Writing R Extensions

Version 0.90.1

R Development Core Team

Table of Contents

Legalese

Acknowledgments

The contributions of Saikat DebRoy (who wrote the first draft of a guide to using `.Call` and `.External`) are gratefully acknowledged.

1 Creating R packages

Packages provide a mechanism for loading optional code and attached documentation as needed. The R distribution provides several packages, such as **eda**, **mva**, and **stepfun**.

A package consists of a subdirectory containing the files '`DESCRIPTION`', '`INDEX`', and '`TITLE`', and the subdirectories '`R`', '`data`', '`exec`', '`inst`', '`man`', and '`src`' (some of which can be missing).

The '`DESCRIPTION`' file contains basic information about the package in the following format:

```
Package: e1071
Version: 0.7-3
Author: Compiled by Fritz Leisch <Friedrich.Leisch@ci.tuwien.ac.at>.
Description: Miscellaneous functions used at the Department of
        Statistics at TU Wien (E1071).
Depends:
License: GPL version 2 or later
```

Continuation lines (e.g., for descriptions longer than one line) start with a whitespace character. The license field should contain an explicit statement or a well-known abbreviation (such as '`GPL`', '`LGPL`', '`BSD`', or '`Artistic`'), perhaps followed by a reference to the actual license file. It is very important that you include this information! Otherwise, it may not even be legally correct for others to distribute copies of the package.

The '`TITLE`' file contains a line giving the name of the package and a brief description. '`INDEX`' contains a line for each sufficiently interesting object in the package, giving its name and a description (functions such as print methods not usually called explicitly might not be included). Note that you can automatically create this file using something like `R CMD Rdindex man/*.Rd > INDEX`, provided that Perl is available on your system.

The '`R`' subdirectory contains R code files. The code files to be installed must start with a (lower- or uppercase) letter and have one of the extensions '`.R`', '`.S`', '`.q`', '`.r`', or '`.s`'. We recommend using '`.R`', as this extension seems to be not used by any other software. It should be possible to read in the files using `source()`, so R objects must be created by assignments. Note that there need be no connection between the name of the file and the R objects created by it. If necessary, one of these files (historically '`zzz.R`') should use `library.dynam()` *inside* `.First.lib()` to load compiled code.

The '`man`' subdirectory should contain R documentation files for the objects in the package. The documentation files to be installed must also start with a (lower- or uppercase) letter and have the extension '`.Rd`' (the default) or '`.rd`'.

The '`R`' and '`man`' subdirectories may contain OS-specific subdirectories named `unix`, `windows` or `mac`.

The C or FORTRAN source and optionally a '`Makefile`' for the compiled code is in '`src`'. Note that the '`Makefile`' most likely is not needed, and if one is to be distributed considerable care is needed to make it general enough to work on all R platforms.

The '`data`' subdirectory is for additional data files the package makes available for loading using `data()`. Currently, data files can have one of three types as indicated by their extension: plain R code ('`.R`' or '`.r`'), tables ('`.tab`', '`.txt`', or '`.csv`'), or `save()` images

('.RData' or '.rda'). The subdirectory should contain a '00Index' file that describes the datasets available.

The contents of the 'inst' subdirectory will be copied recursively to the installation directory.

Finally, 'exec' could contain additional executables the package needs, typically shell or Perl scripts. This mechanism is currently not used by any Unix package, and still experimental.

1.1 Package bundles

Sometimes it is convenient to distribute several packages as a *bundle*. (The main current example is VR which contains four packages.) The installation scripts on both Unix and Windows can handle package bundles as from R 0.90.1.

The 'DESCRIPTION' file of a bundle has an extra Bundle: field, as in

```
Bundle: VR
Contains: MASS class nnet spatial
Version: 6.1-6 (1999/11/26)
Author: S original by Venables & Ripley.
  R port by Brian Ripley <ripley@stats.ox.ac.uk>, following earlier
  work by Kurt Hornik and Albrecht Gebhardt.
BundleDescription: Various functions from the libraries of Venables and
  Ripley, 'Modern Applied Statistics with S-PLUS' (3rd edition).
License: GPL (version 2 or later)
```

The Contains: field lists the packages, which should be contained in separate subdirectories with the names given. These are standard packages in all respects except that the 'DESCRIPTION' file is replaced by a 'DESCRIPTION.in' file which just contains fields additional to the 'DESCRIPTION' file of the bundle, for example

```
Package: spatial
Description: Functions for kriging and point pattern analysis.
```

2 Writing R documentation

2.1 The documentation source tree

The help files containing detailed documentation for (potentially) all R objects are in the '`src/library/*/man`' subdirectories of the R source tree, where '`*`' stands for **base** where all the standard objects are, and for "standard" packages such as **eda** and **mva**. The '`doc/manual`' subdirectory contains code for running the translated help files through LaTeX and further documents pertaining to R.

2.2 Documentation format

The help files are written in a form and syntax—closely resembling TeX and LaTeX—which can be processed into a variety of formats, including LaTeX, [TN]roff, and HTML. The translation is carried out by the PERL script '`Rdconv`' in '`$R_HOME/bin`'.

For a given R function `myfun`, use the R command `prompt(myfun)` to produce the file '`myfun.Rd`', a "raw" documentation file that can now be filled in with information. The basic layout of such a file is as follows.

`\name{myfun}`
: *myfun* is the basename of the file.

`\alias{myfun}`
`\alias{`*more_aliases_1*`}`
`\alias{`*more_aliases_2*`}`
: etc. Need one `\alias{}` for each topic explained in the help file.

> **Note:** Each file should contain at least the `\alias{`*name*`}` line.

`\title{`*Title*`}`

`\description{...}`
: A short description of what the function(s) do(es) (one paragraph, a few lines only).

`\usage{myfun(`*arg1*`, `*arg2*`, ...)}`
: One or more lines showing the synopsis of the function(s) and variables documented in the file. These are set verbatim in typewriter font.

`\arguments{...}`
: Description of the function's arguments, in the following form:

> *Some optional text before the optional list.*
> `\item{`*arg1*`}{`*Description of arg1.*`}`
> `\item{`*arg2*`}{`*Description of arg2.*`}`
> etc.
> *Some optional text after the list.*

`\details{...}`
: A detailed if possible precise description of the functionality provided. Sometimes, precise `\references{}` can be given instead.

`\value{...}`
: Description of the function's return value. If a list with multiple values is returned, you can use

  `\item{comp1}{`*Description of result component 'comp1'*`}`
  `\item{comp2}{`*Description of result component 'comp2'*`}`

  etc.

`\references{...}`
: Section of references to the literature; use `\url{}` for web pointers. Optional as well as all the following sections.

`\section{`*name*`}{`*text*`}`
: and maybe more `\section{}` environments.

`\note{`*Some note you want to have pointed out.*`}`

`\author{...}`
: Who you are. Use `\email{}` without extra delimiters ('`( )`' or '`< >`') or `\url{}`.

