Programming 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

17 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/`.

Introduction

These complements are made available on-line to supplement the book

- to explain more technical material that was omitted from the book to reduce its length. Much of this material (in Sections 4.6, 4.9 and C.2) will be of particular interest to those writing and distributing S software.
- to document features that change rapidly between versions of S-PLUS, for example the interfaces with compiled code, and to explain features in new versions of S-PLUS as they appear.

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-PLUS 4.x and for statistical methods available from `http://www.stats.ox.ac.uk/pub/MASS2/`.

Contents

Chapter 3

Graphical Output

3.1 Graphics devices

Greyscale and colour postscript output

The default settings of the Unix graphics device `postscript` will produce postscript which when printed on a colour postscript printer (or previewed on a colour screen) has grey lines or symbols corresponding to `col > 1`, and grey-levels for the fills used in barplots and for `image` plots. Users are often surprised that when these plots are printed on a monochrome printer, the grey lines (but not the fills) become black[1]. This is deliberate, to avoid the poor reproduction of grey lines by half-toning on most low-resolution printers. To ensure that such lines are reproduced in grey, use the undocumented parameter

```
black.and.white="false"
```

in `ps.options` or `ps.options.send`. The same effect can be obtained by editing the file and changing the lines

```
/bw statusdict begin /processcolors where
{pop processcolors} {1} ifelse end 1 eq
 def
```

to

```
/bw false def
```

This conversion also occurs with colour postscript output when proofed on a monochrome printer, and the workaround will ensure the usual conversion of colours to greylevels for both lines and fills.

[1] or white if the grey is very light, lighter than any of the default 'colours'.

Chapter 4

Programming in S

We end these complements on S programming with a section of tips and a checklist of pitfalls starting on page 47.

4.5 Introduction to object orientation

See Section 4.13 below for more details of group method functions (see page 139 of the book).

4.6 Editing, correcting and documenting functions

More on creating help functions

The text outlines the standard procedures under both Unix and Windows. In these complements we give more details of the structure of the help files, and how to produce standard Windows help files.

Structure of help files

Unix help files are `nroff` files, and have a structure like

```
.BG
.FN function/dataset name
.TL
    ....
.KW sysdata
.WR
```

Normally they have a `.d` extension. Check that S terms, in particular those in the `.SA` section (to be cross-referenced), are between left and right single quotes, and that left quotes are not used elsewhere. References should be entered in the form


```
.SH REFERENCES
Roeder, K. (1990) Density estimation with confidence sets
exemplified by superclusters and voids in galaxies.
.ul
Journal of the American Statistical Association
\fB85\fP, 617-624.
.br
Postman, M., Huchra, J. P. and Geller, M. J. (1986)
Probes of large-scale structures in the Corona Borealis region.
.ul
Astrophysical Journal
\fB92\fP, 1238-1247.
```

Other details are best gleaned from examples, for example the system help files (in directories `.Functions/.Help` or `.Datasets/.Help` under the S-PLUS home directory) or our own examples. As the help files may be processed by several variants of `nroff` (for the on-line help), `troff` (to make printed copies) and perhaps conversion scripts, it is best to keep their structure simple.

Keywords

A Unix help file ends with one or more `.KW` lines, then `.WR`. The `.KW` lines denote keywords that are used to organize the information into categories in the `help.start` help system. The list of keywords and their category names are given in the file

```
$SHOME/splus/lib/X11/keywords
```

Some of the more useful keywords are (sorted by the category)

```
aplot           Add to Existing Plot
category        Categorical Data
dplot           Computations Related to Plotting
sysdata         Data Sets
hplot           High-Level Plots
algebra         Linear Algebra
math            Mathematical Operations
mixed           Mixed Effects Models
multivariate    Multivariate Techniques
nonlinear       Non-linear Regression
nonparametric   Nonparametric Statistics
optimize        Optimization
print           Printing
distribution    Probability Distributions and Random Numbers
regression      Regression
tree            Regression and Classification Trees
robust          Robust/Resistant Techniques
smooth          Smoothing Operations
htest           Statistical Inference
models          Statistical Models
survival4       Survival Analysis
```

```
ts              Time Series
utilities       Utilities
```

It is possible to add keywords and associated categories by the command

```
Splus ADDKEYWORD keyword category
```

although use this with care as it will replace any existing entry for the keyword. This does need to be run from the user account owning S-PLUS, so it is helpful to use existing categories as well as new ones.

Creating standard Windows *help files*

The Windows help files used by the system and in our libraries are standard help files, but they were converted automatically from the Unix help files. A set of tools to do this is available in the file `spwinhlp.zip` at

```
http://www.stats.ox.ac.uk/pub/SWin/
http://lib.stat.cmu.edu/DOS/S/SWin/
```

This `zip` archive contains a `PERL` script that can be used on a Unix or Windows 95/NT system to convert a set of `nroff`–style help files to Microsoft's *rich text format*. This can be edited in a Windows word processor if desired. The `.rtf` help file is then compiled to a `.hlp` file using Microsoft's help compiler.[1]

Of course, Windows users will not usually be starting with `nroff`–style files. Our tools provide two routes to make such files. One is a `PERL` script which converts the ASCII files generated by `prompt` to `nroff`–style. Such files will not have the usual enhancements (such as using italic and bold in references) but these can be added by editing the `nroff`–style files or by editing the `.rtf` file in a Windows word processor. Do ensure that S-PLUS quantities are quoted (between left and right single quotes) before conversion.

The second route is a function `uprompt` that works in the same way as `prompt` under Unix, providing a skeleton `nroff` file for completion. This will be the preferred route for experienced writers of help files on Unix (and probably for no one else).

Using standard Windows *help files*

The interface of `help` and `?` to Windows help files in S-PLUS 3.x is not as helpful as it might be, so it is necessary to advise your users how to access help. The simplest way is to advise them to use

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

(obtainable from the URLs in the introduction) when they can use helpfiles in attached library sections without any further actions. (S-PLUS 4.x has adopted our enhancements.) Otherwise they can only access `.hlp` files in sections of the standard library directory, by

[1] This is supplied with most Windows compilers, and an earlier version is also available on the Internet. Details are given in `spwinhlp.zip`.


```
help(fn_name, library="section_name")
```

where `fn_name` is optional and if it is omitted the `.hlp` file is opened at its contents page. Alternatively, the `.hlp` file can be opened by double-clicking or from within a running copy of `winhelp`.

Using help on Unix *machines without* `nroff`

Some Unix platforms, principally SGI, are shipped without the `nroff` program which is used to convert the help files into a readable form for the on-line help system. The system help files can be read, as pre-processed versions are stored in the directory `.Cat.Help` parallel to the `.Help` directory. This workaround can be used for users' help files too, provided that the corresponding entry in the `.Help` both exists and is older than that in `.Cat.help`.

The simplest way to create pre-processed help files is to run

```
Splus CATHELP .Data
```

which processes the files in `.Data/.Help` to `.Data/.Cat.Help`. It is also possible to use a script, for example

```
SHOME=‘Splus SHOME‘; export SHOME
mkdir .Data/.Cat.Help
for f in *.d
do
  nr2a $f > .Data/.Cat.Help/‘basename $f .d‘
done
```

where nr2a is the script

```
#!/bin/sh
sed -e "/^\.ul/d
    /\‘/s/\’//g
    /\‘/s/\‘//g" "$1" |
    nroff -Ttn300 $SHOME/cmd/help.nr -
```

The FSF equivalent of `nroff`, `groff`, can be used with `-Tascii`.

If you have `nroff` or `groff` and the target users might not, consider distributing the `.Cat.Help` directory.

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 7 provides an example.)

- The first argument to `.C` or `.Fortran` is a character string giving the name[2] 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 7.1 on page 197 of Becker, Chambers & Wilks, 1988.)

S storage mode	C	FORTRAN
"logical"	long *	LOGICAL
"integer"	long *	INTEGER
"single"	float *	REAL
"double"	double *	DOUBLE PRECISION
"character"	char **	CHARACTER (*)
"complex"	struct { double re, im;} *	DOUBLE COMPLEX
"list"	void **	

This is the only place in S where a distinction must be made between storage modes `double`, `single` and `integer` all of which correspond to S mode `numeric`, and the creation functions `double`, `single` and `integer` are all specializations of `numeric` used only in this context, to ensure that arguments have the correct storage mode. Functions such as `as.integer` create new objects of the correct mode but strip all attributes and so reduce arrays to vectors, for example. If preserving attributes is important it is necessary to use the assignment form:

```
storage.mode(x) <- "single"
```

[2] 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.

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 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)
  storage.mode(x) <- "double"
```

```
    p <- x
    storage.mode(b) <- "double"
    .C("poly", m, val = p, x, n, b)$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.

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 dim attribute 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. Note that an ANSI C compiler is not used on all Unix platforms, and some of the compilers that are used reject ANSI constructs, especially prototypes. Our solution is to write and test in ANSI C but to distribute source code converted by the GNU utility unprotoize (part of gcc).

Static and dynamic loading

S-PLUS already has many compiled FORTRAN and C functions loaded for use by functions such as svd, qr and eigen. There are several ways of including other functions, such as those from our horner.c, but only two or three of these are available for each platform.

1. *Static loading* (Unix and Windows). This option creates a private copy of S-PLUS for the user which includes all the S-PLUS routines together with any others that may be needed. Since S-PLUS is large (around 6 Mb) this should only be done in special circumstances, such as when a group of users will use the same copy, or when the alternative facilities listed below are not available.

   There is a LOAD facility under Unix. For our function it amounts to running S-PLUS with an initial command line argument

   ```
   $ Splus LOAD horner.c
   ```

   This does several things. First it compiles the external routines using the compiler appropriate to the file extension. Then it creates an executable file called local.Sqpe in the local directory containing all of S together with

   the extra routines. Once this executable is made, when S-PLUS is invoked from that directory it uses this file instead of the system executable file.

   Under S-PLUS for Windows 3.x the command is just LOAD and the executable file is called nsplus.exe. You do need to arrange to run this, by changing the program name in the Target field in the Properties of its shortcut (in Windows 95/NT4; analogously on other systems).

   With S-PLUS for Windows 4.x the command is either LOAD or Splus /LOAD which builds a new version of the S-PLUS DLL called nsqpe.dll. This will automatically be used if it is in the working directory or in SHOME\cmd or anywhere on the path. The path to the DLL to be used may be set explicitly via the environment variable S_ENGINE_LOCATION.

   This method is the least desirable, but sometimes is the only feasible route. Libraries for both C and FORTRAN are needed, which may mean that both compilers need to be available, and it can be important to use precisely the right version of the compilers to ensure that compatible libraries are used. (Using the wrong versions can have unpredictable consequences on existing code in S-PLUS, not just your own extensions.) We have always managed to avoid this solution.

2. *Incremental loading* with dyn.load (Unix). This is possibly the most convenient where available. The dyn.load function dynamically loads object files and makes the functions available to the version of S-PLUS presently running. Each file should be compiled in the normal way (but see below). If more than one file is to be loaded, they can be accumulated into a single relocatable object file using

   ```
   $ ld -r -o objects.o horner.o bessel.o utilities.o
   ```

   (and possibly other flags on some platforms) after which the file may be dynamically loaded within S-PLUS by using

   ```
   dyn.load("objects.o")
   ```

   The result is an invisible character string vector giving the entry points of the routines loaded. (Note: multiple files can be specified directly to dyn.load, but they can not then reference later files in the same load.)

   Libraries can also be specified on the ld command, but on dynamically-loading systems these must be the *static* versions (such as /lib/libm.a). Do *not* include libc.

   This method is available on some Unix S-PLUS systems (in version 3.4, SunOS4, Sun Solaris and IBM RS/6000).

   There is a COMPILE facility that handles most of the compilation details, so our horner.c file was compiled and loaded as follows:

   ```
   > !Splus COMPILE horner.c
   targets= horner.o
   make -f /usr/local/newsplus/newfun/lib/S_makefile  horner.o
   ```

```
cc -c -I${SHOME-`Splus SHOME`}/include -O2   horner.c
> print(dyn.load("horner.o"))
[1] "_poly"   "_horner"
```

Note that this will consult the `Makefile` (or `makefile`) in the current directory, which can be used to set or change the compiler flags. (Beware: despite its name, this is a `make` facility. Thus it will not recompile if `name.o` is newer than its source file, and if both `name.c` and `name.f` are present, `Splus COMPILE name.c` will compile `name.f`!)

3. *Incremental loading* with `dyn.load` (Windows). The `dyn.load` function dynamically loads object files and makes the functions available to the version of S-PLUS presently running. There is a `COMPILE` batch script to interface to the Watcom compilers (and no other can be used), after several environment variables have been set. We use a batch file containing

```
@echo off
set watcom=c:\watcom
set SHOME=c:\packages\splus
path=%path%;%SHOME%\cmd;%watcom%\binnt
set include=%watcom%\h
```

but the details will vary between versions of the compiler. The object produced has extension `.obj`. The batch script can compile more than one source file, but they must either be all C or all FORTRAN. *A warning:* the batch script will not cope with filenames containing spaces, so if S-PLUS 4.x is installed in the default location `C:\Program Files\Splus4`, use the short version of the name[3] for `SHOME`.

