  cat("\nCoefficients of linear discriminants:\n")
  print(x@scaling, ...)
  svd <- x@svd
  names(svd) <- dimnames(x@scaling)[[2]]
  if(length(svd) > 1) {
    cat("\nProportion of trace:\n")
```

```
    print(round(svd^2/sum(svd^2), 4), ...)
  }
  invisible(x)
}
```

Next, we need to recover from having no `terms` component. We used the following scheme in `predict.lda`.

```
predict.lda <-
function(object, newdata, prior = object@prior, dimen,
         method = c("plug-in", "predictive", "debiased"), ...)
{
  if(!inherits(object, "lda")) stop("object not of class lda")
  if((missing(newdata) ||!is(newdata, "model.matrix")) &&
      is.form(form <- object@call[[2]])) { #
    # formula fit
    if(missing(newdata)) newdata <- model.frame(object)
    else newdata <- model.frame(delete.response(terms(form)), newdata,
                                na.action=function(x) x)
    x <- model.matrix(delete.response(terms(form)), newdata)
    xint <- match("(Intercept)", dimnames(x)[[2]], nomatch=0)
    if(xint > 0) x <- x[, -xint, drop=F]
  } else { #
    # matrix or data-frame fit
    if(missing(newdata)) {
      if(!is.null(sub <- object@call$subset))
        newdata <- eval(parse(text=paste(deparse(object@call$x),"[",
                              deparse(sub),",]")), sys.parent())
      else newdata <- eval(object@call$x, sys.parent())
      if(!is.null(nas <- object@call$na.action))
        newdata <- eval(call(nas, newdata))
    }
    if(is.null(dim(newdata)))
      dim(newdata) <- c(1, length(newdata))  # a row vector
    x <- as.matrix(newdata)             # to cope with dataframes
  }
    ....
is.form <- function(x) is.call(x) && (x[[1]] == "~")
```

We test for an object produced by `lda.formula` by testing the first argument of the call (which is of class `"call"`, not `"formula"` as one might expect). We can re-create the `terms` object from the formula. Note that we do need a method for `model.frame` now, as we no longer have a list object, and the names of our arguments in the call are non-standard for a model-fitting function.

```
model.frame.lda <-
function(formula, data = NULL, na.action = NULL, ...)
{
  oc <- formula@call
  oc[[1]] <- as.name("model.frame")
  names(oc)[2:3] <- c("formula", "data")
```

```
  oc$prior <- oc$tol <- oc$method <- oc$CV <- oc$nu <- NULL
  oc[[1]] <- as.name("model.frame")
  if(length(data)) {
    oc$data <- substitute(data)
    eval(oc, sys.parent())
  }
  else eval(oc, list())
}
```

Finally, `update` will no longer work, and we need

```
formula.lda <- function(object) object@call[[2]]
```

and `update.lda` which is `update.default` with `object$call` replaced by `object@call`. The problem with using new-style classes when the S model-fitting functions have not been converted is that many of the utility functions expect a list as the class object. Perhaps in due course they will be updated to look for slots or lists.

Chapter 6

Linear Statistical Models

6.7 Multiple comparisons

As we all know, the theory of p-values of hypothesis tests and of the coverage of confidence intervals applies to pre-planned analyses. However, the only circumstances in which an adjustment is routinely made for testing after looking at the data is in multiple comparisons of contrasts in designed experiments. Consider the experiment on yields of barley in our dataset `immer`[1]. This has the yields of five varieties of barley at six experimental farms in both 1931 and 1932; we will average the results for the two years. An analysis of variance gives

```
> immer.aov <- aov((Y1+Y2)/2 ~ Var + Loc, data=immer)
> summary(immer.aov)
          Df Sum of Sq Mean Sq F Value     Pr(F)
      Var  4      2655   663.7   5.989 0.0024526
      Loc  5     10610  2122.1  19.148 0.0000005
Residuals 20      2217   110.8
```

The interest is in the difference in yield between varieties, and there is a statistically significant difference. We can see the mean yields by a call to `model.tables`.

```
> model.tables(immer.aov, type="means", se=T, cterms="Var")
    ....
 Var
      M      P      S      T      V
 94.392 102.54 91.133 118.2 99.183

Standard errors for differences of means
          Var
        6.078
replic. 6.000
```

This suggests that variety `T` is different from all the others, as a pairwise significant difference at 5% would exceed $6.078 \times t_{20}(0.975) \approx 12.6$; however the comparisons to be made have been selected after looking at the fit.

[1] the Trellis dataset `barley` discussed in Cleveland (1993) is a more extensive version of the same dataset.

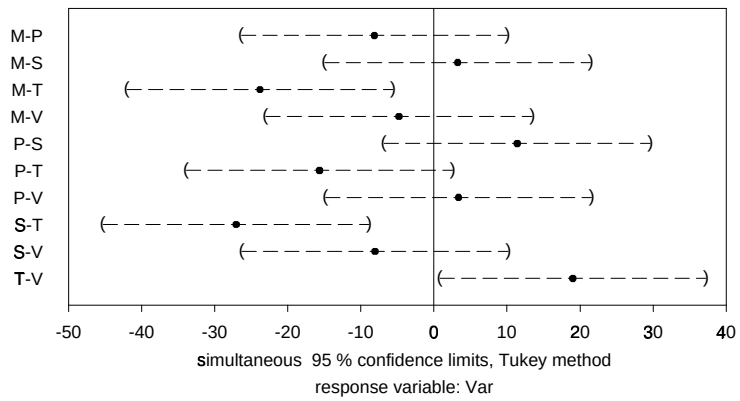

Figure 6.7: Simultaneous 95% confidence intervals for variety comparisons in the `immer` dataset.

Function `multicomp` allows us to compute *simultaneous* confidence intervals in this problem, that is confidence intervals such that the probability that they cover the true values for all of the comparisons considered is bounded above, by 5% for 95% confidence intervals. We can also plot the confidence intervals (Figure 6.7), by