`\seealso{...}`
: Pointers to related R objects, using `\link{}`, usually as `\code{\link{}}`.

`\examples{...}`
: Examples of how to use the function. These are set verbatim in typewriter font.

  > **Note:** Use examples which are *directly* executable! Use random number generators (e.g., `x <- rnorm(100)`), or a standard data set loadable via `data(...)` (see `data()` for info) to define data!

  By default, text inside `\examples{}` will be displayed in the output of the help page and run by *`make check`*. You can use `\dontrun{}` for commands that should only be shown, but not run, and `\testonly{}` for extra commands for testing R that should not be shown to users.

  For example,

`x <- runif(10)`	Shown and run.
`\dontrun{plot(x)}`	Only shown.
`\testonly{log(x}`	Only run.

`\keyword{`*key_1*`}`
`\keyword{`*key_2*`}`
: Use at least one keyword from the list in '`$R_HOME/doc/KEYWORDS`'.

2.3 Sectioning

To begin a new paragraph or leave a blank in an example, just insert an empty line (as in (La)TeX). To break a line, use `\cr`.

In addition to the predefined sections (such as `\description{}`, `\value{}`, etc.), you can "define" arbitrary ones by `\section{`*section_title*`}{...}`. E.g.,

```
\section{Warning}{You must not call this function unless ...}
```

Note that the additonal named sections are always inserted at fixed positions in the output (before `\note`, `\seealso` and the examples), no matter where they appear in the input.

2.4 Marking text

The following logical markup commands are available for indicating specific kinds of text.

`\bold{`*word*`}`	set *word* in **bold** font if possible
`\emph{`*word*`}`	emphasize *word* using *italic* font if possible
`\code{`*word*`}`	for pieces of code, using `typewriter` font if possible
`\file{`*word*`}`	for file names
`\email{`*word*`}`	for email addresses
`\url{`*word*`}`	for URLs

The first two, `\bold` and `\emph`, should be used in plain text for emphasis.

Fragments of R code, including the names of R objects, should be marked using `\code`. Only backslashes and percent signs need to be escaped (by a backslash) inside `\code`.

Finally, `\link{`*foo*`}` (usually in the combination `\code{\link{`*foo*`}}`) produces a hyperlink to the help page for object *foo*. One main usage of `\link` is in the `\seealso` section of the help page, see Section 2.2 [Documentation format], page 4, above. (This only affects the creation of hyperlinks, for example in the HTML pages used by `help.start()`.)

2.5 Mathematics

Mathematical formula are something we want "perfectly" for printed documentation (i.e., for the conversion to LaTeX and PostScript subsequently) and still want something useful for ASCII and HTML online help.

To this end, the two commands `\eqn{`*latex*`}{`*ascii*`}` and `\deqn{`*latex*`}{`*ascii*`}` are used. Where `\eqn` is used for "inline" formula (corresponding to (La)TeX's `$...$`, `\deqn` gives "displayed equations" (a la LaTeX's `displaymath` environment, or TeX's `$$...$$`).

Both commands can also be used as `\eqn{`*latexascii*`}` (only *one* argument) which then is used for both *latex* and *ascii*.

The following example is from the `Poisson` help page:

```
\deqn{p(x) = {\lambda^x\ \frac{e^{-\lambda}}{x!}}
     {p(x) = lambda^x exp(-lambda)/x!}
for \eqn{x = 0, 1, 2, ...}.
```

For the LaTeX manual, this becomes

$$p(x) \;=\; \lambda^x \, \frac{e^{-\lambda}}{x!}$$

for $x \;=\; 0,\; 1,\; 2,\; \ldots$.

For the HTML and the "direct" (man-like) on-line help we get

```
     p(x) = lambda^x exp(-lambda)/x!

for x = 0, 1, 2, ....
```

For historic reasons mostly, the TeX/LaTeX commands `\alpha`, `\Alpha`, `\beta`, `\Gamma`, `\epsilon`, `\lambda`, `\mu`, `\pi`, `\sigma`, `\left(` and `\right)` exist. These can be used directly, without using the `\eqn` diversion.

2.6 Miscellaneous

Use `\R` for the R system itself (you don't need extra '`{}`' or '`\`'). Use `\dots` for the dots in function argument lists '`...`', and `\ldots` for ellipsis dots.

After a '`%`', you can put your own comments regarding the help text. This will be completely disregarded, normally. Therefore, you can also use it to make part of the "help" invisible.

Escaping Special Characters. You can produce a backslash ('\' by escaping it by another backslash. (Note that `\cr` is used for generating line breaks.)

The "comment" and "control" characters '`%`' and '`\`' *always* need to be escaped. Inside the verbatim-like commands (`\code` and `\examples`), no other characters are special.

In "regular" text (no verbatim, no `\eqn`, ...), you currently must escape all LaTeX special characters, i.e., besides '`%`', '`{`', and '`}`', the four specials '`$`', '`&`', '`#`', and '`_`' are produced by preceding each with a '`\`'. Further, enter '`^`' as `\eqn{\hat{}}{^}`, and '`~`' by `\eqn{\tilde{}}{~}`. Also, '`<`', '`>`', and '`|`' must only be used in math mode, i.e., within `\eqn` or `\deqn`.

2.7 Platform-specific documentation

Sometimes the documentation needs to differ by platform. Currently three OS-specific options are available, `unix`, `windows` and `mac`, and lines in the help source file can be enclosed in

```
#ifdef OS
   ...
#endif
```

or

```
#ifndef OS
   ...
#endif
```

for OS-specific inclusion or exclusion.

If the differences between platforms are extensive or the R objects documented are only relevant to one platform, platform-specific `.Rd` files can be put in a '`unix`', '`windows`' or '`mac`' subdirectory.

3 System and foreign language interfaces

3.1 Operating system access

Access to operating system functions is via the R function `system`. The details will differ by platform (see the on-line help), and about all that can safely be assumed is that the first argument will be a string `command` that will be passed for execution (not necessarily by a shell) and the second argument will be `internal` which if true will collect the output of the command into an R character vector.

The function `system.time` is available for timing (although the information available may be limited on non-Unix-like platforms).

3.2 Interface functions `.C` and `.Fortran`

These two functions provide a standard interface to compiled code that has been linked into R, either at build time or via `dyn.load` (q.v.). They are primarily intended for compiled C and Fortran code respectively, but the `.C` function can be used with other languages which can generate C interfaces, for example C++.

The first argument to each function is a character string given the symbol name as known to C or Fortran, that is the function or subroutine name. (The mapping to the symbol name in the load table is given by the functions `symbol.C` and `symbol.For`; that the symbol is loaded can be tested by, for example, `is.loaded(symbol.C("loglin"))`.)

There can be up to 65 further arguments giving R objects to be passed to compiled code. Normally these are copied before being passed in, and copied again to an R list object when the compiled code returns. If the arguments are given names, these are used as names for the components in the returned list object (but not passed to the compiled code).