There is no analogue of `ld -r`, so objects have to be loaded together on the call to `dyn.load`. Their order will be important, as they are loaded incrementally and only entry points in those routines already loaded can be resolved.

Dynamic loading was introduced in version 3.2; the objects compiled for versions 3.2 and 3.3 are incompatible with versions 4.x and conversely.

Installing the Watcom compilers offers a bewildering array of options. For use with S-PLUS 4.x select as target the NT operating system, and select the libraries with the stack-based calling convention.

4. *Shared libraries* with `dyn.load.shared` (Unix). This is another form of incremental loading that is supported by some Unix systems. (It is the only dynamic method on SGI and DEC Alpha from version 3.3, and is also available for Sun Solaris in version 3.4.) The usage is

```
dyn.load.shared("./shlib.so")
```

where `shlib.so` is a shared library; the argument should be an absolute path. The object (`.o`) files in a shared library must be position-independent

[3] probably `C:\PROGRA~1\SPLUS4`.

which may necessitate special compiler options. This is handled by the `SHLIB` script, so

```
Splus SHLIB -o name.so object1.o [object2.o ...]
```

will compile source code to produce the specified objects and then combine them into a shared library. Thus for the `horner.c` example we could use

```
Splus SHLIB -o horner.so horner.c
```

5. *Augmented loading* with `dyn.load2` (Unix). This is similar to `dyn.load` except that it works by loading the extra routines into central memory, temporarily augmenting the symbol table, recording the locations of the new routines and finally discarding the augmented symbol table. The result is that each time new routines are loaded they cannot connect with previously loaded routines. All routines needed for a task must be loaded in one operation; this is not usually a serious restriction. This method used to be widely available, but it is only in the SunOS and HP versions of S-PLUS 3.4. The advantage of `dyn.load2` over `dyn.load` is that it is easier to use libraries. The disadvantages are that it can be very much slower, take up considerably more memory and it is much less forgiving. However, HP users have no other choice of dynamic method.

`Splus COMPILE` can be used for compilation. It is necesary to tell `dyn.load2` how much space to reserve for the object by its `size` argument: the default (10^5 bytes) is usually sufficient, but if not the warning message will say how much is needed.

6. *Dynamic Link Libraries* (Windows). These are the equivalent under Windows of Unix shared libraries, and are loaded using the S-PLUS function `dll.load`.[4] Although there are standard specifications of DLLs, the details of how these are created and how their entries are called are highly compiler-specific. Beware that there are separate standards for 16-bit and 32-bit DLLs; S-PLUS 3.2 and 3.3 can only load 16-bit DLLs and S-PLUS 4.x is only able to load 32-bit DLLs. Many current versions of compilers can only generate 32-bit DLLs. Section 4.5 of the Programmer's Manual Supplement of the S-PLUS 3.3 for Windows manual set provides the details of interfacing 16-bit DLLs, and we give some examples of generating and using 32-bit DLLs in Section 4.14. A supplementary chapter to the S-PLUS 4.0 Programmer's Guide is available from MathSoft's web site at

`http://www.mathsoft.com/splus/splsprod/update2.htm`

Under S-PLUS 3.x code loaded using `dll.load` has no access to the internal code of S-PLUS, so in particular cannot access the S-PLUS random-number generator. Similarly, any I/O should be done via the Windows API or in the S wrapper function; do not use `printf`. (These restrictions are partially removed in S-PLUS 4.x; see Section 4.14.)

[4] They can be unloaded with `dll.unload`.

Ideally, routines should be dynamically loaded only once, although the latest version loaded will be used[5], which can be useful when developing code. The function `is.loaded` may be used to determine if some particular function has been loaded. The argument to `is.loaded` is the name of a function as a character string, but the name as it is known to the symbol table rather than as it may be known to the programmer. From the display on page 9 it can be seen that the C function `poly` has been given the name `"_poly"` on the system used (SunOS4). The function `symbol.C` may be used to find the symbol table name corresponding to any C function name. Similarly `symbol.For` can be used for FORTRAN subroutines[6]. (It is not safe to assume that FORTRAN names have a trailing underscore nor that they are distinct from C names; they shared a name space on HP systems.) For example consider extending our `polynom` function so that if the C function `poly` is not available the file `horner.o` is dynamically loaded first:

```
polynom <- function(x, b) {
  if(!is.loaded(symbol.C("poly"))) dyn.load("horner.o")
    ....
}
```

Other possibilities are to include the dynamic loading in the session startup function `.First`, and for libraries to use `.First.lib` (see page 52 in Section C.2 of these complements).

Note that if the object file is not in the working directory its full path name must be given on the call to `dyn.load` or `dyn.load2`.

Allocating storage

If the C routine dynamically allocates storage it should use `S_alloc` which is similar to `malloc` except that S-PLUS automatically frees the space afterwards. Note that the call is

```
S_alloc(long n, int size)
```

like `calloc` rather than `malloc`, and like `calloc` the allocated memory is zeroed. There is also `S_realloc` which is similar to `realloc` and again zeroes the allocated storage.

The documentation on *when* S-PLUS frees the space is confused. The Programmer's Guides say:

> The storage they allocate lasts until the current evaluation frame goes away (at the end of the function calling `.C()`) ...

[5] except under Solaris with `dyn.load.shared`; see its on-line help for a workaround.

[6] There may be problems with the case which `symbol.For` does not cover: on most Unix systems symbol table names are in lower case whatever case is used for the subroutine name in the FORTRAN code. Thus on Solaris subroutine `BDRppr` will have symbol table name `bdrppr_`.

but that is not usually correct. The storage will always last until the end of the `.C` call; some `.C` calls do not generate an evaluation frame and so the storage will last until the end of the calling function. However, the rules governing which calls do not generate an evaluation frame are complex and this should not be relied on.

There are also functions `Calloc`, `Realloc` and `Free` to replace the usual un-capitalized versions, which have the two advantages of returning S-style error messages and of casting the pointers to the desired type. Memory allocated by these calls lasts until it is explictly freed by `Free`. `S_realloc` can re-allocate a null pointer, but `Realloc` can not.

Since this area is system-dependent users may need to consult their implementation literature for further details[7].

Calling functions from C code

It is possible to call routines within S-PLUS to report warnings and errors; some simple examples are given in our library `nnet`. The S-PLUS random-number generator can be called from C, as in the point process functions in our library `spatial`. (These facilities are not available if `dll.load` is used with S-PLUS 3.x under Windows.) The routines we know of are

`char *S_alloc(long n, int size)`	allocate `n` items of requested size and set them to zero.
`char *S_realloc(char *p, long new, long old, int size)`	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()`	one uniform random number.
`double norm_rand()`	one standard normal variate.
`setseed(long *seed)`	as in `set.seed`.
`seed_in(long *iseed)`	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`.
`seed_out((long*)NULL)`	write the seed back out to `.Random.seed` after the random variates have been generated.

In addition, there are macros

```
PROBLEM ... RECOVER
PROBLEM ... WARNING
```

[7] We are grateful to Bill Dunlap of MathSoft DAPD for clarifying many of the details for us.

for error reporting discussed on page 17.

Declarations for these and other functions available to the C programmer are given in the include file `S.h` which resides in the `include` subdirectory of the `SHOME` directory. Using `Splus COMPILE` or `Splus SHLIB` will ensure that the correct include files are found; alternatively can use the form of `cc` function in the example on page 9.

There is a problem with the use of the usual I/O using `stdin`, `stdout` and `stderr` on S-PLUS 4.x under Windows. To work around this portably, include the following in your C code

```
#include <S.h>
#if defined(SPLUS_VERSION) && SPLUS_VERSION >= 4000
#  include <newredef.h>
#endif
```

Without this, the I/O is delayed or lost. Further, `scanf` cannot be used; use `fgets` and `sscanf` instead.

Calling S from C

As well as calling C routines from S it is also possible to do the inverse operation of calling an S function from a C routine with the C routine `call_S` provided by S. This is necessary, for example, in a routine for numerical integration where the integrand may be an S function. Details and an example are given in the on-line help and a fuller discussion in Becker *et al.* (1988, §7.2.4). Note that it is only possible to use `call_S` to call S functions from within C functions called from S, and that the process may be extremely slow; it is often more convenient to generate a look-up table from the S function and pass that to the C or FORTRAN function. (This was tried for `surf.gls` in our library `spatial`; the look-up table code was several times faster and so was adopted.)

The arguments for `call_S` are

```
void call_S(void *func, long nargs, void **arguments,
  char **modes, long *lengths, char **names,
  long nres, void **results);
```

Here `func` is the S code that is passed to C as a pointer to a list, the next five arguments give the number of arguments of the S function, and pointers to their (vector) values, modes, lengths and (optionally) names. The variable `nres` describes the number of components of the result, and `results` is an array of pointers that the `call_S` inteface sets to storage where the results have been allocated.

An example will be helpful. The essential part of the C code for computing covariances in our `spatial` library was


```
static void *Scovmod;
static double eps;

void VR_cmset(float *ineps, void **Scode)
{
  eps = *ineps;
  Scovmod = *Scode;
}

static void cov(long n, double *d, long pred)
{
  int i;
  char *modes[] = {"double"};
  long lengths[1];
  void *arguments[1], *results[1];

  for (i = 0; i < n; ++i) d[i] = max(sqrt(d[i]), pred*eps);
  *arguments = (void *)d; lengths[0] = n;
  call_S(Scovmod, 1L, arguments, modes, lengths,
         (char **)NULL, 1L, results);
  for (i = 0; i < n; ++i) d[i] = ((double *)(*results))[i];
}
```

and the main calling S code is

```
covmod <- covmod
args <- list(...)
if(length(args)) {
  pm <- pmatch(names(args), names(covmod), nomatch=0)
  if(any(pm == 0)) warning(paste("some of ... do not match"))
  covmod[pm] <- unlist(args)
}
defn <- c("function(x){covmod <-",deparse(covmod),"covmod(x)}")
krcov <- eval(parse(text=defn))
.C("VR_cmset", as.single(eps), list(krcov))
```

This constructs a function `krcov` which takes a single argument and returns a vector of the same length. The specified covariance function is included as a local function within `krcov` to ensure that it is fully specified when stored. The second argument of the call to `VR_cmset` then passes a pointer to this function to the C code which is used to stored the code's address in `Scovmod`.

There *are* ways to call S-PLUS from a standalone C program under Windows. Versions 3.3 and later of S-PLUS for Windows are DDE (Dynamic Data Exchange) servers (unless disabled), and so the compiled C program can start and communicate with a running S-PLUS process by using DDE messages; some details are given in the Programmer's Manual Supplement of the S-PLUS 3.3 for Windows manual set and the S-PLUS 4 Programmer's Guide. In S-PLUS 4.x for Windows the S-PLUS engine is implemented as a DLL, so an ambitious programmer could write an interface[8] to that DLL.

[8] although to our knowledge the interface is not publicly documented.

FORTRAN input/output

S-PLUS does not use FORTRAN input/output routines, and these may not work correctly if included in your FORTRAN code. Instead, the output functions DBLEPR, INTPR and REALPR are provided. These all have the form

```
SUBROUTINE name(LABEL,NCHAR,DATA,NDATA)
```

where LABEL is a character string used for the printout, NCHAR is the length of the label (or zero for no label) and there are NDATA items to be printed from array DATA. The names correspond to the FORTRAN data types of DOUBLE PRECISION, INTEGER and REAL. These functions often suffice, especially for debugging. If elaborate layouts are needed, information can be written to a character string, and any of these routines used to print the string as a label.

A further problem arises with libraries which may call FORTRAN output for error messages. Even if it is possible to ensure that the messages are never produced (say by setting a flag), the routines will still be referenced in the object module. With dyn.load this is no problem, as setting its argument undefined to a positive value allows missing external code to be ignored. However, dyn.load2 is not as forgiving! It will normally search the FORTRAN libraries, and include the FORTRAN output routines. Unfortunately, these will often include names which conflict with the C routines included in S-PLUS, and dyn.load2 will then refuse to proceed. *In extremis* dummy routines may have to be written in C to replace the unused FORTRAN routines.

