R の拡張を書く (Writing R Extensions)

Version 2.1.0 (2001 January)

R Development Core Team

日本語訳注：この R-exts.texi の日本語訳[1] は、英語原文と全く同じ条件の下で自由に配布、利用、修正可能である。R の開発の早さから、こうした文章の日本語訳は常に"旧式化"していることをお断りしておく。R の最新バージョン付属の文章を適宜参照されたい。R-exts.texi は GNU texinfo と呼ばれる計算機マニュアル専用の TEX の方言で書かれており、TEX でコンパイル[2] する。(2001 年 5 月 5 日)

[1] texinfo が日本語対応でないため完全には日本語化されていない。

[2] 日本語版は例えば日本語 TEX の ascii ptex を用いるなら、まず "ptex R-exts.jp.texi"、次に索引作成のため "texindex R-exts.jp.*"、そしてもう一度 "ptex R-exts.jp.texi" と三段階でコンパイルする。

Table of Contents

謝辞

Saikat DebRoy（`.Call` と `.External` の使用法に関する最初の草稿を書いた）と Adrian Trapletti（C++ へのインタフェイスに関する情報をくれた）の貢献に厚く感謝する。

1 R パッケージを作る

パッケージはオプションのコードと付属ドキュメントを読み込む機構を提供する。R の配布物は、**eda**, **mva**, そして **stepfun** といった、幾つかのパッケージを含む。

1.1 パッケージの構造

一つのパッケージは、ファイル 'DESCRIPTION' と 'INDEX' を含むサブディレクトリ, そしてファイル 'R','data', 'exec', 'inst', 'man', 'src', そして 'tests' (some of which can be missing) を含むサブディレクトリからなる（幾つかは欠けているかも知れない）。

場合によると、パッケージはまたスクリプトファイル 'configure' と 'cleanup' を含むかも知れない。これらは、Unix への移植の前（'--clean' が与えられたとき）と後に実行される、See Section 1.2 [コンフィギュアと後片付け], page 5。

1.1.1 DESCRIPTION ファイル

'DESCRIPTION' ファイルはパッケージに関する基本的な情報を含み、次のような書式を持つ：

```
Package: pkgname
Version: 0.5-1
Date: 2000/01/04
Title: My first collection of functions
Author: Friedrich Leisch <F.Leisch@ci.tuwien.ac.at>.
Depends: R (>= 0.99), nlme
Description: A short (one paragraph) description of what
        the package does and why it may be useful.
License: GPL version 2 or newer
URL: http://www.r-project.org, http://www.another.url
```

継続行（例えば、一行を越える記述）は一つの空白もしくはタブで始まる。欄 'Package', 'Version','Author', そして 'Description' は不可欠である。残りの欄 ('Date', 'Depends', 'Address','URL', ...) は無くても良い。'Author' 欄は角括弧で括った電子メイルアドレスを含むべきである（バグ報告等を送るため）。

'License' 欄は正確な文書、もしくは良く知られた省略型（例えば、'GPL', 'LGPL', 'BSD', もしくは 'Artistic'）を含むべきである。おそらく、実際のライセンスファイルへの参照が続くであろう。この情報を含むことは極めて重要である！さもなければ、他の人がこのパッケージのコピーを配布することすら法的に正しくなくなるかも知れない。

'Title' 欄はパッケージの簡略な記述を含むべきで、如何なる継続行も含むべきではない。この情報は R の古いバージョンでは別個の 'TITLE' ファイルに含まれていた。これからは、'DESCRIPTION' ファイルの 'Title' 欄を使用して欲しい。

オプションの 'URL' 欄は、コンマや空白で区切られた URL のリストを含むことががができる。例えば、作者のホーム頁やソフトウェアの追加情報を記述する頁である。これらの URL は CRAN の実際のハイパーリンクに変換される。

オプションの 'Depends' 欄は、このパッケージが依存している他のパッケージのコンマや空白で区切られたリストを与える。パッケージ名は、オプションで比較演算子（現在の所 '>=' と '<=' だけがサポートされている）と括弧に入れたバージョン番号を含む。また、あなたのパッケージが特定の

R のバージョンに依存しているのなら、特定のパッケージ名 'R' を含んで良い。つまり、パッケージがバージョン 0.90 及びそれ以降でのみ動くのであれば、'Depends' 欄に 'R (>= 0.90)' を入れる。将来の R のバージョンはこの欄を必要なパッケージの自動読み込みのために使うであろう。したがって、'Depends' 欄を必要かも知れないソフトウェアの注釈のために使ったり、不正確な構文を使わないで欲しい。他の依存関係は 'Description' 欄か、別の 'README' ファイルに羅列すべきである。R の 'INSTALL' 機構は既に、使用中の R のバージョンが移植されるパッケージに十分な程新しいかどうかを検査する。