The following table gives the mapping between the modes of R vectors and the types of arguments to a C function or Fortran subroutine.

R storage mode	C type	Fortran type
`logical`	`int *`	`INTEGER`
`integer`	`int *`	`INTEGER`
`double`	`double *`	`DOUBLE PRECISION`
`complex`	`Rcomplex *`	`DOUBLE COMPLEX`
`character`	`char **`	`CHARACTER*255`

C type `Rcomplex` is a structure with `double` members `r` and `i` defined in the header file '`Complex.h`'. Only a single character string can be passed to or from Fortran, and the success of this is compiler-dependent. Other R objects can be passed to `.C`, but it is better to use one of the other interfaces. An exception is passing an R function for use with `call_R`, when the object can be handled as `void *` en route to `call_R`, but even there `.Call` is to be preferred.

It is possible to pass numeric vectors of storage mode `double` to C as `float *` or Fortran as `REAL` by setting the attribute `Csingle`, most conveniently by using the R functions `as.single`, `single` or `storage.mode`. This is intended only to be used to aid interfacing to existing C or Fortran code.

Unless formal argument NAOK is true, all the other arguments are checked for missing values NA and for the IEEE special values NaN, Inf and -Inf, and the presence of any of these generates an error. If it is true, these values are passed unchecked.

Argument DUP can be used to suppress copying. It is dangerous: see the on-line help for arguments against its use. It is not possible to pass numeric vectors as float * or REAL if DUP=TRUE.

Finally, argument PACKAGE confines the search for the symbol name to a specific shared library (or use "base" for code compiled into R). Its use is highly desirable, as there is no way to avoid two package writers using the same symbol name, and such name clashes are normally sufficient to cause R to crash.

Note that the compiled code should not return anything except through its arguments: C functions should be of type void and Fortran subprograms should be subroutines.

To fix ideas, let us consider a very simple example which convolves two finite sequences. (This is hard to do fast in interpreted R code, but easy in C code.) We could do this using .C by

```
void convolve(double *a, int *na, double *b, int *nb, double *ab)
{
  int i, j, nab = *na + *nb - 1;

  for(i = 0; i < nab; i++) ab[i] = 0.0;
  for(i = 0; i < *na; i++)
    for(j = 0; j < *nb; j++) ab[i + j] += a[i] * b[j];
}
```

called from R by

```
conv <- function(a, b)
  .C("convolve", as.double(a), as.integer(length(a)),
     as.double(b), as.integer(length(b)),
     ab = double(length(a) + length(b) - 1))$ab
```

Note that we take care to coerce all the arguments to the correct R storage mode before calling .C; mistakes in matching the types can lead to wrong results or hard-to-catch errors.

3.3 dyn.load and dyn.unload

Compiled code to be used with R is loaded as a shared library (Unix) or DLL (Windows). The library/DLL is loaded by dyn.load and unloaded by dyn.unload. Unloading is not normally necessary, but it is needed to allow the DLL to be re-built on some platforms, including Windows.

The first argument to both functions is a character string giving the path to library. Programmers should not assume a specific file extension for the library (such as .so) but use a construction like

```
file.path(path1, path2, paste("mylib", .Platform$dynlib.ext, sep=""))
```

for platform independence. On Unix systems the path supplied to dyn.load can be an absolute path, one relative to the current directory or, if it starts with ~, relative to the user's home directory.

Loading is most often done via a call to `library.dynam` in the `.First.lib` function of a package. This has the form

```
library.dynam("libname", package, lib.loc)
```

where `libname` is the library/DLL name with the extension omitted.

Under some Unix systems there is a choice of how the symbols are resolved when the library is loaded, governed by the arguments `local` and `now`. Only use these if really necessary: in particular using `now=FALSE` and then calling an unresolved symbol will terminate R unceremoniously.

If a library/DLL is loaded more than once the most recent version is used. More generally, if the same symbol name appears in several libraries, the most recently loaded occurrence is used. The `PACKAGES` argument provides a good way to avoid any ambiguity in which occurrence is meant.

3.4 Interfacing C++ code

(Contributed by *Adrian Trapletti*.)

Suppose we have the following hypothetical C++ library, consisting of the two files '`X.hh`' and '`X.cc`', which we want to use in R:

```
// X.hh

class X
{
public:
  X ();
  ~X ();
};

class Y
{
public:
  Y ();
  ~Y ();
};
```

```
// X.cc

#include <iostream.h>
#include "X.hh"

static Y y;

X::X()  { cout << "constructor X" << endl; }
X::~X() { cout << "destructor X" << endl; }
Y::Y()  { cout << "constructor Y" << endl; }
Y::~Y() { cout << "destructor Y" << endl; }
```

implementing the 2 classes `X` and `Y`. The only thing we have to do is writing a wrapper function and ensuring that the function is enclosed in

```
extern "C" {

}
```

For example,

```
// X_main.cc:

#include "X.hh"

extern "C" {

void X_main ()
{
  X x;
}

}
```

Compiling and linking should be done with the C++ compiler-linker. For example, under Linux we might use

```
g++ -c X.cc
g++ -c X_main.cc
g++ -shared -o X.so X_main.o X.o
```

Otherwise (i.e., linking, e.g., with GNU ld) the C++ initialization code (and hence the constructor of the static variable `Y`) are not called.

Now starting R yields

```
R: Copyright 1999, The R Development Core Team
Version 0.63.2  (January 12, 1999)
...
Type    "q()" to quit R.

R> dyn.load("X.so")
constructor Y
```

```
R> .C("X_main")
constructor X
destructor X
R> q()
Save workspace image? [y/n/c]: y
destructor Y
```

3.5 Handling R objects in C

Using C code to speed up the execution of an R function is often very fruitful. Traditionally this has been done via the `.C` function in R. One restriction of this interface is that the R objects can not be handled directly in C. This becomes more troublesome when one wishes to call R functions from within the C code. There is a C function provided called `call_R` (also known as `call_S` for compatibility with S) that can do that, but it is cumbersome to use, and the mechanisms documented here are usually simpler to use, as well as more powerful.

If a user really wants to write C code using internal R data structures, then that can be done using the `.Call` and `.External` function. The syntax for the calling function in R in each case is similar to that of `.C`, but the two functions have rather different C interfaces. Generally the `.Call` interface (which is modelled on the interface of the same name in S version 4) is a little simpler to use, but `.External` is a little more general.

A call to `.Call` is very similar to `.C`, for example

```
.Call("convolve2", a, b)
```

The first argument should be a character string giving a C symbol name of code that has already been loaded into R. Up to 65 R objects can passed as arguments. The C side of the interface is

```
#include <S.h>

SEXP convolve2(SEXP a, SEXP b)
 ...
```

A call to `.External` is almost identical

```
.External("convolveE", a, b)
```

but the C side of the interface is different, having only one argument

```
#include <Rinternals.h>

SEXP convolveE(SEXP args)
 ...
```

Here `args` is a `LISTSXP`, a Lisp-style list from which the arguments can be extracted.