Consider the following example, under S-PLUS 3.4 on SunOS4. First the file test.f:

```
      subroutine test(x, ifail)
      real x
      if(ifail .ge. 0) write(*, 1000) x
      return
1000  format(" X is ",f6.3)
      end
```

and the S-PLUS session:

```
> testf <- function(x, ifail)
    invisible(.Fortran("test", as.single(x), as.integer(ifail)))
> dyn.load("test.o",1)
Warning messages:
1: No definition for symbol "_s_wsFe" in: dyn.load("test.o",1)
2: No definition for symbol "_do_f_out" in: dyn.load("test.o",1)
3: No definition for symbol "_e_wsfe" in: dyn.load("test.o",1)
> testf(13.7, -1)
```

Using dyn.load2 fails, but writing dummy.c:

```
#include <stdio.h>
s_wsFe() {fprintf(stderr, "called dummy Fortran I/O\n");}
do_f_out() {fprintf(stderr, "called dummy Fortran I/O\n");}
e_wsfe() {fprintf(stderr, "called dummy Fortran I/O\n");}
```


and compiling it saves the day:

```
> dyn.load2(c("test.o", "dummy.o"))
> testf(13.7, -1)
```

It is wise to put warning messages in the dummy routines, in case they *are* called, and indeed this is a good idea for production code using dyn.load or dyn.load2.

Handling errors

A number of authors have left calls to exit (C) or STOP (FORTRAN) in their compiled code. This is anti-social, as such calls will cause S-PLUS to terminate, not just the author's code. Fortunately, more elegant ways have been provided (and documented).

From C

The mechanisms

```
#include <S.h>
PROBLEM "warning message" WARNING(NULL_ENTRY);
PROBLEM "error message" RECOVER(NULL_ENTRY);
```

can be used to emulate calls to warn and stop respectively. The unusual syntax covers macro calls to sprintf, so the string can be replaced by any set of arguments to that function, separated by commas. Thus we could use

```
PROBLEM "#params (%d) is too large",nparams RECOVER(NULL_ENTRY);
```

To use these with S-PLUS 4.0 release 3 or later, add the compiler flag

```
-DS_ENGINE_BUILD
```

when compiling for use with dynamic or static loading (but not DLL loading).

From FORTRAN

There is a function XERROR called as

```
CALL XERROR('msg', LEN('msg'), errno, errlev)
```

where errlev is the error level (2 for a fatal error, 0 for a warning, -1 for a warning to be given just once) and errno is a strictly positive identifier for that error message, unique within that subroutine. See the help page for further details and the similar function XERRWV.

Missing and special values

Internally S-PLUS works with missing values (`NA`) and the special values of IEEE floating point arithmetic (`Inf` and `NaN`, the latter denoting an indefinite result like $0./0.$). By default an error is generated if either occur in an argument passed to `.C` or `.Fortran`.

When C or FORTRAN code is being used for speed, it may be important to handle the missing and special values in the compiled code. The arguments `NAOK` and `specialsok` to `.C` or `.Fortran` can be set to true, in which case the S calling routine performs no checking, so the user's code must check *all* the arguments.

S-PLUS provides a set of C macros to test and set missing and special values. These include

```
is_na(x, mode)   is_nan(x, mode)         na_set(x, mode)
is_inf(x, mode)  inf_set(x, mode, sign)  na_set3(x, mode, type)
```

Here `x` is a *pointer* to a numeric type, and mode is one of the symbolic constants `INT`, `REAL`, `DOUBLE`, `COMPLEX`, `CHAR` or `LGL` (for logical). Note that `NaN` is treated as a type of missing value. The function `is_na` returns 0, `Is_NA` or `Is_NaN`, and the `type` argument to `na_set3` should be one of these two symbolic constants. IEEE infinite values are signed, so `is_inf` returns 0 or ± 1.

No facilities are provided for handling missing values in FORTRAN code.

As an example, consider rewriting the function `dist`. This operates on an $n \times p$ matrix, and for each pair of rows i, j computes the distance omitting columns in which either row has a missing value, and rescaling appropriately. The S function is

```
ourdist <- function(x)
{
   n <- nrow(x)
   res <- .C("ourdist",  as.double(x), as.integer(n),
               as.integer(ncol(x)),  res = double(n*(n-1)/2),
               NAOK = T)$res
   attr(res, "Size") <- n
   res
}
```

and the (unoptimized) C code is

```
#include <S.h>
#include <math.h>

void  ourdist(double *x, long *nin, long *pin, double *res)
{
   int     i, j, k, den, n = *nin, p = *pin;
   double  item, tmp;

   for (i = 0; i < n-1; i++)
```



```
      for (j = i+1; j < n; j++) {
         den = 0; tmp = 0.0;
         for (k = 0; k < p; k++) {
            if (!is_na(x + i + n * k, DOUBLE) &&
                !is_na(x + j + n * k, DOUBLE)) {
               den++;
               item = x[i + n * k] - x[j + n * k];
               tmp += item * item;
            }
         }
         if (!den) na_set3(&tmp, DOUBLE, Is_NaN);
         else tmp = sqrt(tmp * p / den);
         *res++ = tmp;
      }
}
```

Calling FORTRAN from C

It is quite common to want to call FORTRAN routines from C routines to be compiled into S-PLUS; perhaps the commonest use is to call numerical analysis routines either in separately compiled FORTRAN code or those already included in the S-PLUS executable. Most package writers just call the FORTRAN name with a trailing underscore; this works on most Unix machines (but not on HP systems) but not on Windows. The portable approach is to use a set of macros defined by including `S.h` in the C code.

Here is an example from porting the `polyclass` library to Windows. The original C code was

```
int lusolinv(a,n,b,k)
int n,k;
double **a,*b;
/* various lu things
   k=0 inverse, non symmetric
   k=1 inverse, symmetric
   k=2 solve, symmetric */
{
   double aa[MAXSPACE+5][MAXSPACE+5],bb[MAXSPACE+5],det[2],inert[3];
   int kpvt[MAXSPACE+5],info,i,j;
   if(k<2) for(i=0;i<n;i++) for(j=0;j<n;j++)aa[i][j]=a[j][i];
   else for(i=0;i<n;i++){
      for(j=0;j<n;j++)aa[i][j]=a[j][i];
      bb[i]=b[i];
   }
   i=MAXSPACE+5;
   j=1;
   if(k==0){
      xdgefa_(aa,&i,&n,kpvt,&info);
      xdgedi_(aa,&i,&n,kpvt,det,bb,&j);
      for(i=0;i<n;i++) for(j=0;j<n;j++)a[i][j]=aa[j][i];
```

```
   }
   if(k==1){
      xdsifa_(aa,&i,&n,kpvt,&info);
      xdsidi_(aa,&i,&n,kpvt,det,inert,bb,&j);
      for(i=0;i<n;i++) for(j=i;j<n;j++)a[i][j]=aa[j][i];
      for(i=0;i<n;i++) for(j=0;j<i;j++)a[i][j]=aa[i][j];
   }
   if(k==2){
      xdsifa_(aa,&i,&n,kpvt,&info);
      if(info!=0)return 0;
      xdsisl_(aa,&i,&n,kpvt,bb);
      for(i=0;i<n;i++)b[i]=bb[i];
   }
   return 1;
}
```

The portable form becomes

```
#include "S.h"

void F77_NAME(xdgefa)(double*, int*, int*, int*, int*);
void F77_NAME(xdgedi)(double*, int*, int*, int*,
                      double*, double*, int*);
void F77_NAME(xdsifa)(double*, int*, int*, int*, int*);
void F77_NAME(xdsisl)(double*, int*, int*, int*, double*);
void F77_NAME(xdsidi)(double*, int*, int*, int*, double*,
                      int*, double*, int*);
int lusolinv(a,n,b,k)
int n,k;
double **a,*b;
/* various lu things
   k=0 inverse, non symmetric
   k=1 inverse, symmetric
   k=2 solve, symmetric */
{
   double aa[MAXSPACE+5][MAXSPACE+5],bb[MAXSPACE+5],det[2];
   int kpvt[MAXSPACE+5],info,i,j,inert[3];
/* inert was wrongly declared as double */
   if(k<2) for(i=0;i<n;i++) for(j=0;j<n;j++)aa[i][j]=a[j][i];
   else for(i=0;i<n;i++){
      for(j=0;j<n;j++)aa[i][j]=a[j][i];
      bb[i]=b[i];
   }
   i=MAXSPACE+5;
   j=1;
   if(k==0){
      F77_CALL(xdgefa)(aa,&i,&n,kpvt,&info);
      F77_CALL(xdgedi)(aa,&i,&n,kpvt,det,bb,&j);
      for(i=0;i<n;i++) for(j=0;j<n;j++)a[i][j]=aa[j][i];
   }
   if(k==1){
```



```
      F77_CALL(xdsifa)(aa,&i,&n,kpvt,&info);
      F77_CALL(xdsidi)(aa,&i,&n,kpvt,det,inert,bb,&j);
      for(i=0;i<n;i++) for(j=i;j<n;j++)a[i][j]=aa[j][i];
      for(i=0;i<n;i++) for(j=0;j<i;j++)a[i][j]=aa[i][j];
   }
   if(k==2){
      F77_CALL(xdsifa)(aa,&i,&n,kpvt,&info);
      if(info!=0)return 0;
      F77_CALL(xdsisl)(aa,&i,&n,kpvt,bb);
      for(i=0;i<n;i++)b[i]=bb[i];
   }
   return 1;
}
```

The prototypes for the forward declaration were found by examining the FORTRAN code; they showed an error in the C code[9]. There are similar macros `F77_COMDECL` and `F77_COM` to declare and use a FORTRAN common block. To write a C routine which is callable from FORTRAN use the macro `F77_SUB` for the name: this will append an underscore on systems where this is needed.

The curious reader can find the definitions of these macros in the file `system.h` in the `include` subdirectory of the main S-PLUS directory; the machine-specific definitions are in `cdefs.h`. For S-PLUS 3.x under Windows the compilers add a trailing underscore in C but not FORTRAN, and `F77_NAME` and `F77_COMDECL` use a `Fortran` keyword to suppress the underscore.

4.12 Object orientation: an extended statistical example

Model formulae have become much more widely used in S-PLUS in recent years. We will explore the paradigms of using model formulae via an improved `lda` function. We do so by making the function generic, and renaming the workhorse function as `lda.default`. Thus we have

```
lda <- function(x, ...)
{
  if(is.null(class(x))) class(x) <- data.class(x)
  UseMethod("lda", x, ...)
}
lda.default <- function(x, grouping, prior = proportions,
  tol = 0.0001, method = c("moment", "mle", "mve", "t"),
  nu = 5, ...)
    ....
```

[9] As an unrelated problem this code caused S-PLUS 4.x rel 2 to crash when run from the GUI, although it worked from the `Sqpe` command line. This was eventually traced to the automatic allocation of the array `aa`: `MAXSPACE` is 100, and so this array is of size 86Kb. It was declared to be static; objects of this size cannot be allocated on the stack in S-PLUS 4.x releases 1 and 2.

(The full sources of these functions are in library `MASS`.)

When working with model formulae it is helpful to store the function call as component `call` of the output list. There is a standard way to do this using the function `match.call`, so the return value of `lda.default` becomes

```
structure(list(prior = prior, counts = counts, means =
               group.means, scaling = scaling, lev = levels(g),
               svd = X.s$d[1:rank], N = n, call = match.call()),
          class = "lda")
```

Once this is done the function `update` can be used to update the object just as for linear models. It is also useful to have a record in the object of how it was created. Note that it is not possible to record the call of the generic function, as `UseMethod` does not return but merely implements method dispatch.

We allow the first argument of `lda` to be either a matrix-like object (such as a data frame or Matrix) or a formula. Since matrices do not necessarily have a class assigned, our generic function `lda` assigns a class via the function `data.class`. The method despatch mechanism will then use the functions `lda.matrix`, `lda.data.frame` and `lda.formula` as appropriate.

The function `lda.data.frame` is quite simple, as it calls the matrix method and then fixes up the recorded call

```
lda.data.frame <- function(x, ...)
{
  x <- structure(data.matrix(x), class = "matrix")
  res <- lda.matrix(x, ...)
  res$call <- match.call()
  res
}
```

It is tempting to use `NextMethod`, but this calls the *second* method, here the default method. The method function for Matrices is similar.