```
> multicomp(immer.aov, plot=T)
95 % simultaneous confidence intervals for specified
linear combinations, by the Tukey method

critical point: 2.9925
response variable: Var

intervals excluding 0 are flagged by '****'

    Estimate Std.Error Lower Bound Upper Bound
M-P    -8.15      6.08     -26.300       10.00
M-S     3.26      6.08     -14.900       21.40
M-T   -23.80      6.08     -42.000       -5.62 ****
M-V    -4.79      6.08     -23.000       13.40
P-S    11.40      6.08      -6.780       29.60
P-T   -15.70      6.08     -33.800        2.53
P-V     3.36      6.08     -14.800       21.50
S-T   -27.10      6.08     -45.300       -8.88 ****
S-V    -8.05      6.08     -26.200       10.10
T-V    19.00      6.08       0.828       37.20 ****
```

This does not allow us to conclude that variety `T` has a significantly different yield from variety `P`.

We may want to restrict the set of comparisons, for example to comparisons with a control treatment. The dataset `oats` is discussed on page 300; here we ignore the split-plot structure.

```
> oats1 <- aov(Y ~ N + V + B, data=oats)
```

```
> summary(oats1)
          Df Sum of Sq Mean Sq F Value    Pr(F)
        N  3     20020  6673.5  28.460 0.000000
        V  2      1786   893.2   3.809 0.027617
        B  5     15875  3175.1  13.540 0.000000
Residuals 61     14304   234.5
> multicomp(oats1, focus="V")

95 % simultaneous confidence intervals for specified
linear combinations, by the Tukey method

critical point: 2.4022
response variable: N

intervals excluding 0 are flagged by '****'

                      Estimate Std.Error Lower Bound
Golden.rain-Marvellous    -5.29      4.42      -15.90
   Golden.rain-Victory     6.88      4.42       -3.74
    Marvellous-Victory    12.20      4.42        1.55
                      Upper Bound
Golden.rain-Marvellous        5.33
   Golden.rain-Victory       17.50
    Marvellous-Victory       22.80 ****

> multicomp(oats1, focus="N", comparison="mcc", control=1)
    ....
              Estimate Std.Error Lower Bound Upper Bound
0.2cwt-0.0cwt     19.5       5.1        7.24        31.8 ****
0.4cwt-0.0cwt     34.8       5.1       22.60        47.1 ****
0.6cwt-0.0cwt     44.0       5.1       31.70        56.3 ****
```

Note that we need to specify the control level: perversely by default the last level is chosen. We might also want to know if all the increases in nitrogen give significant increases in yield, which we can examine by

```
> lmat <- matrix(c(0,-1,1,rep(0, 11), 0,0,-1,1, rep(0,10),
                0,0,0,-1,1,rep(0,9)),,3, dimnames=list(NULL,
        c("0.2cwt-0.0cwt", "0.4cwt-0.2cwt", "0.6cwt-0.4cwt")))
> multicomp(oats1, lmat=lmat, bounds="lower", comparisons="none")
    ....
              Estimate Std.Error Lower Bound
0.2cwt-0.0cwt    19.50       5.1        8.43 ****
0.4cwt-0.2cwt    15.30       5.1        4.27 ****
0.6cwt-0.4cwt     9.17       5.1       -1.90
```

There are a bewildering variety of methods for multiple comparisons reflected in the options for `multicomp`. Miller (1981), Hsu (1996) and Yandell (1997, Chapter 6) give fuller details. Do remember that this tackles only part of the problem; the analyses here have been done after selecting a model and specific

factors on which to focus: the allowance for multiple comparisons is only over contrasts of one selected factor in one selected model.

Chapter 8

Robust Statistics

8.3 Robust regression

S-PLUS 4.5 introduced a new method of robust regression, `lmRobMM` due to Yohai *et al.* (1991), which is also in S-PLUS 5.x. This comes with a full set of method functions, even for `add1` and `drop1`, so can be used routinely as a replacement for `lm`. The method used is an M-estimate with a re-descending ψ function and a starting value for the optimization that is chosen as a highly resistant 'S' estimator. The optimization algorithm uses a random search, so the results will not be exactly repeatable.

Let us try it on the `phones` data.

```
> phones.lmr <- lmRobMM(calls ~ year, data=phones)
> summary(phones.lmr)
Final M-estimates.