In each case the R objects are available for manipulation via a set of functions and macros defined in the header file '`Rinternals.h`' or some higher-level macros defined in '`Rdefines.h`' (included by `S.h`). Details on `.Call` and `.External` are given further below.

Before you decide to use `.Call` or `.External`, you should look at other alternatives. First, consider working in interpreted R code; if this is fast enough, this is normally the best option. You should also see if using `.C` is enough. If the task to be performed in C is simple enough requiring no call to R, `.C` suffices. The new interfaces are recent additions to S and R, and a great deal of useful code has been written using just `.C` before they were available. The `.Call` and `.External` interfaces allow much more control, but they also impose much greater responsibilities so need to be used with care.

There are two approaches that can be taken to handling R objects from within C code. The first (historically) is to use the macros and functions that have been used to implement the core parts of R through `.Internal` calls. A public subset of these is defined in the header file '`Rinternals.h`' in the directory '`$R_HOME/include`' that should be available on any R installation.

A more recent approach is to use R versions of the macros and functions defined for the S version 4 interface `.Call`, which are defined in the header file '`Rdefines.h`', included by '`S.h`'. This is a somewhat simpler approach, and is certainly to be preferred if the code might be shared with S at any stage.

A substantial amount of R is implemented using the functions and macros described here, so the R source code provides a rich source of examples and "how to do it':' indeed many of the examples here were developed by examining closely R system functions for similar tasks. Do make use of the source code for inspirational examples.

It is necessary to know something about how R objects are handled in C code. All the R objects you will deal with will be handled with the type *SEXP*[1], which is a pointer to a structure. Think of this structure as a *variant type* that can handle all the usual types of R objects, that is vectors of various modes, functions, environments, language objects and so on. The details are given later in this section, but for most purposes the programmer does not need to know them. Think rather of a model such as that used by Visual Basic, in which R objects are handed around in C code (as they are in interpreted R code) as the variant type, and the appropriate part is extracted for, for example, numerical calculations, only when it is needed. As in interpreted R code, much use is made of coercion to force the variant object to the right type.

3.5.1 Handling the effects of garbage collection

We need to know a little about the way R handles memory allocation. The memory allocated for R objects is not freed by the user; instead, the memory is from time to time *garbage collected*. That is, all the allocated memory not being used is freed, and the objects that are in use may be moved. If you create an R object in your C code, you must tell R that you are using the object via call to the `PROTECT` macro. This has two effects. First it tells R that the object is in use so it is not destroyed. Second, it ensures that the `SEXP` pointer to the object is updated if the object's structure is moved in memory during garbage collection. (Because of this it is not safe to save and re-use pointers to parts of an object's structure.)

The programmer is solely responsible for housekeeping the calls to `PROTECT`. There is a corresponding macro `UNPROTECT` that takes as argument an `int` giving the number of `SEXP`s to unprotect when they are no longer needed. The protection mechanism is stack-based, so `UNPROTECT(n)` unprotects the last `n` objects which were protected. The calls to `PROTECT` and `UNPROTECT` must balance when the user's code returns, even if it returns because of an

[1] SEXP is an acronym for *S*imple *EXP*ression, common in LISP-like language syntaxes.

error (calling `error` or `errorcall` for example). R will warn about `stack imbalance in .Call` (or `.External`) if the housekeeping is wrong.

Here is a small example of creating an R numeric vector in C code. First we use the macros in '`Rdefines.h`':

```
#include <Rdefines.h>

  SEXP ab;
    ....
  PROTECT(ab = NEW_NUMERIC(2));
  NUMERIC_POINTER(ab)[0] = 123.45;
  NUMERIC_POINTER(ab)[1] = 67.89;
  UNPROTECT(1);
```

and then those in '`Rinternals.h`':

```
#include <Rinternals.h>

  SEXP ab;
    ....
  PROTECT(ab = allocVector(REALSXP, 2));
  REAL(ab)[0] = 123.45; REAL(ab)[1] = 67.89;
  UNPROTECT(1);
```

Now, the reader may ask how the R object could possibly get removed during those manipulations, as it is just our C code that is running. The answer is we do not know (nor want to know) what is hiding behind the R macros and functions we use, and any of them might cause memory to be allocated, hence garbage collection and hence our object `ab` to be (re)moved. It is wise to err on the side of caution and assume that any of the R macros and functions might (re)move the object.

Protection is not needed for objects which R already knows are in use. In particular, this applies to function arguments.

There is a less-used[2] macro `UNPROTECT_PTR(s)` that unprotects the `SEXP` s.

3.5.2 Allocating storage

For many purposes it is sufficient to allocate R objects and manipulate those. There are quite a few `allocXxx` functions defined in '`Rinternals.h`' — you may want to explore them. These allocate R objects of various types, and for the standard vector types there are `NEW_XXX` macros defined in '`Rdefines.h`'.

If storage is required for C objects during the calculations this is best allocating by calling `R_alloc`; such storage is automatically released once the call to `.C` or `.Call` or `.External` returns. `R_alloc` is defined in '`Memory.h`' (included by '`Rinternals.h`' and by '`Rdefines.h`') as

```
char*  R_alloc(long, int);
```

so a typical usage is

```
v = (double*) R_alloc(nlag, sizeof(double));
```

[2] for historical reasons, `UNPROTECT_PTR(s)` has existed only since R version 0.63.0 (Nov. 1998)

Memory allocated by `R_alloc` is not zeroed, but the related function `S_alloc` calls `R_alloc` and then zeroes the memory.

All of these memory allocation routines do their own error-checking, so the programmer may assume that they will raise an error and not return if the memory cannot be allocated.

3.5.3 Details of R types

Users of the '`Rinternals.h`' macros will need to know how the R types are known internally: this is more or less completely hidden if the '`Rdefines.h`' macros are used.

The different R data types are represented in C by *SEXPTYPE*. Some of these are familiar from R and some are internal data types. The usual R object modes are given in the table.

SEXPTYPE	**R equivalent / explanation**
`REALSXP`	numeric with storage mode `double`
`INTSXP`	integer
`CPLXSXP`	complex
`LGLSXP`	logical
`STRSXP`	character
`VECSXP`	list (generic vector)
`LISTXP`	"dotted-pair" list
`DOTSXP`	a ... object
`NILSXP`	NULL
`SYMSXP`	name/symbol
`CLOSXP`	function or function closure
`ENVSXP`	environment

Among the important internal `SEXPTYPE`s are `LANGSXP`, `CHARSXP` etc.

Unless you are very sure about the type of the arguments, the code should check the data types. Sometimes it may also be necessary to check data types of objects created by evaluating an R expression in the C code. You can use functions like `isReal`, `isInteger` and `isString` to do type checking. See the header file '`Rinternals.h`' for definitions of other such functions. All of these take a `SEXP` as argument and return 1 or 0 to indicate *TRUE* or *FALSE*. Once again there are two ways to do this, and '`Rdefines.h`' has macros such as `IS_NUMERIC`.