We could do without an explicit `lda.matrix` and rely on `lda.default`, but we chose to add `subset` and `na.action` arguments for matrices.

```
lda.matrix <- function(x, grouping, ...,
                       subset, na.action = na.fail)
{
  if(!missing(subset)) {
    x <- x[subset,  , drop = F]
    grouping <- grouping[subset]
  }
  if(!missing(na.action)) {
    dfr <- na.action(structure(list(g = grouping, x = x),
                               class = "data.frame"))
    grouping <- dfr$g
    x <- dfr$x
  }
  res <- NextMethod("lda")
```



```
  res$call <- match.call()
  res
}
```

After these preliminaries we can write the method for model formulae. How would we wish to interpret a model formula? Our idea is to follow the closely related logistic discrimination, so the left-hand side should be a factor giving which group the case belongs to, and the right-hand side should give a vector of explanatory variables. We do *not* want an intercept term (LDA centres groups on the group mean), so any intercept term (usually implicit) should be ignored. It would not be usual to expand factors or to allow interactions, but there seems no reason to restrict a knowledgeable user. It should come as no surprise that we construct a matrix and grouping factor, call the default method and adjust the recorded call.

```
lda.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$... <- NULL
  m[[1]] <- as.name("model.frame")
  m <- eval(m, sys.parent())
  Terms <- attr(m, "terms")
  grouping <- model.extract(m, "response")
  x <- model.matrix(Terms, m)
  xint <- match("(Intercept)", dimnames(x)[[2]], nomatch=0)
  if(xint > 0) x <- x[, -xint, drop=F]
  res <- lda.default(x, grouping, ...)
  res$terms <- Terms
  res$call <- match.call()
  class(res) <- "lda"
  res
}
```

The first steps are to match the call, without expanding the `...` term, and then to ensure that the `data` argument is a list or data frame, not a matrix. We then call `model.frame` with the essential subset of the arguments of the original call, setting all arguments that do not apply to `model.frame` to `NULL`. (At this level of abstraction all the arguments specific to `lda` are still in the `...` term.) The call to `model.frame.default` returns a data frame with columns corresponding to the response, the main effects and any extra variables such as weights, handling `na.action` and `subset`. (Other methods for `model.frame` can be written, and indeed `tree`, `loess`, `coxph` and even `lm` have their own methods.) If factors are not anticipated, it is simpler to remove the intercept in the formula by

```
attr(Terms, "intercept") <- 0
x <- model.matrix(Terms, m)
```

but this does not do the right thing when factors *are* used.

For the `predict` method we have to be able to extract the data matrix `x` from a new data frame. We made life easy by storing the `terms` attribute in the `lda` object, so we can use

```
predict.lda <-
function(object, newdata, prior = object$prior, dimen, ...)
    ....
  if(!is.null(Terms <- object$terms)) { #
    # formula fit
    if(missing(newdata)) newdata <- model.frame.lm(object)
    else
      newdata <- model.frame(as.formula(delete.response(Terms)),
                       newdata, na.action=na.pass)
    x <- model.matrix(delete.response(Terms), 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)) {
      xname <- eval(substitute(object$call$x))
      if(!is.null(sub <- object$call$subset))
        newdata <- eval(parse(text=paste(deparse(xname),"[",
                      deparse(sub),",]")), sys.parent())
      else
        newdata <- eval(parse(text=deparse(xname)), sys.parent())
      if(!is.null(nas <- object$call$na.action))
        newdata <- do.call(nas, list(newdata))
    }
    if(is.null(dim(newdata)))    # a row vector
      dim(newdata) <- c(1, length(newdata))
    x <- as.matrix(newdata)  # to cope with data frames
  }
  if(dim(x)[2] != dim(object$means)[2])
    stop("wrong number of variables")
    ....
```

This has to handle both cases in which `newdata` is a new matrix-like object and those in which it is a data frame from which the matrix is to be extracted. We allow the first possibility only if the original call specified a matrix-like object and the second only if a formula was given. The purpose of the rest of the code is to attempt to retrieve the original data in the parent frame if `newdata` is not specified.

Some of the details of `predict.lda` may not be obvious. We must not assume that the object was created by `lda`, so we identify formula-based objects by the presence of the `terms` component. Asking for

```
model.matrix(Terms, newdata)
```


will cause an error unless the left-hand side of the formula can be matched; `delete.response` works around this. The rest of the code ensures that the right data frame is constructed and that missing values are handled sensibly.

In the case of a matrix-based object we illustrate another approach. Suppose the object `cr.lda` had call

```
lda.data.frame(x = lcrabs, crabs.grp)
```

Then `substitute(object$call$formula)` evaluates to `cr.lda$call$x` and `xname` is `lcrabs`. The next lines of code simulate typing `newdata <- lcrabs` or `newdata <- lcrabs[subset, ]` in the parent frame. This is rather tricky; the original specification might have been an arbitrary S expression (we use `log(ir)` in Section 13.3). Thus we need to parse the expression before evaluation.

More elaborate routes for predict methods are illustrated by the functions `predict.lm`, `predict.factanal` and `predict.tree`, which allow more general objects as `newdata`.

It was not necessary to use the generic function mechanism to use model formulae with `lda`; the function `princomp` shows a different approach. Note though that the first argument needs to be named `formula` for `update` to be fully operational.

There is an apparently undocumented trap in writing method functions. The first argument of the method function must have the same name as the first argument of the generic function; thus the first argument of all `print` methods should be `x`, and of all `summary`, `predict`, `coef`, ... methods should be `object`. Otherwise the first argument will be evaluated twice, once by the generic function and once by the method function. If the first argument is a call to a fitting function, the (possibly time-consuming) fit would be performed twice.

It is good practice for a method function to have a `...` argument if the generic function does (and `predict` does). This can then 'mop up' arguments that are only relevant to other methods but might appear in the current call if our method has been called from another by `NextMethod` or will call another by `NextMethod`. (Unfortunately there are system examples such as `predict.lm` where this advice has not been followed.) It is also good practice for a generic function to have a `...` argument so as not to restrict future method functions.

4.13 Group method functions

The unary and binary operators such as `+`, `-` and `>` are functions like any other. An expression such as `x + 3` is merely a shorthand for `"+"(x, 3)` which is a also legitimate usage.

The operators are generic functions, so methods may be written for them. They are special in that method dispatch is determined by the class of both arguments. If either argument has a specified class the other argument must either have the same class or no class. In this section we illustrate how to write a new methods for a class of polynomials.

Polynomial objects

We will illustrate methods for the operators by constructing a class for polynomials of one variable, which we will take to be "x". An object in the class will be represented by a vector of numeric coefficients with a class attribute. Polynomials will be written in 'power series' form, that is with the powers in increasing order, and the leading (last) coefficient will be guaranteed to be non-zero unless the polynomial itself is identically zero.

A constructor function for polynomial objects is

```
polynomial <- function(a = c(0, 1))
{
  a <- as.numeric(a)
  while((la <- length(a)) > 1 && a[la] == 0) a <- a[-la]
  structure(a, class = "polynomial")
}
```

The first step coerces `a` to numeric, but just as importantly strips off other attributes such as `dim`, `dimnames` or `names` which if present might cause problems later. The next step ensures the leading coefficient is non-zero. Note that zero coefficients of lower degree are not omitted so the degree of the polynomial is always one less than the length of the coefficient vector.

With any class it is conventional and often useful to have two other functions defined, a *predicate* (`is.xxxx`) function and a *coercion* (`as.xxxx`) function. For polynomials these are very simple

```
  is.polynomial <- function(p) inherits(p, "polynomial")

  as.polynomial <- function(p)
      if(is.polynomial(p)) p else polynomial(p)
```

Unary and binary operations

Methods for the unary and binary operators may be written the same way any method functions are written. For example addition may be allowed by writing a method function such as

```
"+.polynomial" <- function(e1, e2)
{
  if(missing(e2)) return(e1)
  e1 <- c(unclass(e1), rep(0, max(0, length(e2) - length(e1))))
  e2 <- c(unclass(e2), rep(0, max(0, length(e1) - length(e2))))
  polynomial(e1 + e2)
}
```

Notice that unary form of operators have the second argument missing rather than the first.

The subtraction method could be almost identical but it is preferable to write a single method function for the *group generic function* `Ops`. The method dispatch

mechanism will then call the function `Ops.polynomial` when a polynomial method is needed for any operator, supplying the name of the operator that initiated the dispatch as a local frame variable, `.Generic`. Group method functions usually have a `switch` statement using the `.Generic` variable to separate the specific actions required.

Table 4.5 lists all the group generic functions, together with the component functions that initiate method dispatch to each of them. Note that `%*%` is a generic binary operator but not a member of a group and `%o%` is not generic.

Group	Component functions
`Ops`	`+, -, *, /, ^, %/%, %%, <, <=, >, >=, ==, !=, &, |, !`
`Math`	`abs, acos, acosh, asin, asinh, atan, atanh, ceiling, cos, cosh, cumsum, exp, floor, gamma, lgamma, log, log10, round, signif, sin, sinh, tan, tanh, trunc`
`Summary`	`all, any, max, min, prod, range, sum`

Table 4.5: The group generic functions and their components.

Note that the three group generic functions `Ops`, `Math` and `Summary` are *not* visible objects in S-PLUS but may only be called indirectly through their component functions. Method functions written for them, such as our `Ops.polynomial` below, are necessarily visible but are also never called directly by the user.

Figure 4.3 gives a group method function for the binary operators which are useful for polynomials. For simplicity division is supported only by scalars. Powering is only allowed for non-negative integer exponents and is somewhat inefficiently implemented. Amongst the logical operators only exact equality and inequality are supported.

The most useful thing for a `Summary` method to do is to signal that the user has probably made a slip.

```
Summary.polynomial <- function(p)
{
    stop(paste(.Generic, "invalid for polynomials"))
}
```

Methods for functions in the `Math` group like `round` and `signif` are occasionally needed to manipulate the coefficients, but most others are probably best disallowed.

```
Math.polynomial <- function(p, digits)
{
  switch(.Generic,
         round = , signif = , floor = , ceiling = ,
         trunc = polynomial(NextMethod(.Generic)),
         stop(paste(.Generic, "unsupported for polynomials")))
}
```

```
Ops.polynomial <- function(e1, e2)
{
  if(missing(e2))
    return(switch(.Generic,
                  "+" = e1,
                  "-" = polynomial(NextMethod(.Generic)),
                  stop("unsupported unary operation")))
  e1 <- unclass(e1)
  e2 <- unclass(e2)
  l1 <- length(e1)
  l2 <- length(e2)
  e1.op.e2 <-
    switch(.Generic,
           "+" = , "-" = {
             e1 <- c(e1, rep(0, max(0, l2 - l1)))
             e2 <- c(e2, rep(0, max(0, l1 - l2)))
             NextMethod(.Generic)
           },
           "*" = if(l1 == 1 || l2 == 1) e1 * e2 else {
             m <- outer(e1, e2)
             as.vector(tapply(m, row(m) + col(m), sum))
           },
           "/" = {
             if(l2 > 1 || e2 == 0)
               stop("unsupported polynomial division")
             e1/e2
           },
           "^" = {
             if(e2 < 0 || e2 %% 1 != 0)
               stop("unsupported polynomial power")
             switch(as.character(e2),
                    "0" = 1,
                    "1" = e1,
                    {
                      p <- q <- polynomial(e1)
                      for(i in 2:e2)
                        p <- p * q
                      as.numeric(p)
                    })
           },
           "==" = return(l1 == l2 && all(e1 == e2)),
           "!=" = return(l1 != l2 || any(e1 != e2)),
           stop("unsupported operation on polynomials"))
  polynomial(e1.op.e2)
}
```

Figure 4.3: Group method for unary and binary operations with polynomials.

Coercion to character and printing

At least two character forms are important for polynomials: a 'power series' form for display and a nested product, or 'Horner' form for parsing into an expression for the polynomial that may later be evaluated. Casting a polynomial into the horner form is simple and a function to do it is

```
horner <- function(p)
{
  a <- as.character(rev(unclass(p)))
  h <- a[1]
  while(length(a <- a[-1]) > 0) {
    h <- paste("x*(", h, ")", sep = "")
    if(a[1] != 0)
      h <- paste(a[1], " + ", h, sep = "")
  }
  h
}
```

Notice that zero coefficients are omitted.

```
> x <- polynomial()
> p <- (x^2 + 1)^2
> horner(p)
[1] "1 + x*(x*(2 + x*(x*(1))))"
```

The function `as.character`, like most of the system coercion functions, is generic but uses an `.Internal` reference rather than a call to `UseMethod`. This places some restrictions on its use, particularly with functions like `lapply` where *method dispatch will not occur if the generic function name is used as the function to be applied*[10], but the rules for writing methods for them are the same.

The `horner` function could be used as a polynomial method function for `as.character`, but a power series representation is probably a better way to define the coercion for general purposes. The following method goes to some trouble to cast the polynomial in a conventional form and provides a result that may also be parsed into a valid S-PLUS expression.

```
as.character.polynomial <- function(p)
{
  p <- unclass(p)
  lp <- length(p) - 1
  names(p) <- 0:lp
  p <- p[p != 0]

  if(length(p) == 0) return("0")

  signs <- ifelse(p < 0, "- ", "+ ")
  if(signs[1] == "- ") signs[1] <- "-"
```

[10] This is an acknowledged error that is unlikely to be fixed since to do so would reduce the efficiency of `lapply` too severely.

```
  else signs[1] <- ""

  np <- names(p)
  p <- as.character(abs(p))
  p[p == "1" & np != "0"] <- ""

  pow <- paste("x^", np, sep = "")
  pow[np == "0"] <- ""
  pow[np == "1"] <- "x"
  stars <- rep("*", length(p))
  stars[p == "" | pow == ""] <- ""
  paste(signs, p, stars, pow, sep = "", collapse = " ")
}
```

Our previous example now looks like

```
> as.character(p)
[1] "1 + 2*x^2 + x^4"
```

Notice that, following convention, zero coefficients are not displayed and unit coefficients are subsumed into the power.