Call: lmRobMM(formula = calls ~ year, data = phones)

Residuals:
    Min     1Q Median     3Q    Max
 -1.719 -0.46 0.2267 39.03 188.5

Coefficients:
               Value Std. Error  t value Pr(>|t|)
(Intercept) -52.3103   3.7851   -13.8199   0.0000
       year   1.0990   0.0636    17.2853   0.0000

Residual scale estimate: 2.027 on 22 degrees of freedom
Proportion of variation in response explained by model: 0.4898

Test for Bias
            Statistics P-value
 M-estimate      1.601   0.449
LS-estimate      0.243   0.886
> plot(phones.lmr)
```

This works well, rejecting all the spurious observations. The 'test for bias' is of the M-estimator against the initial S-estimator; if the M-estimator appears biased the initial S-estimator is returned.

The `compare.fits` function makes it easy to compare this fit with that from `lm` or similar functions.

```
> phones.lm <- lm(calls ~ year, data=phones)
> compare.fits(phones.lmr, phones.lm)
    ....
Coefficients:
            phones.lmr phones.lm
(Intercept)    -52.310  -260.059
       year      1.099     5.041

Residual Scale Estimates:
phones.lmr : 2.027 on 22 degrees of freedom
 phones.lm : 56.22 on 22 degrees of freedom
```

This also has `summary` and `plot` methods.

For Brownlee's stack loss data we get similar results to `rlm` (page 263) but with a smaller estimated scale.

```
> stack <- data.frame(stack.x, loss=stack.loss)
> stack.lmr <- lmRobMM(loss ~ ., data=stack)
> summary(stack.lmr, cor=F)
Final M-estimates.

Call: lmRobMM(formula = loss ~ ., data = stack)

Residuals:
   Min      1Q Median    3Q   Max
 -8.63 -0.6713 0.3594 1.151 8.174

Coefficients:
               Value Std. Error  t value Pr(>|t|)
(Intercept) -37.6525   5.0026    -7.5266   0.0000
   Air.Flow   0.7977   0.0713    11.1886   0.0000
 Water.Temp   0.5773   0.1755     3.2905   0.0043
 Acid.Conc.  -0.0671   0.0651    -1.0297   0.3176

Residual scale estimate: 1.837 on 17 degrees of freedom
```

and for the `hills` data we have

```
> summary(lmRobMM(time ~ dist + climb, data=hills,
                  weights=1/dist^2))
    ....
Coefficients:
              Value Std. Error t value Pr(>|t|)
(Intercept) -3.3745  4.8433    -0.6967  0.4910
```

```
       dist  5.5365  1.2347     4.4840  0.0001
      climb  0.0086  0.0037     2.3282  0.0264

Residual scale estimate: 0.7921 on 32 degrees of freedom

> summary(lmRobMM(ispeed ~ grad, data=hills))
Coefficients:
             Value Std. Error t value Pr(>|t|)
(Intercept)  5.0754  0.4210    12.0563  0.0000
       grad  0.0077  0.0016     4.8817  0.0000

Residual scale estimate: 0.8189 on 33 degrees of freedom
```

8.4 Resistant regression

S-PLUS 4.x and S-PLUS 5.x have an alternative formula-based interface for `lmsreg` and `ltsreg`, and `print`, `summary` and `plot` methods. We can try these on the stack loss example.

```
> stack <- data.frame(stack.x, loss=stack.loss)
> lmsreg(loss ~ ., data=stack)
$coefficients:
 Intercept Air.Flow Water.Temp Acid.Conc.
    -39.25     0.75        0.5          0
$scale:
        Y
 1.207615
$residuals:
    1    2   3    4     5     6     7    8     9   10   11   12
 7.75 2.75 7.5 8.75 -0.25 -0.75 -0.25 0.75 -0.75 0.75 0.75 0.25
    13    14   15    16   17   18   19   20    21
 -2.25 -1.75 0.75 -0.25 0.25 0.25 0.75 2.25 -8.25
    ....
> ltsreg(loss ~ ., data=stack)
    ....
Coefficients:
 Intercept Air.Flow Water.Temp Acid.Conc.
 -36.2921    0.7362   0.3691     0.0081

Scale estimate of residuals:  1.038

Total number of observations:  21

Number of observations that determine the LTS estimate:  13
> plot(ltsreg(loss ~ ., data=stack))
```

Remember that the results are random, but this version of the code does seem to produce different answers.

The plot method gives a normal QQ-plot of the residuals and plots of the standarized residuals against fitted values, index and robust distance from the centre of the x values.

For the `hills` data we can use

```
> summary(ltsreg(time ~ ., data=hills))
    ....
Coefficients:
 Intercept    dist   climb
 -0.9477    4.7817  0.0086

Scale estimate of residuals: 3.033

Robust Multiple R-Squared: 0.9761

Total number of observations:  35

Number of observations that determine the LTS estimate:  19

Residuals:
  Min. 1st Qu. Median 3rd Qu.  Max.
 -11.8 -0.7073 0.6713   5.812 64.21

Weights:
  0  1
 10 25

> par(pty="s", mfrow=c(1,2))
> plot(ltsreg(time ~ ., data=hills), which=4:3)