What happens if the `SEXP` is not of the correct type? Sometimes you have no other option except to generate an error. You can use the function `error` for this. It is usually better to coerce the object to the correct type. For example, if you find that an `SEXP` is of the type `INTEGER`, but you need a `REAL` object, you can change the type by using, equivalently,

```
PROTECT(newSexp = coerceVector(oldSexp, REALSXP));
```

or

```
PROTECT(newSexp = AS_NUMERIC(oldSexp));
```

Protection is needed as a new *object* is created; the object formerly pointed to by the `SEXP` is re-used is still protected but now unused.

All the coercion functions do their own error-checking, and generate `NA`s with a warning or stop with an error as appropriate.

So far we have only seen how to create and coerce R objects from C code, and how to extract the numeric data from numeric R vectors. These can suffice to take us a long way in interfacing R objects to numerical algorithms, but we may need to know a little more to create useful return objects.

3.5.4 Attributes

Many R objects have attributes: some of the most useful are classes and the `dim` and `dimnames` that mark objects as matrices or arrays. It can also be helpful to work with the `names` attribute of vectors.

To illustrate this, let us write code to take the outer product of two vectors (which `outer` and `%o%` already do). As usual the R code is simple

```
out <- function(x, y) .Call("out", as.double(x), as.double(y))
```

where we expect `x` and `y` to be numeric vectors, possibly with names. This time we do the coercion in the calling R code.
C code to do the computations is

```
#include <Rinternals.h>

SEXP out(SEXP x, SEXP y)
{
  int i, j, nx, ny;
  double tmp;
  SEXP ans;

  nx = length(x); ny = length(y);
  PROTECT(ans = allocMatrix(REALSXP, nx, ny));
  for(i = 0; i < nx; i++) {
    tmp = REAL(x)[i];
    for(j = 0; j < ny; j++)
      REAL(ans)[i + nx*j] = tmp * REAL(y)[j];
  }
  UNPROTECT(1);
  return(ans);
}
```

but we would like to set the `dimnames` of the result. Although `allocMatrix` provides a short cut, we will show how to set the `dim` attribute directly.

```
#include <Rinternals.h>

SEXP out(SEXP x, SEXP y)
{
  int i, j, nx, ny;
  double tmp;
  SEXP ans, dim, dimnames;

  nx = length(x); ny = length(y);
  PROTECT(ans = allocVector(nx*ny, REALSXP));
  for(i = 0; i < nx; i++) {
    tmp = REAL(x)[i];
    for(j = 0; j < ny; j++)
```

```
            REAL(ans)[i + nx*j] = tmp * REAL(y)[j];
      }
      PROTECT(dim = allocVector(INTSXP, 2));
      INTEGER(dim)[0] = nx; INTEGER(dim)[1] = ny;
      setAttrib(ans, R_DimSymbol, dim);

      PROTECT(dimnames = allocVector(VECSXP, 2));
      VECTOR(dimnames)[0] = getAttrib(x, R_NamesSymbol);
      VECTOR(dimnames)[1] = getAttrib(y, R_NamesSymbol);
      setAttrib(ans, R_DimNamesSymbol, dimnames);
      UNPROTECT(3);
      return(ans);
    }
```

This example introduces several new features. The `getAttrib` and `setAttrib` functions get and set individual attributes. Their second argument is a `SEXP` defining the name in the symbol table of the attribute we want; these and many such symbols are defined in the header file '`Rinternals.h`'.

There are shortcuts here too: the functions `namesgets`, `dimgets` and `dimnamesgets` are the internal versions of `names<-`, `dim<-` and `dimnames<-`, and there are functions such as `GetMatrixDimnames` and `GetArrayDimnames`.

What happens if we want to add an attribute that is not pre-defined? We need to add a symbol for it *via* a call to `install`. Suppose for illustration we wanted to add an attribute `"version"` with value `3.0`. We could use

```
      { SEXP version;
      PROTECT(version = allocVector(REALSXP, 1));
      REAL(vector) = 3.0;
      setAttrib(ans, install("version"), version);
      UNPROTECT(1);
      }
```

Using `install` when it is not needed is harmless and provides a simple way to retrieve the symbol from the symbol table if it is already installed.

3.5.5 Classes

In R the class is just the attribute named `"class"` so it can be handled as such, but there is a shortcut `classgets`. Suppose we want to give the return value in our example the class `"mat"`. We can use

```
    #include <Rdefines.h>
        ....
      SEXP ans, dim, dimnames, class;
        ....
      PROTECT(class = allocVector(STRSXP, 1));
      STRING(class)[0] = COPY_TO_USER_STRING("mat");
      classgets(ans, class);
      UNPROTECT(4);
      return(ans);
    }
```

As the value is a character vector, we have to know how to create that from a C character array, which we do using the macro `COPY_TO_USER_STRING` defined in '`Rdefines.h`'.

3.5.6 Handling lists

Some care is needed with lists, as R has moved from using LISP-like lists (now called 'pairlists') to S-like generic vectors. As a result, the appropriate test for a object of mode `list` is `isNewList`, and we need `allocVector(VECSXP, n)` and *not* `allocList(n)`.

List elements can be retrieved or set by direct access to the elements of the generic vector. Suppose we have a list object

```
a <- list(f=1, g=2, h=3)
```

Then we can access `a$g` as `a[[2]]` by

```
  double g;
    ....
  g = REAL(VECTOR(a)[1])[0];
```

This can rapidly become tedious, and the following function (based on one in package `nls`) is very useful:

```
/* get the list element named str, or return NULL */

SEXP getListElement(SEXP list, char *str)
{
  SEXP elmt = R_NilValue, names = getAttrib(list, R_NamesSymbol);
  int i;

  for (i = 0; i < length(list); i++)
    if(strcmp(CHAR(STRING(names)[i]), str) == 0) {
      elmt = VECTOR(list)[i];
      break;
    }
  return elmt;
}
```

and enables us to say

```
  double g;
  g = REAL(getListElement(a, "g"))[0];
```

3.5.7 Finding and setting variables

It will be usual that all the R objects needed in our C computations are passed as arguments to `.Call` or `.External`, but it is possible to find the values of R objects from within the C given their names. The following code is the equivalent of `get(name, envir = rho)`.

```
SEXP getvar(SEXP name, SEXP rho)
{
  SEXP ans;

  if(!isString(name) || length(name) != 1)
     error("name is not a single string");
  if(!isEnvironment(rho))
    error("rho should be an environment");
  ans = findVar(install(CHAR(STRING(name)[0])), rho);
  printf("first value is %f\n", REAL(ans)[0]);
  return(R_NilValue);
}
```

The main work is done by `findVar`, but to use it we need to install `name` as a name in the symbol table. As we wanted the value for internal use, we return `NULL`.