Printing a polynomial involves suppressing excessive significant digits, casting the polynomial into character form and displaying it in such a way that if more than one line is needed the breaks occur at natural places.

```
print.polynomial <- function(p0, ...)
{
  p <- as.character(signif(p0,
                        digits = options("digits")$digits))
  pc <- nchar(p)
  ow <- max(35, options("width")$width)
  m2 <- 0
  while(m2 < pc) {
     m1 <- m2 + 1
     m2 <- min(pc, m2 + ow)
     if(m2 < pc)
        while(substring(p, m2, m2) != " " && m2 > m1 + 1)
           m2 <- m2 - 1
     cat(substring(p, m1, m2), "\n")
  }
  invisible(p0)
}
```

The `...` argument is needed to absorb some additional arguments such as `quote` that the implicit call to the `print` generic transmits to its methods but are not needed here. Note that the return value must be set `invisible` to avoid an infinite recursion.

Coercion to functions

One of the most useful things to provide for polynomial objects is a method for coercing them to S-PLUS functions so that they may be evaluated. The horner form is a natural form for evaluation. The S-PLUS function `parse` can be used to convert a character string into an unevaluated expression. The first component of the result is an object of mode `call` that may be used as the body of a single statement function.

Functions themselves in S-PLUS may be manipulated very much like lists of length $n+1$ where n is the number of arguments. The last component of the object is the function body.

```
as.function.polynomial <- function(p)
{
  f <- function(x) NULL
  f[[2]] <- parse(text = horner(p))[[1]]
  f
}
```

An example is

```
> fp <- as.function(p)
> fp
function(x)
1 + x * (x * (2 + x * (x * (1))))
> fp(1:3)
[1]   4  25 100
```

If the function required more than one line, it could be initially set up as a character string vector and parsed. The resulting object is a vector of expressions which may be given mode `"{"`, in which form it may be used as a function body. The following example illustrates the process, but as the functions produced are unnecessarily complicated this version is not recommended.

```
as.function.poly2 <-  function(p)
{
    f <- function(x) {}
    body <- c(paste("val <- ", horner(p)), "val")
    body <- parse(text = body)
    mode(body) <- "{"
    f[[2]] <- body
    f
}
> as.function.poly2(p)
function(x)
{
  val <- 1 + x * (x * (2 + x * (x * (1))))
  val
}
```

Orthogonal polynomials

An important use of the polynomial class is to manipulate orthogonal polynomials in regression. The orthogonal polynomials may be constructed as vectors of values using the `poly` function. When this is called with one argument the value it returns has attributes that provide the coefficients for a two-term recurrence relation between the polynomials to be used. (See the help information for `poly` for the precise details.) The following function calculates the polynomials as a list using the recurrence relation. The `norm` argument specifies whether the polynomials are to be left as monic or normalized so that the resulting vectors have unit length.

```
poly.orth <- function(x, degree = length(unique(x)) - 1,
                      norm = T)
{
  at <- attr(poly(x, degree), "coefs")
  a <- at$alpha
  N <- at$norm2
  x <- polynomial()
  p <- list(polynomial(0), polynomial(1))
  for(j in 1:degree)
    p[[j + 2]] <-
      (x - a[j]) * p[[j + 1]] - N[j + 1]/N[j] * p[[j]]
  p <- p[-1]
  if(norm) {
    sqrtN <- sqrt(N[-1])
    for(j in 1 + 0:degree) p[[j]] <- p[[j]]/sqrtN[j]
  }
  p
}
```

As a simple example consider the first five orthogonal polynomials on the discrete set `0:5`:

```
> poly.orth(0:5, degree = 4, norm = F)
[[1]]:
1
[[2]]:
-2.5 + x
[[3]]:
3.3333 - 5*x + x^2
[[4]]:
-3 + 13.7*x - 7.5*x^2 + x^3
[[5]]:
1.7143 - 28.571*x + 30.714*x^2 - 10*x^3 + x^4
```

Exercises

4.13.1 Extend the `Ops.polynomial` group method function so that division of one polynomial by another is allowed in general, but always gives a

polynomial result discarding the remainder. Make this the method for both `/` and `%/%` operators.

Also, add a method for the remainder operator, `%%`.

4.13.2 Write a method function for the `predict` generic to evaluate a polynomial at a given argument `x`.

Given a polynomial $p(x)$ it is occasionally necessary to find the coefficients of another polynomial, $q(x)$ such that $p(x) \equiv q(x-a)$ for some constant a. Write a one line function to find q given p and a.

4.13.3 Write generic functions `integral` and `derivative` with methods for objects of class `polynomial` to provide a facility for elementary calculus.

(Alternatively there are two existing generic functions `diff` and `deriv` that might be given polynomial methods to supply the derivative.)

4.13.4 The S-PLUS function `polyroot` may be used to calculate the zeros of a polynomial. Write a method for the existing generic function `solve` that will find the zeros of a polynomial using `polyroot`. Make sure it does something sensible in the cases of degree 0 and 1.

4.13.5 Write a `summary` method for polynomials that will provide the zeros, stationary points and points of inflexion of a polynomial. Give the result some special class, say `polySummary`, and write a print method to display it in a neat and informative way.

4.13.6 Write a `plot` method for polynomials. Allow the user to specify the x-region of interest, or deduce it from the polynomial itself by requiring it to cover the real parts of all zeros, stationary points and points of inflexion.

4.14 Writing Dynamic Link Libraries for 4.x

The use of dynamic link libraries (DLLs) changed very considerably between S-PLUS versions 3.x and 4.0, both because 32-bit DLLs are required and since the S-PLUS interface has been changed. This section was written before the interface was documented, but the following examples have been proved to work. Illuminating comments by Luke Tierney have helped improve this section. Using DLLs is now documented in a supplementary chapter to the S-PLUS 4.0 Programmer's Guide is available from MathSoft's web site at

`http://www.mathsoft.com/splus/splsprod/update2.htm`

Generating the DLL

How to generate the DLL depends on the compiler. We will illustrate it by building a DLL of `horner.c` (Figure 4.2) using Watcom and Microsoft compilers.

A 32-bit DLL may have a single main routine giving actions to be performed on loading and unloading. This will normally do nothing and may be omitted under some compilers/linkers, including those used here.

Using Watcom 10.x the code needed is unchanged: we use `horner.c`. To compile this we used a link file `horWAT.lnk`:

```
system nt_dll initinstance terminstance
export poly
file horner
name horWAT
```

and compiled and linked by

```
wcc386 -5s -bd -bt=nt -DWIN32=1 horner.c
wlink @horWAT
```

to generate `horWAT.dll`. Multiple object files can be combined into a single DLL by specifying them on the `files` line and listing all the symbols to be exported.

Under Microsoft Visual C++ 4.2 and 5.0 we can use the file `horVC.c`:

```
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;
}
__declspec(dllexport) 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);
}
```

which differs only in the declaration of `poly`. The simplest way to build a DLL is within the development environment, but it can also be built[11] from the command line by

```
cl /MT /Ox /D "WIN32"  /c horVC.c
link /dll /out:horVC.dll horVC.obj
```

An alternative is to use `horner.c` unchanged, but to declare the symbol on the link command by

```
link /dll /out:horVC.dll /export:poly horner.obj
```

or via a `.def` file.

For the Cygwin port of the GCC compiler we can use `horner.c` unchanged by

[11] As far as we understand it, this makes a DLL following the `__cdecl` calling convention: to use the (more widespread) `__stdcall` convention add the flag `/Gz`.


```
gcc -mno-cygwin -O2 -c horner.c
echo LIBRARY horner > horner.def
echo EXPORTS >> horner.def
nm horner.o | \
sed -n "/^........ [BCDRT] _/s/^........ [BCDRT] _/ /p" \
 >> horner.def
gcc -mno-cygwin -mdll -Wl,--base-file,horner.b \
   -o horner.dll horner.o
dlltool -k --dllname horner.dll --base-file horner.b \
   --output-exp horner.e --def horner.def
gcc -mno-cygwin -mdll -Wl,--base-file,horner.b \
   -o horner.dll horner.e horner.o
dlltool -k --dllname horner.dll --base-file horner.b \
   --output-exp horner.e --def horner.def
gcc -mno-cygwin -mdll -o horner.dll  horner.e horner.o
rm -f horner.b horner.e
```

For the free `lcc-win32` compiler[12] we can use `horLCC.c`:

```
int _stdcall LibMain(void *hinstDll, unsigned long dwReason,
                     void *reserved) {return(1);}

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;
}
__declspec(dllexport) 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);
}
```

and build and use a DLL by

```
lcc -c -O -DWIN32 horLCC.c
lcclnk -dll horLCC.obj
```

and then altering the symbol names to start with an underline (for example `_poly`) in the `.C` call and in `dll.load` (see below).

[12] available from `http://www.cs.virginia.edu/~lcc-win32`

Loading and using the DLL

We can load either DLL[13] and test that the symbol has been recognised by using the appropriate name in

```
> dll.load("/dlltest/horVC.dll", call="cdecl", symbols="poly")
[1] 1
> dll.load.list()
[1] "/dlltest/horVC.dll" "slapi"
> dll.symbol.list()
[1] "S_api_get_message" "poly"
```

Note that all the symbols to be used do need to be declared in a call to `dll.load`. The call which loads the DLL returns `1`; subsequent calls can be used to declare further symbols and will return `0`. Negative return values indicate errors: use `?dll.load` for details. The functions `dll.load.info`, `dll.symbol.info`, `is.dll.loaded` and `is.dll.symbol` provide more specific information.

We can test the code and unload the DLL by

```
> 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
> dll.unload("/dlltest/horVC.dll")
```

It is desirable to unload the DLL before loading a new version, and Windows will not permit the creation of a new version whilst it is loaded.

Calling S entry points

It is possible to use routines within the S DLL by importing their symbols (from `Sqpe.dll`), but the details can be very tricky, so this subsection is for the brave.

We give an (unnecessary) example of a function to generate gamma random variables; much of the code in the literature for random-number generation is written in Fortran and we used that from the Appendix of Ripley (1987).

Our example used Watcom 10.x compilers. We need files `gamma.f`:

```
      FUNCTION RND()
      DOUBLE PRECISION unif_rand
C      unif_rand() is a call to the S random number generator.
      RND = unif_rand()
      END

      SUBROUTINE MYGAMMA(N, ALPHA, X)
      REAL ALPHA, X(N)
```

[13] The argument `calling.convention` needs to be set appropriately. The options are `"stdcall"` and `"cdecl"`. Visual C++ DLLs can be built with either convention. The difference is whether the caller or callee is responsible for clearing the stack. We believe `"cdecl"` is correct for Watcom DLLs.


```
      IF(ALPHA .GE. 1.0) THEN
      DO 10 I=1,N
 10      X(I) = GCF(ALPHA)
      ELSE
      DO 20 I=1,N
 20      X(I) = GS(ALPHA)
      ENDIF
      END

      FUNCTION GS (ALPHA)
      REAL GS,ALPHA,B,P,X
      DATA E/2.71828182/
      B = (ALPHA+E)/E
   10 P = B*RND()
      IF (P .GT. 1.0) GO TO 20
      X = P**(1./ALPHA)
      IF (X .GT. -LOG(RND())) GO TO 10
      GS = X
      RETURN
   20 X = -LOG((B-P)/ALPHA)
      IF (X**(ALPHA-1.0) .LT. RND()) GO TO 10
      GS = X
      RETURN
      END

      FUNCTION GCF (ALPHA)
      REAL GCF,ALPHA,AA,APREV,C1,C2,C3,C4,C5,U1,U2,W,X
      SAVE APREV
      DATA APREV/0.0/
      IF (ALPHA .EQ. APREV) GO TO 10
      C1 = ALPHA-1.0
      AA = 1.0/C1
      C2 = AA*(ALPHA-1.0/(6.0*ALPHA))
      C3 = 2.0*AA
      C4 = C3+2.0
      IF (ALPHA .GT. 2.5) C5 = 1.0/SQRT(ALPHA)
   10 U1 = RND()
      U2 = RND()
      IF (ALPHA .LE. 2.5) GO TO 20
      U1 = U2+C5*(1.0-1.86*U1)
      IF (U1.LE.0.0 .OR. U1.GE.1.0) GO TO 10
   20 W = C2*U2/U1
      IF (C3*U1+W+1.0/W .LT. C4) GO TO 30
      IF (C3*LOG(U1)-LOG(W)+W .GE. 1.0) GO TO 10
   30 GCF = C1*W
      APREV = ALPHA
      RETURN
      END
```

`Crnd.c`:

```
#include <S.h>
void start_rnd() seed_in((long*)NULL);
void stop_rnd()  seed_out((long*)NULL);
```

and `gamma.lnk`:

```
system nt_dll initinstance terminstance
export mygamma
export start_rnd
export stop_rnd
import unif_rand Sqpe
import seed_in Sqpe
import seed_out Sqpe
file gamma
file Crnd
name mygamma
```

and we can prepare the DLL by

```
COMPILE Crnd.c
COMPILE gamma.f
wlink @gamma
```

Note that the `COMPILE` script does set suitable compiler flags for the compile steps.