1.1.2 INDEX ファイル

ファイル 'INDEX' はパッケージ中の十分興味ある各オブジェクト毎に、名前と記述を与える一行を含む（print メソッド等の通常明示的に呼び出されない関数は含める必要は無いかも知れない）。あなたのシステムで Perl が使えるか、package builder (see Section 1.3 [パッケージの検査と構築], page 6) を使えば、`R CMD Rdindex man > INDEX` 等の機能でこのファイルを自動的につくり出すことができることを注意しよう。

1.1.3 パッケージのサブディレクトリ

'R' サブディレクトリは R のコードファイルを含む。移植されるべきコードファイルは（小もしくは大）文字で始まり、'.R', '.S', '.q', '.r', もしくは '.s' のいずれかの拡張子を持たなければならない。我々は '.R' を用いることを勧める。なぜなら、この拡張子は他のソフトウェアで使われていないように見えるからである。R オブジェクトが付値により生成できるように、`source()` を用いてファイルを読み込むことができるべきである。ファイルと、それにより生成される R オブジェクトの名前の間には関連が無くても良い。必要なら、これらのファイルのどれか（歴史的慣例では 'zzz.R'）はコンパイル済みコードを読み込むために `.First.lib()` 内で `library.dynam()` を使用すべきである。

'man' サブディレクトリはパッケージ中のオブジェクトの "R documentation" (Rd) 書式によるドキュメントを含むべきである。移植されるべきドキュメントファイルは同じく（小もしくは大）文字で始まり、拡張子 '.Rd' （既定では）もしくは '.rd' を持つべきである。詳細は See Chapter 2 [R のドキュメントを作る書く], page 8 を参照せよ。パッケージ中の全てのユーザーレベルのオブジェクトはドキュメント化されるべきである。もし、パッケージ *pkg* が "内部的" のみに使用されるユーザーレベルのオブジェクトを含むならば、そうしたオブジェクト全てを文章化したファイル '*pkg*-internal.Rd' を用意すべきで、明確にこれらはユーザーにより呼び出されることを企図していないことを述べるべきである。例えば、R ディストリビューション中のパッケージ **ts** を見よ。

'R' と 'man' サブディレクトリは OS に依存したサブディレクトリ named 'unix', 'windows' もしくは 'mac' を含んで良い。

コンパイル済みコードに対する C, C++, もしくは FORTRAN のソースファイル、そしてオプションのファイル 'Makevars' もしくは 'Makefile' は 'src' 中に置く。あるパッケージが `R CMD INSTALL` を用いインストールされるときは、コンパイルとリンクを制御し R 中に読み込まれる共有ライブラリにするために Make が使われる。このための既定の変数と規則がある (R が configure される時に決定され '$R_HOME/etc/Makeconf' に記録される)。もしあるパッケージが、ヘッダファイル ('-I' オプション) やリンクのための追加のライブラリ ('-l' と '-L' オプション) を探すために追加のディレクトリを指定する必要がある時は、これを 'src/Makevars' 中の変数 `PKG_CPPFLAGS` と `PKG_LIBS` を用いて行う。(C, C++, 又は FORTRAN コンパイラに引き渡される追加のフラグはそれぞれ、変数 `PKG_CFLAGS`, `PKG_CXXFLAGS` そして `PKG_FFLAGS` を用いる。) この機構はパッケージ固有の 'Makefile' を必要としない程に十分一般的であることを注意しよう。もしそうしたファイルを配布するときは、それが全ての R のプラットフォームで動作するに十分なだけ一般的に

するために、かなりの注意が必要である。もし必要なら、プラットフォーム固有のファイルを使っても良い。例えば、Windows 上では 'Makevars.win' もしくは 'Makefile.win' が 'Makevars' や 'Makefile' に優先する。

'data' サブディレクトリは、パッケージが data() を用いて読み込めるようにするための追加のデータファイル様である。現在のところ、データファイルはその拡張子で指示される次の三つのタイプのいずれかを持つことが出来る：プレーン R コード ('.R' 又は '.r')、表 ('.tab', '.txt' 又は '.csv')、そして save() イメージ ('.RData' 又は '.rda')。(可搬性のために、save(, ascii=TRUE) を用いてセーブしたイメージを使用して欲しい。) R のコードは "自己充足的" で、パッケージが提供する余分の機能を使わないようにするべきであることを注意しよう。こうすることにより、データファイルはまたパッケージを読み込むことなしに使用できるようになる。'data' サブディレクトリは又使用可能なデータセットを説明する '00Index' ファイルを含むべきである。理想的には、これは各データセットのオンライン記述と、'man' ディレクトリ中の完全なドキュメントを持つべきである。