Similar functions with syntax

```
void defineVar(SEXP symbol, SEXP value, SEXP rho)
void setVar(SEXP symbol, SEXP value, SEXP rho)
```

can be used to assign values to R objects, in the specified environment frame and to perform the equivalent of `assign(x, value, envir = rho, inherits = TRUE)` respectively.

3.6 Interface Functions `.Call` and `.External`

In this section we consider the details of the R/C interfaces.

These two interfaces have almost the same functionality. `.Call` is based on the interface of the same name in S version 4, and `.External` is based on `.Internal`. `.External` is more complex but allows a variable number of arguments.

3.6.1 Calling `.Call`

Let us convert our finite convolution example to use `.Call`, first using the '`Rdefines.h`' macros. The calling function in R is

```
conv <- function(a, b) .Call("convolve2", a, b)
```

which could hardly be simpler, but as we shall see all the type checking must be transferred to the C code, which is

```
#include <S.h>

SEXP convolve2(SEXP a, SEXP b)
{
  int i, j, na, nb, nab;
  double *xa, *xb, *xab;
  SEXP ab;

  PROTECT(a = AS_NUMERIC(a));
  PROTECT(b = AS_NUMERIC(b));
  na = LENGTH(a); nb = LENGTH(b); nab = na + nb - 1;
  PROTECT(ab = NEW_NUMERIC(nab));
  xa = NUMERIC_POINTER(a); xb = NUMERIC_POINTER(b);
  xab = NUMERIC_POINTER(ab);
  for(i = 0; i < nab; i++) xab[i] = 0.0;
  for(i = 0; i < na; i++)
    for(j = 0; j < nb; j++) xab[i + j] += xa[i] * xb[j];
  UNPROTECT(3);
  return(ab);
}
```

Note that unlike the macros in S version 4, the R versions of these macros do check that coercion can be done and raise an error if it fails. They will raise warnings if missing values are introduced by coercion. Although we illustrate doing the coercion in the C code here, it often is simpler to do the necessary coercions in the R code.

Now for the version in R-internal style. Only the C code changes.

```
#include <Rinternals.h>

SEXP convolve2(SEXP a, SEXP b)
{
  int i, j, na, nb, nab;
  double *xa, *xb, *xab;
  SEXP ab;

  PROTECT(a = coerceVector(a, REALSXP));
  PROTECT(b = coerceVector(b, REALSXP));
  na = length(a); nb = length(b); nab = na + nb - 1;
  PROTECT(ab = allocVector(nab, REALSXP));
  xa = REAL(a); xb = REAL(b);
  xab = REAL(ab);
  for(i = 0; i < nab; i++) xab[i] = 0.0;
  for(i = 0; i < na; i++)
    for(j = 0; j < nb; j++) xab[i + j] += xa[i] * xb[j];
  UNPROTECT(3);
  return(ab);
}
```

This is called in exactly the same way.

3.6.2 Calling `.External`

We can use the same example to illustrate `.External`. The R code changes only by replacing `.Call` by `.External`

```
conv <- function(a, b) .External("convolveE", a, b)
```

but the main change is how the arguments are passed to the C code, this time as a single SEXP. The only change to the C code is how we handle the arguments.

```
#include <Rinternals.h>

SEXP convolveE(SEXP args)
{
  int i, j, na, nb, nab;
  double *xa, *xb, *xab;
  SEXP a, b, ab;

  PROTECT(a = coerceVector(CADR(args), REALSXP)));
  PROTECT(b = coerceVector(CADDR(args), REALSXP)));
    ...
}
```

Once again we do not need to protect the arguments, as in the R side of the interface they are objects that are already in use. The macros

```
first = CADR(args);
second = CADDR(args);
third = CADDDR(args);
fourth = CAD4R(args);
```

provide convenient ways to access the first four arguments. More generally we can use the `CDR` and `CAR` macros as in

```
args = CDR(args); a = CAR(args);
args = CDR(args); b = CAR(args);
```

which clearly allows us to extract an unlimited number of arguments (whereas `.Call` has a limit, albeit at 65 not a small one).

More usefully, the `.External` interface provides an easy way to handle calls with a variable number of arguments, as `length(args)` will give the number of arguments supplied (of which the first is ignored). We may need to know the names ('tags') given to the actual arguments, which we can by using the `TAG` macro and using something like the following example, that print the names and the first value of its arguments if they are vector types.

```
SEXP showArgs(SEXP args)
{
  int i, nargs;
  Rcomplex cpl;
  char *name;

  if((nargs = length(args) - 1) > 0) {
    for(i = 0; i < nargs; i++) {
      args = CDR(args);
      name = CHAR(PRINTNAME(TAG(args)));
      switch(TYPEOF(CAR(args))) {
      case REALSXP:
        printf("[%d] '%s' %f\n", i+1, name, REAL(CAR(args))[0]);
      break;
      case LGLSXP:
      case INTSXP:
        printf("[%d] '%s' %d\n", i+1, name, INTEGER(CAR(args))[0]);
      break;
      case CPLXSXP:
        cpl = COMPLEX(CAR(args))[0];
        printf("[%d] '%s' %f + %fi\n", i+1, name, cpl.r, cpl.i);
        break;
      case STRSXP:
        printf("[%d] '%s' %s\n", i+1, name,
               CHAR(STRING(CAR(args))[0]));
        break;
      default:
        printf("[%d] '%s' R type\n", i+1, name);
      }
    }
  }
  return(R_NilValue);
}
```

This can be called by the wrapper function

```
showArgs <- function(...) .External("showArgs", ...)
```

Note that this style of programming is convenient but not necessary, as an alternative style is

```
showArgs <- function(...) .Call("showArgs1", list(...))
```

3.6.3 Missing and special values

One piece of error-checking the `.C` call does (unless `NAOK` is true) is to check for missing (`NA`) and IEEE special values (`Inf`, `-Inf` and `NaN`) and give an error if any are found. With the `.Call` interface these will be passed to our code. In this example the special values are no problem, as IEEE arithmetic will handle them correctly. In the current implementation this is also true of `NA` as it is a type of `NaN`, but it is unwise to rely on such details. Thus we will re-write the code to handle `NA`s using macros defined in '`Arith.h`'.

The code changes are the same in any of the versions of `convolve2` or `convolveE`:

```
      ...
    for(i = 0; i < na; i++)
      for(j = 0; j < nb; j++)
          if(ISNA(xa[i]) || ISNA(xb[j]) || ISNA(xab[i + j]))
            xab[i + j] = NA_REAL;
          else
            xab[i + j] += xa[i] * xb[j];
      ...
```

Note that the `ISNA` macro, and the similar macros `ISNAN` (which checks for `NaN` or `NA`) and `R_FINITE` (which is false for `NA` and all the special values), only apply to numeric values of type `double`. Missingness of integers, logicals and character strings can be tested by equality to the constants `NA_INTEGER`, `NA_LOGICAL` and `NA_STRING`. These and `NA_REAL` can be used to set elements of R vectors to `NA`.