We can use it by

```
dll.load("/dlltest/mygamma.dll", call="cdecl",
         symbols=c("mygamma", "start_rnd", "stop_rnd"))

mygamma <- function(n, alpha)
{
  .C("start_rnd")
  res <- .Fortran("mygamma",
                  as.integer(n), as.single(alpha), x=single(n))
  .C("stop_rnd")
  res$x
}
```

S-PLUS has a perfectly good inbuilt function `rgamma`, but this example will provide a template for adding functions to generate samples from other distributions.

Unlike S-PLUS 3.3, it is possible for both C and FORTRAN routines in a DLL to perform standard I/O. For C the include files mentioned on page 14 need to be included: for FORTRAN the routines[14] `DBLEPR`, `INTPR` and `REALPR` are used. Various entry-points will need to be declared as imports from `Sqpe` at the link phase: the simplest way to discover which is to try linking and see

[14] see page 16

which entry-points are not satisifed. Alternatively, for some linkers the import library `%SHOME%\lib\Sqpe.lib` (Microsoft VC++) or `%SHOME%\lib\Sqpew.lib` (Watcom) can be included in the link.

There are very many[15] external symbols in `Sqpe.dll` but only a few are documented. Those we know of are listed on page 13. The release notes claim that `PROBLEM ... RECOVER` and `WARNING ... MESSAGE` can be used in code to be included in DLLs, but in our experiments with S-PLUS 4.0 release 2 this caused a system crash if invoked from a DLL built with a Watcom compiler. (The header files in release 3 have been changed so that these can be used with Watcom 10.5 but not with earlier compiler releases.)

It did work for VisualC++. Consider `VCtest.c`

```
#include <S.h>
#  include <newredef.h>

LibExport void  probtest(long int *i)
{
  int j = *i;
  printf("i is %d\n", j);
  if(j > 10) PROBLEM "test" WARNING(NULL_ENTRY);
}
```

Note that the S-PLUS header files are set up to work with Visual C++ as well as Watcom compilers. We can compile and link[16] this by

```
cl /MT /Ox /D "WIN32"  /I%SHOME%\include /c VCtest.c
link /dll /out:VCtest.dll VCtest.obj %SHOME%\lib\Sqpe.lib
```

Then we can use;

```
> dll.load("/dlltest/VCtest.dll", call="cdecl",
         symbols="probtest")
> invisible(.C("probtest", as.integer(1)))
i is 1
> invisible(.C("probtest", as.integer(11)))
i is 11
Warning messages:
  test in: .C("probtest",  ...
```

There are problems with return values from functions in `Sqpe.dll` called from code generated under Visual C++, so it is not easy to use the random-number functions, for example. The header file `S_ansi.h` contains the following warning:

> Exports from `sqpe.dll` use the standard Watcom calling convention—not stdcall like Microsoft and most DLLs.

[15] 3941 on the version we examined

[16] provided the development tools were installed when S-PLUS was installed, as these contain the import library for `Sqpe.dll`.

but this fails to mention the return conventions. We did manage to use the `unif_rand()` function from Microsoft Visual C++ by using tricks with unions using `VCtest2.c`:

```
typedef struct uu1 uu;
struct uu1 { int a; int b; };
union { double d; uu fake; } u;

__declspec(dllimport) uu unif_rand();
__declspec(dllimport) void setseed();

__declspec(dllexport) void
urand(long int *m, long int *seed, double *p)
{
  long i;
  setseed(seed);
  for (i = 0; i < *m; i++) {
    u.fake = unif_rand();
    p[i] = u.d;
  }
}

cl /MT /Ox /D "WIN32"  /I%SHOME%\include /c VCtest2.c
link /dll /out:VCtest2.dll VCtest2.obj %SHOME%\lib\Sqpe.lib

> set.seed(123); runif(4)
[1] 0.8756982 0.5321866 0.6700785 0.9921576
> dll.load("/dlltest/VCtest2.dll", call="cdecl", symbols="urand")
[1] 1
> .C("urand", as.integer(4), as.integer(123), x=double(4))$x
[1] 0.8756982 0.5321866 0.6700785 0.9921576
```

Similar trickery will be needed with `float` values, since for both double and float return values the Watcom convention is to use ordinary registers whereas Microsoft appears[17] to use floating point registers.

4.15 Dialogs and Menus in S-PLUS 4.x

S-PLUS 4.x introduced a new way to program menus and dialogs, quite different from that in S-PLUS for Windows 3.3. In essence the new system programs the Axum engine on which the 4.x GUI is based. There are some examples in the S-PLUS User's Guide and Programmer's Guide, but the explanations and documentation are rather incomplete.

The main step is to associate a dialog box with a special-purpose S function. Then when this function is invoked from a menu or a toolbar button or (by double-clicking) in an object browser the dialog box is launched; when completed it

[17] by examining assembler code

supplies the arguments to the S function. This in turn will call standard functions and apply the requested actions to the results. Conventionally such functions have names that begin with `menu` such as `menuLm` and the system examples are in library `menu`.

Let us consider how to set up a special-purpose function interface to `lda`. The function might be defined[18] as

```
menuLDA <-
function(formula, data, subset, na.omit.p=T,
         method="moment", newdata=NULL, predict.save=NULL,
         predict.method="plug-in", plot.p=F, plot.dimen=99)
{
  fun.call <- match.call()
  fun.call[[1]] <- as.name("lda.formula")
  if(na.omit.p) fun.call$na.action <- as.name("na.omit")
  else fun.call$na.action <- as.name("na.fail")
  fun.args <- is.element(arg.names(fun.call),
     c("formula", "data", "subset", "na.action", "method"))
  fun.call <- fun.call[c(T, fun.args)]
  ldaobj <- eval(fun.call)
  tabPredict.lda(ldaobj, newdata, predict.save, predict.method)
  if(plot.p) tabPlot.lda(ldaobj, plot.dimen=plot.dimen)
  invisible(ldaobj)
}

tabPredict.lda <-
function(object, newdata=NULL, predict.save=NULL,
         predict.method="plug-in")
{
  if(!is.null(predict.save)) {
    if(is.null(newdata))
      predobj <- predict(object, method=predict.method)
    else
      predobj <- predict(object, newdata=newdata,
                         method=predict.method)
    if(exists(predict.save, where = 1)) {
      newsave.name <- unique.name(predict.save, where = 1)
      assign(newsave.name, predobj, where = 1)
      warning(paste("Predictions saved in", newsave.name))
    } else assign(predict.save, predobj, where = 1)
  }
  invisible(NULL)
}

tabPlot.lda <- function(object, plot.dimen=99)
{
  plot.lda(object, dimen=plot.dimen)
  invisible(NULL)
}
```

[18] The function in our libraries has more features.

The function is called when the `OK` or `Apply` button on the dialog box is selected. By default a dummy dialog box will be generated that allows the arguments to be filled in. Figure 4.4 shows the default dialog box for our `menuLDA`.

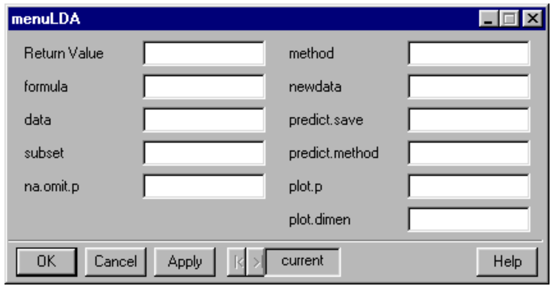

Figure 4.4: Default dialog box for `menuLDA`.

A customized dialog box is selected by specified a `FunctionInfo` property. For our example we might use

```
guiCreate("FunctionInfo", Name = "menuLDA",
          Function = "menuLDA",
          HelpCommand = "help(lda)",
          DialogHeader = "Linear Discriminant Analysis",
          PropertyList = c("ldaModelPage", "ldaPlotPage",
            "ldaPredictPage", "SPropPFEnableButton"),
          CallBackFunction = "backLDA",
          ArgumentList = "#0=ldaSaveAs, #1=SPropPFFormula,
  #2=SPropDataFrameList, #3=SPropSubset, #4=SPropOmitMissing,
  #5=ldaFitMethod, #6=SPropPredictNewdata,
  #7=SPropSavePredictObject, #8=ldaPredictMethod,
  #9=ldaPlot, #10=ldaPlotDimen")
```

The result will be the multi-tabbed box shown in Figure 4.5.

`guiCreate` is a call to the GUI programming language, and defines information that is stored in the user's `_Prefs` directory when the current session finishes. This information refers to other properties, which are either created by `guiCreate` or (as in `SPropPFEnableButton`) are already defined. (All of these further properties are also stored in the `_Prefs` directory, even the system ones.) The dialog box is built up hierarchically. At the top level the `FunctionInfo` property defines the dialog as being made up of three tabbed pages and the standard bottom row of buttons. Each page is then made up of *groups* (or wide groups), for example

```
guiCreate("Property", Name="ldaModelPage", Type="Page",
          DialogPrompt="Model",
          PropertyList=c("ldaModelData", "SPropPFFormulaG",
            "ldaMethod", "ldaSaveModel"))
```


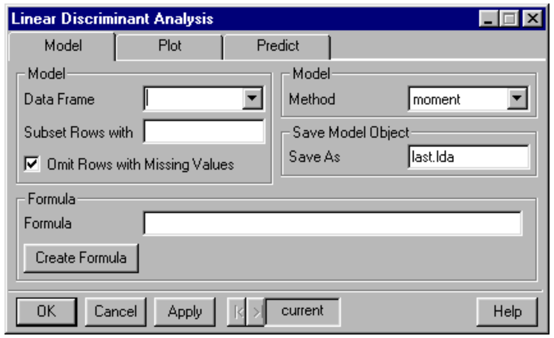

Figure 4.5: Customized dialog box for `menuLDA`.

```
guiCreate("Property", Name="ldaModelData", Type="Group",
          DialogPrompt="Data",
          PropertyList=c("SPropDataFrameList", "SPropSubset",
            "SPropOmitMissing"))
guiCreate("Property", Name="ldaMethod", Type="Group",
          DialogPrompt="Fit method",
          PropertyList="ldaFitMethod")
guiCreate("Property", Name="ldaSaveModel", Type="Group",
          DialogPrompt="Save Model Object",
          PropertyList="ldaSaveAs")
```

and the groups contain basic properties such as

```
guiCreate("Property", Name="ldaFitMethod",
          DialogPrompt="Method",
          DialogControl="List Box",
          OptionList=c("moment", "mle", "mve", "t"),
          DefaultValue="moment", UseQuotes=T)
guiCreate("Property", Name="ldaSaveAs", DialogPrompt="Save As",
          DialogControl="String", DefaultValue="last.lda")
```

We can begin to see how the return list is made up of the values of (most of) the fields in the basic properties. Item `#0` is used to assign the result of the function call, and `#1`, `#2`, ... specify the order in which the fields in the dialog box are matched to the function arguments.

The layout of the dialog box is done automatically, and optimizing seems a matter of trail-and-error. Two columns are used, and the groups/items are used in order. However, wide groups (such as the formula builder `SPropPFFormulaG`) span both columns, and it may be necessary to add newline properties judiciously to ensure that the side groups are assigned to the first column and not overlaid by entries in the second column.

There is one argument in the definition of the `FunctionInfo` property that we have not yet mentioned, the *callback function*. This is an S function that is called whenever a field in the dialog box is altered. It will normally contain calls to the GUI to change other properties of the dialog box. For `menu.lda` we used

```
"backLDA" <- function(data)
{
  activeprop <- cbGetActiveProp(data)
  activevalue <- cbGetCurrValue(data, activeprop)
  switch(activeprop,
         ldaPlot = {
           if(activevalue == "F") {
             data <- cbSetEnableFlag(data, "ldaPlotDimen", F)
           } else {
             data <- cbSetEnableFlag(data, "ldaPlotDimen", T)
           }
         }
         ,
         SPropPFButton = {
           data <- backform1(data) # code copied from backLM
         }
         ,
         SPropPFEnableButton = {
           data <- cbSetEnableFlag(data, "SPropDataFrameList", T)
           data <- cbSetEnableFlag(data, "SPropPFFormula", T)
           data <- cbSetEnableFlag(data, "SPropPFButton", T)
         }
         )
  data
}
```

The main purpose of this code is to disable the 'integer spinner' that sets the dimension of the plots unless plotting is selected and to handle the Create Formula button (by bringing up another dialog box and substituting its result in the appropriate field).