> hills$ispeed <- hills$time/hills$dist
> hills$grad <- hills$climb/hills$dist
> ltsreg(ispeed ~ grad, data=hills)
Coefficients:
 Intercept   grad
 4.5707    0.0090
```

Two of the diagnostic plots are shown in Figure 8.4.

8.5 Multivariate location and scale

Function `cov.mcd` provides an alternative robust estimator of multivariate location and covariance or correlations developed by Rousseeuw (1984); see also Rousseeuw & Leroy (1987, p.262). MCD stands for 'minimum covariance determinant' and corresponds to taking the covariance of (about) half the points with the smallest determinant rather than with the smallest enclosing volume as in `cov.mve`. In a sense discussed in that paper, `cov.mve` generalizes LMS and `cov.mcd` generalizes LTS.

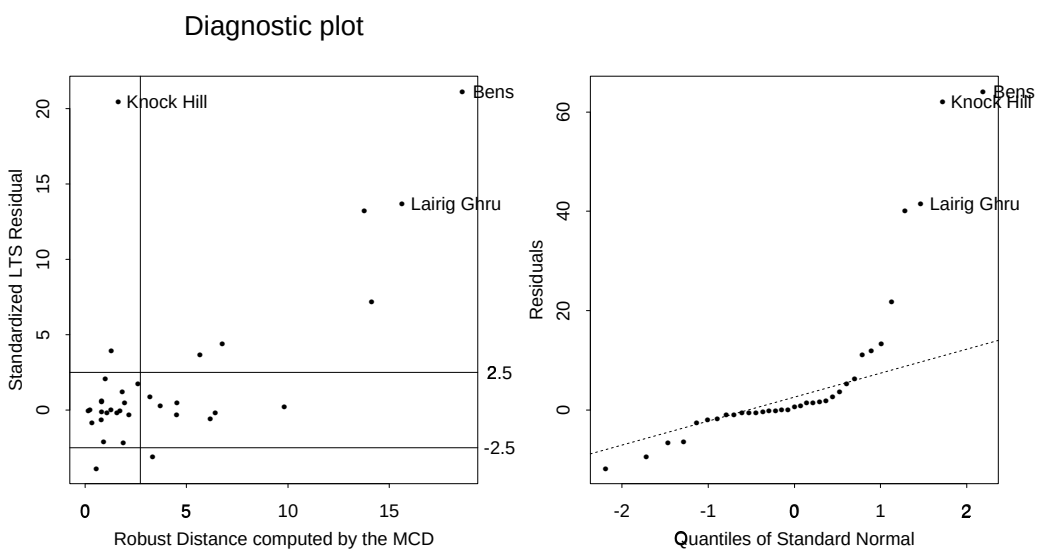

Figure 8.4: Diagnostic plots for an `ltsreg` fit to the `hills` dataset.

Chapter 12

Survival Analysis

S-PLUS 4.5 and S-PLUS 5.0 introduced further functions for survival analysis from a different author: these have a very substantial overlap with those already in S-PLUS but are more general in that they allow *truncation* as well as *censoring*. Either or both censoring and truncation occur when subjects only observed for part of the time axis. An observation T_i is right-censored if it is known only that $T_i > U_i$ for a censoring time U_i, and left-censored if it is only known that that $T_i \leqslant L_i$. (Both left- and right-censoring can occur in a study, but not for the same individual.) Interval censoring is usually taken to refer to subjects known to have an event in $(L_i, U_i]$, but with the time of the event otherwise unknown. Truncation is similar but subtly different. For left and right truncation, subjects with events before L_i or after U_i are not included in the study, and interval truncation refers to both left and right truncation. (Notice the inconsistency with interval censoring.)

We can consider the corresponding contributions to the likelihood. Suppose we have interval truncation on $[0 \leqslant L_i, R_i \leqslant \infty)$. Then the contributions are

(a) $f(t_i)/[F(U_i) - F(L_i)]$ for an observed event at t_i.

(b) $[F(C_i) - F(L_i)]/[F(U_i) - F(L_i)]$ for an event left-censored at C_i), that is known to occur in the observation interval prior to C_i.

(c) $[F(U_i) - F(C_i)]/[F(U_i) - F(L_i)]$ for an event right-censored at C_i but know to occur before the end of the observation interval.

(d) $[F(D_i) - F(C_i)]/[F(U_i) - F(L_i)]$ of an event interval-censored in $(C_i, D_i]$.

Clearly right-censoring with right-truncation or left-censoring with left-truncation will rarely make sense.

12.1 Estimators of survival curves

The new functions use functions `censor` which is almost equivalent to `Surv` but whose output is tailored for the new functions (and is incompatible with that from `Surv`, so the correct function must be used). There is a function `kaplanMeier` which computes Kaplan-Meier estimates of the survival curve in a very similar way to `survfit`. For example, compare