The constants `R_NaN`, `R_PosInf`, `R_NegInf` and `R_NaReal` can be used to set `double`s to the special values.

3.7 Evaluating R Expressions from C

We noted that the `call_R` interface could be used to evaluate R expressions from C code, but the current interfaces are much more convenient to use. The main function we will use is

```
SEXP eval(SEXP expr, SEXP rho);
```

the equivalent of the interpreted R code `eval(expr, envir = rho)`, although we can also make use of `findVar`, `defineVar` and `findFun` (which restricts the search to functions).

To see how this might be applied, here is a simplified internal version of `lapply` for expressions, used as

```
a <- list(a = 1:5, b = rnorm(10), test = runif(100))
.Call("lapply", a, quote(sum(x)), new.env())
```

with C code

```
SEXP lapply(SEXP list, SEXP expr, SEXP rho)
{
  int i, n = length(list);
  SEXP ans;

  if(!isNewList(list)) error("‘list’ must be a list");
  if(!isEnvironment(rho)) error("‘rho’ should be an environment");
  PROTECT(ans = allocVector(VECSXP, n));
  for(i = 0; i < n; i++) {
    defineVar(install("x"), VECTOR(list)[i], rho);
    VECTOR(ans)[i] = eval(expr, rho);
  }
  setAttrib(ans, R_NamesSymbol, getAttrib(list, R_NamesSymbol));
  UNPROTECT(1);
  return(ans);
}
```

It would be closer to `lapply` if we could pass in a function rather than an expression. One way to do this is *via* interpreted R code as in the next example, but it is possible (if somewhat obscure) to do this in C code. The following is based on the code in ‘`src/main/optimize.c`’.

```
SEXP lapply2(SEXP list, SEXP expr, SEXP rho)
{
  int i, n = length(list);
  SEXP R_fcall, s, ans;

  if(!isNewList(list)) error("‘list’ must be a list");
  if(!isFunction(fn)) error("‘fn’ must be a function");
  if(!isEnvironment(rho)) error("‘rho’ should be an environment");
  PROTECT(R_fcall = lang2(fn, R_NilValue));
  PROTECT(ans = allocVector(VECSXP, n));
  for(i = 0; i < n; i++) {
    CADR(R_fcall) = VECTOR(list)[i];
    VECTOR(ans)[i] = eval(expr, rho);
  }
  setAttrib(ans, R_NamesSymbol, getAttrib(list, R_NamesSymbol));
  UNPROTECT(1);
  return(ans);
}
```

used by

```
.Call("lapply2", a, sum, new.env())
```

Function `lang2` creates an executable ‘list’ of two elements, but this will only be clear to those with a knowledge of a LISP-like language.

3.7.1 Zero-finding

In this section we re-work the example in ‘`demos/dynload`’ of `call_R` (based on that for `call_S` in Becker, Chambers & Wilks (1988)) on finding a zero of a univariate function. The R code and an example are

```
zero <- function(f, guesses, tol = 1e-7) {
  f.check <- function(x) {
    x <- f(x)
    if(!is.numeric(x)) stop("Need a numeric result")
    as.double(x)
  }
  .Call("zero", body(f.check), as.double(guesses), as.double(tol),
        new.env())
}

cube1 <- function(x) (x^2 + 1) * (x - 1.5)
zero(cube1, c(0, 5))
```

where this time we do the coercion and error-checking in the R code. The C code is

```
SEXP mkans(double x)
{
    SEXP ans;
    PROTECT(ans = allocVector(REALSXP, 1));
    REAL(ans)[0] = x;
    UNPROTECT(1);
    return ans;
}

double feval(double x, SEXP f, SEXP rho)
{
    defineVar(install("x"), mkans(x), rho);
    return(REAL(eval(f, rho))[0]);
}

SEXP zero(SEXP f, SEXP guesses, SEXP stol, SEXP rho)
{
    double x0 = REAL(guesses)[0], x1 = REAL(guesses)[1],
           tol = REAL(stol)[0];
    double f0, f1, fc, xc;
    SEXP res;

    if(tol <= 0.0) error("non-positive tol value");
    f0 = feval(x0, f, rho); f1 = feval(x1, f, rho);
    if(f0 == 0.0) return mkans(x0);
    if(f1 == 0.0) return mkans(x1);
    if(f0*f1 > 0.0) error("x[0] and x[1] have the same sign");
    for(;;) {
        xc = 0.5*(x0+x1);
        if(fabs(x0-x1) < tol) return  mkans(xc);
        fc = feval(xc, f, rho);
```

```
            if(fc == 0) return  mkans(xc);
            if(f0*fc > 0.0) {
                x0 = xc; f0 = fc;
            } else {
                x1 = xc; f1 = fc;
            }
        }
    }
```

The C code is essentially unchanged for the `call_R` version, just using a couple of functions to convert from `double` to `SEXP` and to evaluate `f.check`.

3.7.2 Calculating numerical derivatives

We will use a longer example (by Saikat DebRoy) to illustrate the use of evaluation and `.External`. This calculates numerical derivatives, something that could be done as effectively in interpreted R code but may be needed as part of a larger C calculation.

An interpreted R version and an example are

```
numeric.deriv <- function(expr, theta, rho=sys.frame(sys.parent()))
{
  eps <- sqrt(.Machine$double.eps)
  ans <- eval(substitute(expr), rho)
  grad <- matrix(,length(ans), length(theta),
                 dimnames=list(NULL, theta))
  for (i in seq(along=theta)) {
    old <- get(theta[i], envir=rho)
    delta <- eps * min(1, abs(old))
    assign(theta[i], old+delta, envir=rho)
    ans1 <- eval(substitute(expr), rho)
    assign(theta[i], old, envir=rho)
    grad[, i] <- (ans1 - ans)/delta
  }
  attr(ans, "gradient") <- grad
  ans
}
omega <- 1:5; x <- 1; y <- 2
numeric.deriv(sin(omega*x*y), c("x", "y"))
```

where `expr` is an expression, `theta` a character vector of variable names and `rho` the environment to be used.

For the compiled version the call from R will be

```
.External("numeric_deriv", expr, theta, rho)
```

with example usage

```
.External("numeric_deriv", quote(sin(omega*x*y)),
          c("x", "y"), .GlobalEnv)
```

Note the need to quote the expression to stop it being evaluated.