As the GUI properties are stored in the user's preferences files, they are not easily shared in a library, for example. The friendliest way to share a dialog box is to create the properties in the library's `.First.lib` function and to remove them in the `.Last.lib` function. Properties (and `FunctionInfo` objects) can be deleted by a call to `guiRemove`, for example by

```
menuLDAPropDel <- function()
{
  if(!exists("guiRemove")) return(NULL)
  guiRemove("FunctionInfo", Name="menuLDA")
  guiRemove("Property", Name="ldaModelPage")
  guiRemove("Property", Name="ldaModelData")
  guiRemove("Property", Name="ldaMethod")
  guiRemove("Property", Name="ldaSaveModel")
  guiRemove("Property", Name="ldaFitMethod")
  guiRemove("Property", Name="ldaSaveAs")
    ....
}
```


Note that we have to take care to invoke `guiRemove` (or `guiCreate`) only if the GUI is actually running, as we could have invoked the library from `sqpe.exe`. Despite its name, the `Last.lib` in an S database is called whenever the database is detached.

How do we find the details of these properties? The documentation is sparse, and we can either copy existing examples or examine the GUI dialog boxes used to create new properties. In either case an object browser must be used. Open an object browser and create a page with filtering set to the interface class `Property`. There are many properties, and the right-hand panel will (probably) have several pages. Selecting any property, right-clicking and selecting Create Property... will bring up a dialog box that describes the possible arguments to `guiCreate` for a property: using the Help button on that dialog box can be informative. Such dialog boxes can be used to create all the properties: the history log will then show the commands that can be used from S to recreate them. As there is no simple way to find out at a later date what commands have been used to create the properties (or even what properties have been added), it is important to keep a copy of the relevant parts of the history log. However, a command to re-create[19] a property can be generated by dragging the icon for a property from an object browser to an open *script* window.

Adding items to the menus

Menu items are added in exactly the same way as dialog boxes, by adding GUI properties, this time of class `MenuItem`. To add a separate menu for the `MASS` library we could use

```
guiCreate("MenuItem",
          Name="SPlusMenuBar$MASS", Type="Menu",
          MenuItemText= "&MASS", Index=11, OverWrite=F)
guiCreate("MenuItem",
          Name="SPlusMenuBar$MASS$LDA", Type="MenuItem",
          Action="Function", Command="menuLDA",
          MenuItemText= "&LDA...")
   ....
# to remove this use
guiRemove("MenuItem", Name="SPlusMenuBar$MASS")
```

The numbering of the menus is potentially confusing, as not all menus are enabled at any one time. The way to discover the numbers required is to open an object browser, filter on the interface class `MenuItem` and explore the full menu tree. This example adds a MASS menu after the Statistics menu (which is number 10).

An alternative approach is to add items to the Statistics menu, which is of course more likely to be upset by future changes to S-PLUS. We could use

[19] including all the fields which were not

```
guiCreate("MenuItem",
  Name="SPlusMenuBar$Statistics$Multivariate$Discrimination",
  Type="Menu", MenuItemText= "&Discrimination")
guiCreate("MenuItem",
  Name="SPlusMenuBar$Statistics$Multivariate$Discrimination$LDA",
  Type="MenuItem", Action="Function", Command="menuLDA",
  MenuItemText= "&LDA...")
guiCreate("MenuItem",
  Name="SPlusMenuBar$Statistics$Multivariate$Discrimination$QDA",
  Type="MenuItem", Action="Function", Command="menuQDA",
  MenuItemText= "&QDA...")
# to remove this use
guiRemove("MenuItem",
  Name="SPlusMenuBar$Statistics$Multivariate$Discrimination")
```

to add a pull-out menu to the existing Multivariate sub-menu of the Statistics menu.

Again, cooperative behaviour is needed; the library's `.First.lib` function might add the menu items and its `.Last.lib` function might remove them, as the menu settings are stored in the user's `_Prefs` directory. Unfortunately, the `.Last.lib` will sometimes fail: until this bug is fixed supply add and remove functions for your users to call.

Context-sensitive menus

Another use for menus is to provide class-specific items on the pop-up menu produced by right-clicking (or double-clicking) on an item in an object browser. First we define the additional items for this menu and then assign it (as a property of type `ClassInfo`).

```
guiCreate("MenuItem", Name="lda", Type="Menu",
          DocumentType="lda")
guiCreate("MenuItem", Name="lda$print", Type="MenuItem",
          DocumentType="lda", Action="Function",
          Command="tabResults.lda", MenuItemText="Print",
          ShowDialogOnRun=F)
guiCreate("MenuItem", Name="lda$plot", Type="MenuItem",
          DocumentType="lda", Action="Function",
          Command="tabPlot.lda", MenuItemText="Plot...",
          ShowDialogOnRun=T)
guiCreate("MenuItem", Name="lda$predict", Type="MenuItem",
          DocumentType="lda", Action="Function",
          Command="tabPredict.lda", MenuItemText="Predict...",
          ShowDialogOnRun=T)

guiCreate("ClassInfo", Name="lda", ContextMenu="lda",
          DoubleClickAction = "tabResults.lda")
```

We use the functions that we have already created for use in `menuLDA` as well as


```
tabResults.lda <- function(object) invisible(print.lda(object))
```

However, these functions now need to be associated with dialogs, and the appropriate defaults will be slightly different from those of the main dialog box. Some appropriate definitions are

```
guiCreate("FunctionInfo", Name = "tabPlot.lda",
          Function = "tabPlot.lda",
          DialogHeader = "LDA plot",
          PropertyList = c("SPropInvisibleReturnObject",
            "SPropCurrentObject","ldaPlot2Page",
            "SPropPFEnableButton"),
          ArgumentList = "#0=SPropInvisibleReturnObject,
                   #1=SPropCurrentObject, #2=ldaPlotDimen2")
guiCreate("Property", Name="ldaPlot2Page", Type="Page",
          DialogPrompt="Plot",
          PropertyList="ldaPlotDimen2")
guiCreate("Property", Name="ldaPlotDimen2", DialogPrompt="Dimen",
          DialogControl="Integer Spinner", DefaultValue=3)

guiCreate("FunctionInfo", Name = "tabPredict.lda",
          Function = "tabPredict.lda",
          DialogHeader = "LDA prediction",
          PropertyList = c("SPropInvisibleReturnObject",
          "SPropCurrentObject","ldaPredictPage",
            "SPropPFEnableButton"),
          ArgumentList = "#0=SPropInvisibleReturnObject,
            #1=SPropCurrentObject, #2=SPropPredictNewdata,
            #3=SPropSavePredictObject, #4=ldaPredictMethod")
```

Restoring the defaults

In experimenting with menus and properties it is easy to leave unwanted elements behind, and on occasion to remove wanted properties or parts of the menu tree. It is helpful to know that the menu tree is stored in the file `_Prefs\smenu.smn` and the basic properties in `_Prefs\axprop.dft`; if either of these files is deleted it will be replaced by a copy of the system version of the file when S-PLUS is next started. Similarly, the `FunctionInfo` and `ClassInfo` properties are stored in files `_Prefs\axfunc.dft` and `_Prefs\axclinfo.dft`.

4.16 Tips and Checklist for Programmers

Our book, even Chapter 4, is not primarily about programming. Most users of S-PLUS use the S language casually, writing a few functions for their own use. Some users are employed solely to produce S code. Even though S is a high level language and programming in it is not as challenging as in lower-level languages it is a skill and the formal techniques developed for software engineering are highly pertinent and very useful. It is easy to ignore this and to offer idiosyncratic and poorly documented code for public use with a comprehensive disclaimer. Such code is at best unhelpful. The forthcoming book about S version 4, Chambers (1998), applies the viewpoint of a professional programmer to software development in S.

- *Check* your code. It is clear from the postings to the `S-news` discussion list that many S programmers have not tested their proferred code at all, let alone thoroughly. It is helpful for a library to be distributed to come with a set of examples to test out its facilities, with the expected answers, yet most do not. For example, our software for the book comes for Unix with a `test` directory that tests out all the code that needs to be compiled as this is the main difficulty in moving between Unix platforms. More generally, the scripts for all the examples in the book and the complements are available, and the expected answers are in the book. We check these before every new release of our software, and find it a very revealing check on new versions of S-PLUS.
- *Organize* your code. How to do this is a matter of taste, but we have found it essential to have a master archive and scripts to generate all the distributed versions (including those for Windows, and the formatted help pages and so on). If feasible[20] consider the use of a version control system (such as RCS, SCCS, CVS) to ensure that all changes are applied sequentially and recorded.
- *Refine* and *polish* your code in the same way that you would polish your English prose. As you would use a dictionary, use your reference materials (the on-line help and especially the S code of system functions) to check your understanding. Other people will be reading your code and in refining it you will often find ways to improve and simplify it that you did not expect.
- *Document* your code fully, clearly and (preferably) in a way that can be verified by formal tests. This is another way of ensuring that your logic is correct and your plan is complete.

Checklist

Here is a list of common mistakes that are so easy to make that we check our own code against it before releasing it. We leave to the reader to find a favourite way to do this: we use `grep` on the source code or a regular-expression search in an editor.

- Did you really mean `max` and `min` with more than one argument rather than `pmax` and `pmin`? The function `max` is equivalent to concatenating (applying `c` to) the arguments and then finding the largest value in a single vector. It may be better to use explicitly `max(c(arg1, arg2))` for future reference.

[20] This is not feasible when working across continents and on different machines which are not permanently networked, so we do not use these. Rather, our master archive is a `zip` archive.

- Any comparison with `NA`, for example `x == NA`, should be written using `is.na`.
- Check that `&` should not be `&&` and *vice versa*, and that `|` and `||` are not confused.
- The construction `1:length(x$y)` will give `c(1,0)` if `x$y` is empty or `x` does not have component `y`. Almost always `seq(along=x$y)` is better.
- Watch the precedence of `:` which comes below `^` and unary minus but above the other arithmetic operators. Use parentheses copiously.
- Consider if `!is.null(x)` or `length(x) > 0` is the better test. The difference is that empty vectors are not null. The most common idiom is to test for the existence of a component of list: `is.null` is appropriate as `x$y` is null if the component does not exist[21].
- Check the behaviour you want when subscripting matrices and arrays. The default is to drop dimensions of size one, which can result in a vector. For safety, add `drop=F` or `drop=T` to all such calls.
- Check that when `dim` is called its argument is known to be a matrix, array or data frame.
- Ensure that the last statement in a `for` loop is `NULL`. In S-PLUS 3.x this avoids the last expression of the loop being retained in memory. In S-PLUS 4.x it is not needed, but is harmless.
- Check that method functions have exactly the same set of arguments as the generic function *including* a `...` argument[22].
- Avoid formal argument names such as `print` and `plot` and other common function names. Use a name like `plot.it` or even `plot.` which partial matching will allow the user to abbreviate to `plot`. This avoids warnings in S-PLUS 3.x and also works around some bugs.
- Do not rely on partial matching of arguments or names of components, but use the full name. This avoids any problems with future extra arguments or lists with more components than you expected.
- Check that the storage modes of *all* arguments passed to `.C` or `.Fortran` are known, probably by coercion (as `as.double`) or by setting `storage.mode`. Here is an example of what can go wrong.

```
> for(df in 2) print(gam(log(perm) ~ area + peri +
    s(shape, df=df), data=rock))
```

[21] It is also null if the component exists but is literally `NULL`, but such components are rare: for example `x$y <- NULL` removes component `y` rather than setting it to `NULL`. If this difference matters, you could use `!is.na(pmatch("y", names(x)))`.
[22] to mop up arguments given to future methods that might call this one. We wish the authors of `predict.lm` and `summary.survreg` had remembered this one; its omission made fixing `predict.glm` needlessly complicated.

```
Degrees of Freedom: 48 total; 42.99963 Residual
Residual Deviance: 30.61766
> for(df in 2:3) print(gam(log(perm) ~ area + peri +
    s(shape, df=df), data=rock))

Degrees of Freedom: 48 total; -1072707733 Residual
Residual Deviance: 31.92164
    ....
```

The problem is that `2:3` gives `df` storage mode `"integer"`, and the author of `s.wam` did not check the mode in the call to the FORTRAN function `backfit`.

This will become much more pressing in future versions of S-PLUS, where it will be much easier to get vectors of storage mode `"integer"`.

- Remember the scope rules (page 154); in particular consider if your functions will work when called from within a function.

  Within a function, a call to `eval` probably needs a non-default value for `local`.

  Functions defined within functions do not have automatic access to local variables of the parent.

  Functions such as `tapply`, `apply` and `nlminb` make provision for extra arguments to be passed down to their function arguments: they are needed more often than is commonly realised.
- Use `masked` or `conflicts` to check that any conflicts of names are deliberate.
- Re-read the tips on page 161 of the book.
- If you have compiled code, check the symbol names (see page 57).