```
> kaplanMeier(censor(time, cens) ~ treat, data=gehan,
  conf.interval="log-log")
treat=6-MP
Number Observed: 21
Number Censored: 12
Confidence Type: log-log
           Survival Std.Err 95% LCL 95% UCL
(-Inf,  6]    1.000   0.000   1.000   1.000
(   6,  7]    0.857   0.076   0.666   0.943
(   7, 10]    0.807   0.087   0.622   0.907
(  10, 13]    0.753   0.096   0.576   0.864
(  13, 16]    0.690   0.107   0.521   0.810
(  16, 22]    0.627   0.114   0.471   0.749
(  22, 23]    0.538   0.128   0.395   0.661
(  23, 35]    0.448   0.135   0.328   0.561

treat=control
Number Observed: 21
Number Censored: 0
Confidence Type: log-log
            Survival Std.Err 95% LCL 95% UCL
(-Inf,   1]    1.000   0.000   1.000   1.000
(   1,   2]    0.905   0.064   0.704   0.972
(   2,   3]    0.810   0.086   0.626   0.909
(   3,   4]    0.762   0.093   0.588   0.870
(   4,   5]    0.667   0.103   0.513   0.781
(   5,   8]    0.571   0.108   0.442   0.682
(   8,  11]    0.381   0.106   0.302   0.459
(  11,  12]    0.286   0.099   0.232   0.342
(  12,  15]    0.190   0.086   0.160   0.223
(  15,  17]    0.143   0.076   0.122   0.165
(  17,  22]    0.095   0.064   0.084   0.108
(  22,  23]    0.048   0.046   0.043   0.052
(  23, Inf)    0.000   0.000      NA      NA
```

There are appears to be no way to plot the results. The functions `qkaplanMeier` estimates quantiles by linear interpolation on the results of a call to `kaplanMeier`.

The advantage of `kaplanMeier` comes with interval-censored data, which it can handle and `survfit` cannot. As a simple example, suppose that the `leuk` data had only be recorded in 4-week periods.

```
mn <- 4 * (leuk$time %/% 4)
kaplanMeier(censor(mn, mn + 4, rep(3, length(mn))) ~ 1)
Number Observed: 33
Number Censored: 33
Confidence Type: log
            Survival Std.Err 95% LCL 95% UCL
(-Inf,   0]    1.000   0.000   1.000   1.000
(   4,   4]    0.818   0.067   0.717   0.933
(   8,   8]    0.636   0.084   0.540   0.750
```

```
(  12,  16]    0.606   0.085   0.513   0.716
(  20,  20]    0.515   0.087   0.434   0.611
(  24,  24]    0.455   0.087   0.384   0.539
(  28,  28]    0.424   0.086   0.358   0.502
(  32,  36]    0.394   0.085   0.333   0.465
(  40,  40]    0.364   0.084   0.309   0.428
(  44,  56]    0.333   0.082   0.284   0.391
(  60,  64]    0.273   0.078   0.234   0.317
(  68, 100]    0.182   0.067   0.159   0.207
( 104, 108]    0.152   0.062   0.134   0.171
( 112, 120]    0.121   0.057   0.108   0.135
( 124, 132]    0.091   0.050   0.082   0.100
( 136, 140]    0.061   0.042   0.056   0.066
( 144, 156]    0.030   0.030   0.029   0.032
( 160, Inf)    0.000   0.000      NA      NA
```

which is not altogether helpful since many of those intervals are empty.

12.2 Parametric models

The new analogue to `survreg` is `censorReg`. This has a longer list of distributions (`"extreme"`, `"weibull"`, `"gaussian"`, `"lognormal"`, `"logistic"`, `"loglogistic"`, `"exponential"`, `"logexponential"`, `"rayleigh"` and `"lograyleigh"`). Remember (page 352) that with `survreg` there is a confusion as to whether the names refer to the distribution of T or of $\log T$. That confusion is even worse here, with what is to `survreg` the `"exponential"` and `"rayleigh"` distributions becoming `"logexponential"` and `"lograyleigh"` (and the details are not documented at all)! Examining the code (in the misleadingly-titled function `make.distribution`) shows that `distribution =`

1. `"weibull"`, `"lognormal"` and `"loglogistic"` give accelerated life models for those distributions.
2. `"logexponential"` and `"lograyleigh"` give accelerated-life models for the exponential and Rayleigh distributions respectively.
3. `"extreme"`, `"gaussian"`, `"logistic"`, `"exponential"` and `"rayleigh"` give additive models for those distributions, that is

$$T \sim \boldsymbol{\beta}^T \boldsymbol{x} + \sigma\,\epsilon$$

known as the identity 'link' in `survreg`.

Let us consider a simple example using `gehan`. We can fit a Weibull model by

```
> options(contrasts=c("contr.treatment", "contr.poly"))
> summary(censorReg(censor(time, cens) ~ treat, gehan))
Call:
```

```
censorReg(formula = censor(time, cens) ~ treat, data = gehan)

Distribution:  Weibull

Standardized Residuals:
             Min   Max
Uncensored 0.046 3.359
  Censored 0.095 1.056

Coefficients:
             Est. Std.Err. 95% LCL 95% UCL z-value   p-value
(Intercept)  3.52    0.252    3.02   4.009   13.96 2.61e-044
      treat -1.27    0.311   -1.88  -0.658   -4.08 4.51e-005

Extreme value distribution: Dispersion (scale) = 0.73219
Observations: 42 Total; 12 Censored
-2*Log-Likelihood: 213
```

If we compare this with the result on page 355,