Here is the complete C code which we will explain section by section.

```
#include <S.h> /* for DOUBLE_EPS */
#include <Rinternals.h>

SEXP numeric_deriv(SEXP args)
{
  SEXP theta, expr, rho, ans, ans1, gradient, par, dimnames;
  double tt, xx, delta, eps = sqrt(DOUBLE_EPS);
  int start, i, j;

  expr = CADR(args);
  if(!isString(theta = CADDR(args)))
    error("theta should be of type character");
  if(!isEnvironment(rho = CADDR(args)))
    error("rho should be an environment");

  PROTECT(ans = coerceVector(eval(expr, rho), REALSXP));
  PROTECT(gradient = allocMatrix(REALSXP, LENGTH(ans), LENGTH(theta)));

  for(i = 0, start = 0; i < LENGTH(theta); i++, start += LENGTH(ans)) {
    PROTECT(par = findVar(install(CHAR(STRING(theta)[i])), rho));
    tt = REAL(par)[0];
    xx = fabs(tt);
    delta = (xx < 1) ? eps : xx*eps;
    REAL(par)[0] += delta;
    PROTECT(ans1 = coerceVector(eval(expr, rho), REALSXP));
    for(j = 0; j < LENGTH(ans); j++)
      REAL(gradient)[j + start] =
        (REAL(ans1)[j] - REAL(ans)[j])/delta;
    REAL(par)[0] = tt;
    UNPROTECT(2); /* par, ans1 */
  }

  PROTECT(dimnames = allocVector(VECSXP, 2));
  VECTOR(dimnames)[1] = theta;
  dimnamesgets(gradient, dimnames);
  setAttrib(ans, install("gradient"), gradient);
  UNPROTECT(3); /* ans  gradient  dimnames */
  return ans;
}
```

The code to handle the arguments is

```
  expr = CADR(args);
  if(!isString(theta = CADDR(args)))
    error("theta should be of type character");
  if(!isEnvironment(rho = CADDR(args)))
    error("rho should be an environment");
```

Note that we check for correct types of `theta` and `rho` but do not check the type of `expr`. That is because `eval` can handle many types of R objects other than `EXPRSXP`. There is no useful coercion we can do, so we stop with an error message if the arguments are not of the correct mode.

The first step in the code is to evaluate the expression in the environment `rho`, by

```
    PROTECT(ans = eval(expr, rho));
```

We then allocate space for the calculated derivative by

```
    PROTECT(gradient = allocMatrix(REALSXP, LENGTH(ans), LENGTH(theta)));
```

The first argument to `allocMatrix` gives the `SEXPTYPE` of the matrix: here we want it to be `REALSXP`. The other two arguments are the numbers of rows and columns.

```
    for(i = 0, start = 0; i < LENGTH(theta); i++, start += LENGTH(ans)) {
      PROTECT(par = findVar(install(CHAR(STRING(theta)[i])), rho));
```

Here, we are entering a for loop. We loop through each of the variables. In the `for` loop, we first create a symbol corresponding to the `i`'th element of the `STRSXP` `theta`. Here, `STRING(theta)[i]` accesses the `i`'th element of the `STRSXP` `theta`. Macro `CHAR()` extracts the actual character representation of it: it returns a pointer. We then install the name and use `findVar` to find its value.

```
      tt = REAL(par)[0];
      xx = fabs(tt);
      delta = (xx < 1) ? eps : xx*eps;
      REAL(par)[0] += delta;
      PROTECT(ans1 = eval(expr, rho));  /* not currently needed */
```

We first extract the real value of the parameter, then calculate delta, the increment to be used for approximating the numerical derivative. Then we change the value stored in `par` (in environment `rho`) by `delta` and evaluate `expr` in environment `rho` again. Because we are directly dealing with original R memory locations here, R does the evaluation for the changed parameter value.

```
      for(j = 0; j < LENGTH(ans); j++)
        REAL(gradient)[j + start] =
          (REAL(ans1)[j] - REAL(ans)[j])/delta;
      REAL(par)[0] = tt;
      UNPROTECT(2);
    }
```

Now, we compute the `i`'th column of the gradient matrix. Note how it is accessed: R stores matrices by column (like Fortran).

```
    PROTECT(dimnames = allocVector(VECSXP, 2));
    VECTOR(dimnames)[1] = theta;
    dimnamesgets(gradient, dimnames);
    setAttrib(ans, install("gradient"), gradient);
    UNPROTECT(3);
    return ans;
  }
```

First we add column names to the gradient matrix. This is done by allocating a list (a `VECSXP`) whose first element, the row names, is `NULL` (the default) and the second element, the column names, is set as `theta`. This list is then assigned as the attribute having the symbol `R_DimNamesSymbol`. Finally we set the gradient matrix as the gradient attribute of `ans`, unprotect the remaining protected locations and return the answer `ans`.

Appendix A R (internal) programming miscellania

A.1 Which R functions should stay <PRIMITIVE>?

In general, all functions should be written using `.Internal()`. However, there are exceptions which are fully specified as follows:

1. "Special functions" which really are *language* elements, however exist as <PRIMITIVE>s in R:

   ```
   {       (         if       for  while  repeat  break  next
   return  function  on.exit
   ```

2. Basic *operator*s (i.e., functions usually *not* called as `foo(a, b, ...)`) for subsetting, assignment, arithmetic and logic. These are the following 1-, 2-, and N-argument functions:

   ```
             [    [[    $
   <-   <<-  [<-  [[<-  $<-
   ----------------------------------------
   +    -    *    /     ^    %%   %*%  %/%
   <    <=   ==   !=    >=   >    \\
   |    ||   &    &&    !
   ```

3. "Low level" 0- and 1-argument functions shall remain <PRIMITIVE>, iff they belong to one of the following groups of functions:

   a. Basic mathematical functions with a single argument, i.e.,

      ```
      sign    abs
      floor   ceiling
      ----------------------
      sqrt    exp
      cos     sin      tan
      acos    asin     atan
      cosh    sinh     tanh
      acosh   asinh    atanh
      ----------------------
      cumsum  cumprod
      cummax  cummin
      ----------------------
      Im      Re
      Arg     Conj     Mod
      ```

      Note that `log` has *two* arguments, and we will use

      ```
      log <- function(x, base = exp(1)) {
        if(missing(base))
          .Internal(log(x))
        else
          .Internal(log(x, base))
      }
      ```

      in order to ensure that `log(x = pi, base = 2)` is identical to `log(base = 2, x = pi)`.

b. Functions rarely used outside of "programming" (i.e., mostly used inside other functions), such as

```
nargs        missing
interactive  is.xxx
.Primitive   .Internal   .External
symbol.C     symbol.For
globalenv    pos.to.env  unclass
```

(where *xxx* stands for almost 30 different notions, such as `function`, `vector`, `numeric`, and so forth).

c. The programming and session management utilities

```
debug    undebug   trace  untrace
browser  proc.time
```

4. The following basic assignment and extractor functions

```
.Alias      environment<-
length      length<-
class       class<-
attr        attr<-
attributes  attributes<-
dim         dim<-
dimnames    dimnames<-
```

5. A few other *N*-argument functions shall also remain <PRIMITIVE>, for efficiency reasons. Care is taken in order to treat named arguments properly:

```
:          ~          c           list        unlist
call       as.call    expression  substitute
UseMethod  invisible
.C         .Fortran   .Call
```

Function and variable index

Concept index