'inst' サブディレクトリの中身はインストール先ディレクトリに再帰的にコピーされるであろう。

サブディレクトリ 'tests' は、R のディストリビューションと一緒に配布される個々の検査の様な、追加のパッケージ-固有の検査用コード用である。

検査用コードは直接 '.R' ファイルに入れるか、又は対応する '.R' ファイルをそれから生成するコードを含む '.Rin' ファイル (例えば、パッケージ中の全ての関数を集め、それからそれらをもっとも奇妙な引数で呼び出す等) 経由で提供される。'.R' ファイルを走らせた結果は '.Rout' ファイルに書き込まれる。もし対応する '.Rout.save' ファイルがあれば、この二つが比較され、エラーを引き起こすこと無しに相違点が報告され。

最後に、'exec' はパッケージが必要とする追加の実行プログラム、典型的にはシェルもしくは Perl スクリプト、を含んでも良いかも知れない。この機構は現在全ての Unix パッケージで使われてはおらず、依然として実験的である。

1.1.4 パッケージの梱

幾つかのパッケージを梱, *bundle* として配布するのが便利な場合がある。(現在の主要な例は四つのパッケージを含む **VR**。) Unix と Windows に於けるインストール手順はパッケージの梱を処理できる。

一つの梱の 'DESCRIPTION' ファイルは次のような余分の 'Bundle' 欄を持つ

```
Bundle: VR
Contains: MASS class nnet spatial
Version: 6.1-6
Date: 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)
```

'Contains' 欄はパッケージをリストする。これは名前を与えられた個々のサブディレクトリに収められているべきである。'DESCRIPTION' ファイルは、梱の 'DESCRIPTION' ファイル に追加の欄

だけを含む ‘DESCRIPTION.in’ ファイルに置き換えられていることを除けば、これらはあらゆる点で標準のパッケージである。

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

1.2 コンフィギュアと後片付け

もしあるパッケージがインストールの前にシステム依存のあるコンフギュレーションを必要とするならば、他の全ての実行の前に R CMD INSTALL により実行されるスクリプト ‘configure’ をこのパッケージに含めても良い。これは autoconf により作り出されるスクリプトでも良いが、作者自身が書いたスクリプトでも良い。パッケージがコンパイルされたり使用される時点でエラーメッセージを与えるよりも、対応するパッケージ中のコードがインストール時に動作不能になるような非標準的なライブラリが存在するかどうかをこれを用いて検出せよ。要約すれば、autoconf の全能力 (変数の代入、ライブラリの検索等) が拡張パッケージで使えるようにせよ。

スクリプト ‘cleanup’ が存在し、オプション ‘--clean’ が与えられれば、R CMD INSTALL により最後に実行され、パッケージのソースツリーを清掃出来る。特に ‘configure’ が作り出したファイルが消される。

例として (C 又は FORTRAN) ライブラリー foo が提供する機能を使いたいとしよう。autoconf を用い、ライブラリーを検査し、変数 HAVE_FOO が存在すれば TRUE にさもなければ FALSE に設定し、それからこの値を (HAVE_FOOを値に持つ入力ファイル中のインスタンス ‘@HAVE_FOO@’ 置き換え) 出力ファイル中に出力する、コンフィギュアスクリプトを書くことが出来る。例えば、名前が bar の関数を、ライブラリー foo をリンクすることにより (つまり ‘-lfoo’ を使う) 使えるようにするには、次のようにすることが考えられる

```
AC_CHECK_LIB(foo, fun, HAVE_FOO=TRUE, HAVE_FOO=FALSE)
AC_SUBST(HAVE_FOO)
...
AC_OUTPUT(foo.R)
```

‘foo.R.in’ 中の対応する R 関数は次のようになるであろう

```
foo <- function(x) {
    if(!@HAVE_FOO@) stop("Sorry, library ‘foo’ is not available")
    ...
```

このファイルから configure は、(必要な機能を持つ) ライブラリー foo が存在しなければ、次のような実際の R のソースファイル ‘foo.R’ を作る。

```
foo <- function(x) {
    if(!FALSE) stop("Sorry, library ‘foo’ is not available")
    ...
```

この場合、上の R コードは実際には関数を無効にする。

利用可能な又は欠けている機能に対する異なった部分ファイルを使うことも出来るであろう。

‘configure’ スクリプトはウィンドウズシステムではうまく働かないかも知れないことを念頭におくべきである (これは autoconf が作り出すスクリプトでは普通であるが、簡単なシェルスクリプトは機能する)。もしあなたのパッケージが広く利用可能になるためには、非ユニックスプラットフォーム上のユーザーに対するコンフィギュアの仕方を文章で十分に与えて欲しい。