```
> summary(survreg(Surv(time, cens) ~ treat, gehan))
Call:
survreg(formula = Surv(time, cens) ~ treat, data = gehan)
Deviance Residuals:
   Min    1Q Median    3Q  Max
 -2.06 -1.05 -0.222 0.841 1.51

Coefficients:
            Value Std. Error z value         p
(Intercept)  3.52      0.252   13.96 2.61e-044
      treat -1.27      0.311   -4.08 4.51e-005

Extreme value distribution: Dispersion (scale) = 0.73219
Degrees of Freedom: 42 Total; 39 Residual
-2*Log-Likelihood: 94.1
```

we see the agreement is good apart from the log-likelihoods. We can look at this more precisely:

```
> censorReg(censor(time,cens) ~ treat, gehan)$loglik
[1] -116.41 -106.58
> survreg(Surv(time,cens) ~ treat, gehan)$loglik
[1] -57.671 -47.064
> censorReg(censor(time,cens) ~ treat, gehan,dist="logexp")$loglik
[1] -116.77 -108.52
> survreg(Surv(time,cens) ~ treat, gehan, dist="exp")$loglik
[1] -57.251 -49.009
```

so the increase in log-likelihood over the null model is about the same. Part of the answer is that `survreg` is quoting the log-likelihood regarding $\log T$ as the data, whereas `censorReg` is (more naturally) regarding T as the data. The formula for

the likelihood ((12.1) on page 344) shows that it is a product of terms for censored observations, which are probabilities, and of terms for uncensored observations, which are densities. The latter are affected by the transformation of T, so

$$L(\text{parameters}; (\log T_i)) = L(\text{parameters}; (T_i)) \times \prod_{\delta_i=1} T_i$$

We can check this for the `gehan` data

```
> attach(gehan)
> sum(log(time[cens==1]))
[1] 59.515
```

which is precisely the difference in the log-likelihood for the fitted models, and for the null model for the exponential distribution. In the Weibull case, `survreg` is quoting the null-model log-likelihood at the shape parameter it fits to the full model, which is not statistically meaningful.

The advantages of `censorReg` come from its wider range of options. As noted above, it allows truncation, by specifying a call to `censor` as the `truncation` argument. Distributions can be fitted with a *threshold*, that is a parameter $\gamma > 0$ such that the failure-time model is fitted to $T - \gamma$ (and hence no failures can occur before time γ). It the parameter `threshold = T`, γ is estimated as 90% of the smallest observed failure time; if `threshold = "Linearized-qq"` γ is chosen by optimizing the linearity of a QQ-plot of the fitted response and a Kaplan-Meier estimate of survival.

There is a `plot` method for `censorReg`, which appears to require the syntax

```
gehan.cr <- censorReg(censor(time, cens) ~ factor(treat), gehan)
plot(gehan.cr)
```

This produces up to seven figures, of residuals against fitted values, the square root of the absolute value of the residuals against fitted values, and the response against the fitted values (all of which are useless here as the fitted values are the mean for the appropriate group and so take just two values), Weibull probability plots of the residuals and by group, a so-called *stress plot* (not for a factor variable) and a figure showing probability plots by group for each of six distributions (Weibull, log-normal and log-logistic and the corresponding additive models). This function often fails for correctly specified models.

The plot methods are available separately as functions `probplot.censorReg`, `stressplot.censorReg` and `probplot6.censorReg`. The probability plots for the `gehan` example are shown in Figures 12.13 and 12.14.

To show a stress plot (Figure 12.15) we had to resort to the following manipulations and model.