Appendix C

Using S-PLUS Libraries

C.2 Creating a library

Appendix C.2 of the printed text gives instructions for creating a simple library section. Here we consider how libraries using compiled code can be created.

The following sequence of steps can be followed.

Unix

1. Create a directory in the library with the section name.
2. Change to that directory and create `.Data` and `.Data/.Help` subdirectories.
3. Run S-PLUS and create the desired objects, which will be saved within the `.Data` directory. Often this is best done by

   ```
   $ Splus < section.q
   ```

   where `section.q` is a file listing the functions and containing expressions to generate the datasets. (The suffix `.q` is recommended to avoid any confusion with assembler code, which often has suffix `.s` on Unix systems.)
4. Use the `prompt` function within S-PLUS to create templates of help files for the objects in the library which will be publicly accessible.
5. Edit the `*.d` files to create the help files. Copy the `*.d` files to the `.Data/.Help` directory *without* the `.d` extension, by

   ```
   Splus HINSTALL .Data/.Help *.d
   ```

   or by hand. For example, C-shell users may use

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

   and SYS V Bourne shell (and BASH and Korn) users can use

```
for f in *.d
do
  cp $f .Data/.Help/`basename $f .d`
done
```

If a help file refers to several objects (by multiple `.FN` lines) link files with the names of the subsequent objects to the first one. (This is done automatically by `HINSTALL`.)

If needed, add pre-processed help files in `.Data/.Cat.Help` (see page 5) by

```
Splus CATHELP .Data
```

6. Create a `README` file in the main directory describing briefly the purpose of the library and with one-line descriptions of the public objects.
7. Add a one-line description of the library to the `README` file in its parent library directory.
8. Compile and link as needed any dynamically-loadable modules within the main directory (see Section 4.9).
9. Create a `.First.lib` object to dynamically load any modules as required. For example, for a `surv4` library section we used:

```
.First.lib <- function(lib.loc, section)
{
   path <- paste(lib.loc, section, "surv4_l.o", sep = "/")
   dyn.load(path)
   options(na.action = "na.omit")
   options(contrasts = c("contr.treatment", "contr.poly"))
   cat("default contrasts changed to contr.treatment")
   cat("default na.action changed to na.omit")
}
```

Note that the `options` settings are those appropriate to this survival analysis library, and may not be elsewhere, hence the warning.

Recent S-PLUS versions (version 3.3 on some platforms and all versions of 3.4) have a function `dyn.load.lib` which can be used to simplify the S code to load modules. The `.First.lib` function can be as simple as

```
.First.lib <- function(...)
   dyn.load.lib(..., basename="myname")
```

This will choose `dyn.load`, `dyn.load.shared` or `dyn.load2` as appropriate for the run-time platform. However, the appropriate object module must have been made available, this being `myname.so` for SGI and DEC Alpha, and `myname_l.o` for all other Unix platforms.

10. Use `help.findsum(".Data")` in S-PLUS to create an index for use by `help.start`, or use

```
Splus help.findsum .Data
```

at the Unix prompt.

Windows

1. Create a directory in the library with the section name.
2. Change to that directory and create `_Data` and `_Data\_Help` subdirectories.
3. Run S-PLUS and create the desired objects, which will be saved within the `.Data` directory. Often this is best done by a BATCH command using a file `section.q` listing the functions and containing expressions to generate the datasets. (The suffix `.q` is conventional for S code.)
4. Use the `prompt` function within S-PLUS to create templates of help files for the objects in the library which will be publicly accessible.
5. The templates are put directly into the `_Data\_Help` directory, and may have mapped names of the form `__27`. Edit these files in your favourite editor.

   (Optional.) Convert these help files to a Windows help file (see page 4).
6. Create a file `README.TXT` in the main directory describing briefly the purpose of the library and with one-line descriptions of the public objects.
7. Add a one-line description of the library to the file `README.TXT` file in its parent library directory.
8. Compile and link as needed any dynamically-loadable modules and DLLs within the main directory (see Section 4.9).
9. Create a `.First.lib` object to dynamically load any modules as required. For example, for a `surv4` library section we used:

```
.First.lib <- function(lib.loc, section)
{
   path <- paste(lib.loc, section, "surv4.obj", sep = "/")
   dyn.load(path)
   options(na.action = "na.omit")
   options(contrasts = c("contr.treatment", "contr.poly"))
   cat("default contrasts changed to contr.treatment")
   cat("default na.action changed to na.omit")
}
```

(The separator could be changed to `"\\"`, but `"/"` will work.)

Note that the `options` settings are those appropriate to this survival analysis library, and may not be elsewhere, hence the warning.

Recent S-PLUS versions (3.3, 4.x) have a function `dyn.load.lib` which can be used to simplify the S code to load modules. The `.First.lib` function can be as simple as

```
.First.lib <- function(...)
   dyn.load.lib(..., basename="myname")
```

However, the appropriate object module must have been made available, this being `myname.obj` under Windows.

10. Version 3.3: Launch S-PLUS with this directory as the working directory, to ensure that its summary file `_Data\__sum.txt` is up to date.

    Version 4.x: Launch S-PLUS and attach the library, then use the function `make.DB.summary(n)` (where `n` is the position at which the library is attached) to update the summary files `_Data\__sum4.txt` and `_Data\__sum4i.txt`.

Distributing library sections

Windows

Library sections for Windows can be distributed in binary form. The convention is to use a `.zip` in a format compatible with `PKUNZIP`. We do this from the library directory (the parent of the section) by using

```
zip -kr section.zip section
```

Large numbers of small files in the `_Data` directory can waste enormous amounts of space, and from S-PLUS 3.2 on it is more efficient to use `__BIG` indexed files for S-PLUS objects, as the system itself does. There is an undocumented function

```
"wbigfile"<-
function(where = 1, o = objects(), directory = ".",
        bigfile = paste(directory, "__BIG", sep = "/"),
        bigfile.index = paste(directory, "__BIGIN", sep = "/"))
{
  data <- vector(mode = "list", length = length(o))
  for(i in seq(along = o))
    data[[i]] <- get(o[[i]], where = where, immediate = T)
  names(data) <- o
  .Internal(put.to.bigfile(data, bigfile, bigfile.index,
    "standard"), "S_put_to_big_file", T, 0)
}
```

which takes a list of objects and produces a single file `__BIG` with index file `__BIGIN`. This can be used to convert a library by the following batch file (run from the main directory of the library section).

```
rem > null.q
start /w /min splus.exe BATCH convert.q
move _Data _Data.old
mkdir _Data
move __BIG _Data
move __BIGIN _Data
rem > _Data\__db3.1
move _Data.old\_Help _Data
start /w /min splus.exe BATCH null.q
del null.q
```


where `convert.q` contains

```
library(wbigfile)
if(exists("last.dump")) rm(last.dump)
if(exists(".Last.value")) rm(.Last.value)
wbigfile()
```

and `wbigfile` is in library `wbigfile`.

Unix

As the binary files (both the files in the `.Data` directory and any compiled modules) differ between platforms under Unix the convention is to distribute the source files, together with instructions as to how to recreate the library section.

There are two approaches. One, exemplified by our software, is to write a shell script or `Makefile` to perform most of the steps. We turned to shell scripts when we found difficulties in writing `Makefile`s that worked on all the S-PLUS platforms. The scripts need to cope with producing compiled objects for both `dyn.load` and `dyn.load.shared` and a suitable `.First.lib` function. For the `nnet` library we used

```
#!/bin/sh
S=${S-Splus}
mkdir .Data .Data/.Help
echo --reading in S files
cat nnet.q multinom.q First.lib.q | $S
echo "--compiling (expect a warning)"
if [ $# -gt 0 ] && [ $1 = "shared" ]
then
    $S SHLIB -o nnet.so nnet.c
else
    $S COMPILE nnet.c
    mv nnet.o nnet_l.o
fi
echo --Installing help
for f in *.d
do
   cp $f .Data/.Help/`basename $f .d`
done
$S help.findsum .Data
rm -f .Data/.Audit .Data/.Last.value
```

This is invoked as `./Ins` or `./Ins shared` as appropriate: if the name of the S-PLUS command is not `Splus`, first set the environmental variable `S` to the command name.

The other approach is to use

```
Splus CHAPTER arguments
```

to create a `Makefile` on the target platform. Consider the steps needed for our `nnet` library, which is distributed in source form. Running

```
Splus CHAPTER *.q *.c *.d
```

creates a `Makefile`. Then we can use

```
$ make install
    ....
mkdir .Data
Splus QINSTALL .Data force_ld.q First.lib.q multinom.q nnet.q
Version 3.4 Release 1 for Sun SPARC, SunOS 5.3 : 1996
Working data will be in /tmp/1_1348_Data

>>> QINSTALL : installing to ".Data"

S-PLUS : Copyright (c) 1988, 1996 MathSoft, Inc.
S : Copyright AT&T.
Version 3.4 Release 1 for Sun SPARC, SunOS 5.3 : 1996
Working data will be in /tmp/2_1348_Data
NULL
NULL
NULL
NULL
Warning: assigning "cat" masks an object of the same name
      on database 2
Warning: assigning "assign" masks an object of the same name
      on database 4
Warning: assigning "get" masks an object of the same name
      on database 4
>>> Installing .First.lib to .Data
>>> Installing VRdyn.load.lib to .Data
>>> Installing add.net to .Data
>>> Installing add1.multinom to .Data
>>> Installing class.ind to .Data
>>> Installing drop1.multinom to .Data
>>> Installing eval.nn to .Data
>>> Installing force.loading to .Data
>>> Installing max.col to .Data
>>> Installing multinom to .Data
>>> Installing nnet to .Data
>>> Installing nnet.Hess to .Data
>>> Installing norm.net to .Data
>>> Installing predict.multinom to .Data
>>> Installing predict.nnet to .Data
>>> Installing print.multinom to .Data
>>> Installing print.nnet to .Data
>>> Installing summary.nnet to .Data
>>> Installing which.is.max to .Data
mkdir .Data/.Help
Splus HINSTALL .Data/.Help class.ind.d multinom.d nnet.Hess.d
        nnet.d predict.nnet.d which.is.max.d
HINSTALL: installing class.ind
HINSTALL: installing multinom
```



```
HINSTALL: installing nnet.Hess
HINSTALL: installing nnet
HINSTALL: installing predict.nnet
HINSTALL: installing which.is.max

The .Help.find.sum file in .Data/.Help may now be out of date.
You can remake it with:

        Splus help.findsum .Data

Splus help.findsum .Data
Making new help find file: .Data/.Help/.Help.find.sum

$ make dyn.load
Splus make.init nnet .Data
This takes a while ... be patient and watch the dots
  (one per object)
...................
cc -c -I${SHOME-`Splus SHOME`}/include  -O nnet_i.c
cc -c -I${SHOME-`Splus SHOME`}/include  -O nnet.c
    ....
Splus LIBRARY nnet.a nnet_i.o nnet.o
nnet.a: No such file or directory
ld -r -o nnet_l.o nnet_i.o nnet.a
dynamically loadable file in nnet_l.o
$ make clean
rm -f nnet.o  nnet_i.o S_load_time.[oc] core
```

Replace `make dyn.load` by `make dyn.load.shared` if appropriate (or even `make static.load`). This will produce an object in the form expected by `dyn.load.lib`, but it will *not* write a `.First.lib` function (and many users forget to do so). The standard `Makefile` does use the name of the current directory as the section name, so on recent systems

```
.First.lib <- function(...) dyn.load.lib(...)
```

will suffice, unless the user moves the library section!

Even after the 'clean', unneeded files `Makefile`, `force_ld.q`, `nnet.a` and `nnet_i.c` are left.

The approach via `Splus CHAPTER` is slow compared to the use of a shell script. Trying this small example on a Sun SparcStation IPC took 125 seconds of CPU time against 60, but both times include 41 seconds to compile `nnet.c`.

Avoiding symbol clashes

It is necessary to check that the symbols exported by your C or FORTRAN code do not conflict with those already linked into S-PLUS. The simplest way is to use unique names (hence all the symbols in the libraries for our book begin with `VR_`). A formal check for potential conflicts can be done by

```
Splus NM myobj.o | cut -f 2-3 -d " " | grep "^T" \
  | cut -f 2 -d " " | sed 's/_$//' > t1
Splus NM `Splus SHOME`/cmd/Sqpe | cut -f 2-3 -d " " \
  | grep "^T" | cut -f 2 -d " " | sed 's/_$//' > t2
comm -12 t1 t2
```

on Unix. Note that we remove any trailing underscores from the symbol names, as on HP systems symbol names from FORTRAN will not have trailing underscores.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The NEW S Language*. New York: Chapman and Hall. (Formerly Monterey: Wadsworth and Brooks/Cole.). pages

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. New York: Springer. pages

Index

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