```
leuk <- leuk
attach(leuk); leuk$lwbc <- log(wbc); detach()
plot(censorReg(censor(time) ~ lwbc, data=leuk))
```

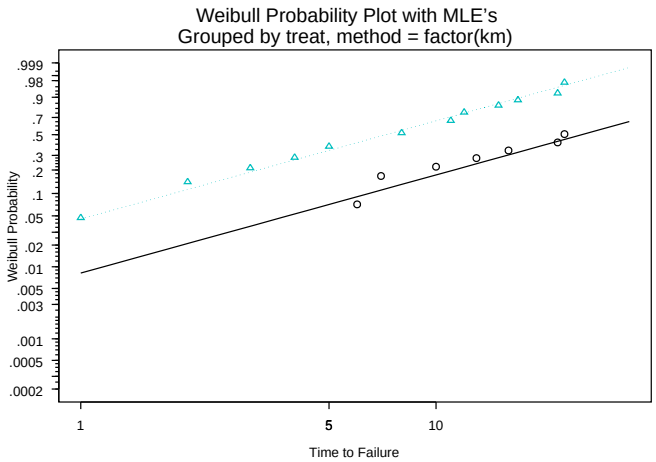

Figure 12.13: Weibull probability plot for `gehan` dataset. The two groups correspond to the two treatments: only the uncensored observations are plotted.

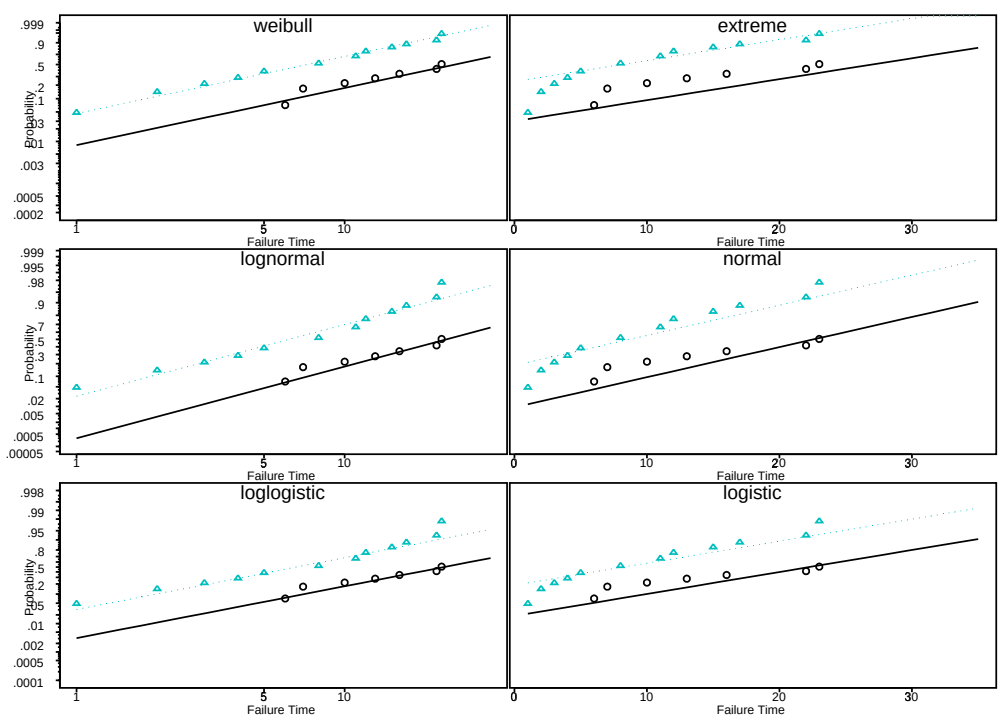

Figure 12.14: Six probability plots for the `gehan` dataset produced by `probplot6.censorReg`. Details as Figure 12.13.

This is a graphical version of the prediction method, showing quantiles against a single numerical covariate (although plotted with x and y axes reversed).

There is a `predict` method that allows prediction of the times at which the probability of an event is as given (by default 10%, 50% and 90%), or the probabilities at specified times.

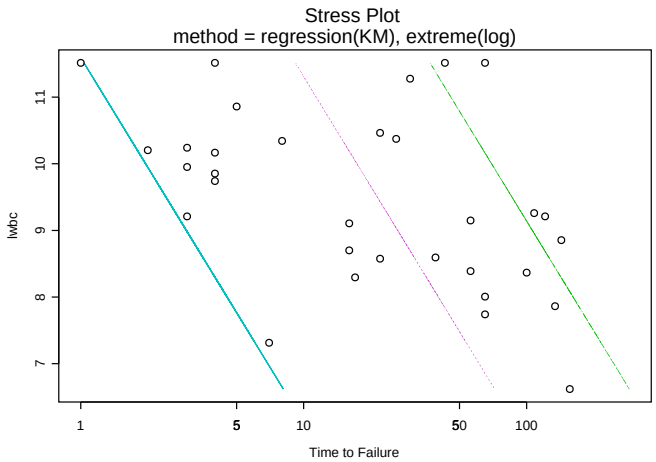

Figure 12.15: 'Stress plot' for a Weibull fit to the `leuk` dataset. The lines correspond to probabilities of 10%, 50% and 90% from left to right.

```
> predict(gehan.cr)
$"factor(treat)=control":
    Estimate Std.Err  95% LCL 95% UCL
0.1   1.8233 0.36576  0.89027  3.7342
0.5   7.2427 0.18407  5.04919 10.3891
0.9  17.4445 0.17041 12.49137 24.3618

$"factor(treat)=6-MP":
    Estimate Std.Err 95% LCL 95% UCL
0.1   6.4752 0.32912  3.3971  12.343
0.5  25.7215 0.24442 15.9310  41.529
0.9  61.9521 0.29598 34.6829 110.662

> predict(gehan.cr, q=seq(10,30,10), type="probability")
$"factor(treat)=control":
   Estimate Std.Err 95% LCL 95% UCL
10  0.65934 0.36507 0.48622 0.79833
20  0.93767 0.72760 0.78327 0.98428
30  0.99200 1.46972 0.87432 0.99955

$"factor(treat)=6-MP":
   Estimate Std.Err  95% LCL 95% UCL
10  0.17366 0.42656 0.083482 0.32654
20  0.38834 0.42417 0.216589 0.59317
30  0.57482 0.50317 0.335227 0.78376
```

Appendix C

Using S-PLUS Libraries

C.2 Creating a library

At least in theory, this process is much simpler than under earlier versions of S-PLUS. You may be able to start at stage 3 with an existing chapter.

1. Create a chapter by `Splus5 CHAPTER filename filename ....`
2. Source the files containing S code, for example by

   ```
   cat *.q | Splus5
   ```

3. Dump the chapter by `Splus5 dumpChapter.`
4. This creates a file `DUMP_FILES` in the chapter directory.
5. Create a directory in the library with the section name, and copy to that directory the files mentioned in `DUMP_FILES` and also any files (`*.[chfr]`, `*.cc`) needed to create compiled code.
6. Move to the library section and run

   ```
   Splus5 CHAPTER
   Splus5 make boot
   ```

Unfortunately, we found that `Splus5 make boot` does not currently work on Linux with files dumped on Solaris, and that no `makefile` is created for a directory with no files to be compiled, when the final stage has to be replaced by (using `sh`)

```
$ Splus5 CHAPTER
$ (BOOTING_S="TRUE" export BOOTING_S; Splus5)
```

Another complication arises if S-PLUS 3.x-style documentation is involved, as that is ignored by the boot process. If you have old-style documentation (`*.d` files), you will need also to distribute those and use

```
mkdir .Data/.Help
foreach help (*.d)
  cp $help .Data/.Help/$help:r
end
```

(`csh`, `tcsh`) or

```
mkdir .Data/.Help
for help in *.d
do
  cp $help .Data/.Help/‘basename $help .d‘
done
```

(`sh`, `ksh`, `bash`) to install them, in 5.0.

For 5.1 the process is slightly different:

```
SHOME=‘Splus5 SHOME‘; export SHOME
mkdir .Data/__Shelp .Data/__Hhelp
for f in *.d
do
  $SHOME/cmd/doc_to_S $f > .Data/__Shelp/‘basename $f‘.sgm
done
```

C.3 Converting libraries written for S-PLUS 3.x

There is an automated way to convert collections of S-PLUS objects for use with S-PLUS 5.x. Suppose you wish to convert library `mylib`

```
$ cd mylib # full path as needed
$ mkdir ../mylib5
$ cd ../mylib5
$ Splus5 CHAPTER
$ cd ..
$ Splus5
> convertOldLibrary("mydata", "mydata5")
> convertOldDoc("mydata", "mydata5") # not 5.0
> q()
```

The converted objects are now in the parallel directory `mylib5`. This process changes the following function calls

from	too
`class`	`oldClass`
`unclass`	`oldUnclass`
`cut`	`oldCut`
`log`	`logb`

and makes some calls to `setOldClass` where objects are generated with multiple classes.

Although safe, some of these conversions may be unnecessary (almost all calls to `log` have one argument), and they will miss the assignment[1] of an old-style

[1] such assignments are still valid, but assignments of more than one class need to be detected to generate a call to `setOldClass`.

class by setting attributes or within a call to `structure`. Also, `oldCut` is not an adequate substitute for `cut` as the argument `include.lowest` is missing.

The alternative is to do the process manually. If there are only simple functions, the simplest process is to use

```
$ cd mylib # full path as needed
$ echo 'data.dump(ls(), "mylib.dmp")' | Splus
$ mkdir ../mylib5
$ cd ../mylib5
$ Splus5 CHAPTER
$ echo 'data.restore("../mylib/mylib.dmp")' | Splus5
```

You can then dump and edit any functions you want to update. The command

```
dumpChapter(meta="regular")
```

will dump all the S objects to `all.S`. It is likely that any `.First.lib` will need to be edited to remove calls to `dyn.load*`.

For efficiency the chapter should be dumped and restored: this writes the S objects in a single file `.Data/__Objects`. If the chapter has a `makefile` (which it will have by default only if it has compiled objects), use[2]

```
$ Splus5 make dump
$ rm -rf .Data
$ Splus5 CHAPTER
$ Splus5 make boot
```

In the absence of `makefile`, use

```
$ Splus5 dumpChapter
$ rm -rf .Data
$ Splus5 CHAPTER
$ (BOOTING_S="TRUE" export BOOTING_S; Splus5)
```

In either case, the documentation will need to be re-installed.

[2] On some Solaris versions this needs a link from `SHOME/cmd/Splus5` to `SHOME/S`.

References

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. New York: Springer. 3, 6, 8, 9, 11, 13, 14, 28, 30, 31

Cleveland, W. S. (1993) *Visualizing Data*. Summit, NJ: Hobart Press. 38

Hsu, J. C. (1996) *Multiple Comparison Procedures: Theory and Methods*. London: Chapman & Hall. 40

Miller, R. G. (1981) *Simultaneous Statistical Inference*. New York: Springer-Verlag. 40

Rousseeuw, P. J. (1984) Least median of squares regression. *Journal of the American Statistical Association* **79**, 871–881. 45

Rousseeuw, P. J. and Leroy, A. M. (1987) *Robust Regression and Outlier Detection*. New York: John Wiley and Sons. 45

Yandell, B. S. (1997) *Practical Data Analysis for Designed Experiments*. London: Chapman & Hall. 40

Yohai, V., Stahel, W. A. and Zamar, R. H. (1991) A procedure for robust estimation and inference in linear regression. In *Directions in Robust Statistics and Diagnostics, Part II*, eds W. A. Stahel and S. W. Weisberg. Springer-Verlag. 42

Index

Entries in `this font` are names of S objects.