稀な場合、コンフィギュレーションと後片付けスクリプトはパッケージがインストールされるべき場所を知る必要がある。一例は C コードを使用し二つの共有ライブラリー/DLL を作り出すパッケージである。普通、R に動的に読みこまれるライブラリーは二番目の従属ライブラリーに対しリンクされる。あるシステムでは、この従属ライブラリーの位置を R が動的に読みこむライブラリーに加えることが出来る。これは各ユーザーが `LD_LIBRARY_PATH` 環境変数の値を設置する必要が無いことを意味し、第二ライブラリーは自動的に参照される。別の例は、パッケージが実行時に必要な支援ファイルをインストールし、それらの位置がインストール時に R のデータ構造に代入される場合である。(これは Java パッケージ中の Java Archive ファイルで起こる。)

上位のライブラリーのディレクトリ名 (つまり、'`-l`' 引数で指定されるもの) とパッケージ自身のディレクトリは、環境変数 `R_LIBRARY_DIR` と `R_PACKAGE_DIR` を用いて、インストールスクリプトに伝えることが出来る。更に、インストールされているパッケージ名 (例えば '`survival`' や '`MASS`' といった) はシェル変数 `R_PACKAGE_NAME` から理用可能である。

1.3 パッケージの検査と構築

R のパッケージ検査プログラム `R CMD check` を用い、R のソースパッケージが正しく動くかどうか検査することが出来る。(Windows での対応する命令は `Rcmd check` である。) これは一連の検査を実行する。

1. 先ずパッケージをインストールしようとする。これはヘルプファイル中の欠けている相互参照と重複したエイリアスを警告する。
2. '`DESCRIPTION`' ファイルが完全かどうか検査する。
3. Rd ファイルの `\name`, `\alias` そして `\keyword` 欄が検査される。
4. パッケージ中の文章化されていないユーザーレベルのオブジェクトを検査する。
5. パッケージの文章中の例が実行される (実行可能な実例コードに関する情報は`\examples`see Chapter 2 [R のドキュメントを作る書く], page 8 をみよ)。
   勿論、公開されたパッケージは少くともそれ自身の例を実行できなければならない。
6. もしパッケージのソースが '`tests`' ディレクトリを含めば、そのディレクトリ中で指示された検査が実行される。 (典型的にはそれらは '`.R`' ソースファイルと目標出力ファイル '`.Rout.save`'. の集まりからなる。)
7. もし実行可能な `latex` プログラムが利用できれば、パッケージマニュアルの '`.dvi`' 版が作られる (Rd ファイルがうまく変換できるかどうか検査するため)。

R パッケージの検査法に関するより多くの情報を得るには *`R CMD check --help`* (Windows 上では *`Rcmd check --help`*) を使おう。フラグを加えることにより、検査の一部分だけを実行できる。

R パッケージの作成命令である `R CMD build` を使って、 (例えば, 引き続くリリース用に) R パッケージをそのソースから作ることが出来る。 Windows における対応物は `Rcmd build` である。

パッケージを通常の gzip された tar ファイルとして構成するに先立ち、様々な診断チェックと整理整頓が実施される。特に、'`DESCRIPTION`' ファイルが必要な項目を含んでいるか、オブジェクトとデータの目録が存在し (さもなければそれらを作り出す) 最新のものと仮定して良いかどうかが検査される。

構築プロセスに先立ち、パッケージが正常に動くかどうかの実行時検査を `R CMD check` を使って行うべきである。

R のパッケージ作成命令に関するより詳しい情報は *`R CMD build --help`* (Windows では *`Rcmd build --help`*) で得ることが出来る。

`R CMD build` はバイナリー配布用に既にコンパイルしたパッケージの版を作ることも出来る。

1.4 CRAN への投稿

CRAN は R の配布物と貢献コード、特に R パッケージ、をおいてある WWW サイトのネットワークである。R のユーザーは共同プロジェクトに参加し、自分自身で貢献して欲しい。

パッケージ **mypkg** を投稿する前に、それが完全で適正に移植出来るかどうか検査するために次のステップを実行して欲しい。

1. `R CMD check` を使ってパッケージ が移植でき、その実例を実行出来るかどうか、ドキュメントが完全で処理可能かどうか、を検査する。
2. `R CMD build` を使って幾つかの更なる検査を実行し、リリース用の '`.tar.gz`' ファイル を構築する。

全ての過程が、理解可能で無くす必要が無いような警告を出すだけで処理できることを確認して欲しい。

全ての検査が終了したら、'`.tar.gz`' を

`ftp://ftp.ci.tuwien.ac.at/incoming`

へアップロードし、それに関する報告を `WWWadmin@ci.tuwien.ac.at` におくって欲しい。CRAN の管理者は投稿を主アーカイブにおく前に、これらの検査を実行するであろう。

2 R のドキュメントを作る書く

2.1 Rd 書式

R のオブジェクトは (La)TEX に良く似た簡単なマークアップ言語である “R documentation” (Rd) 書式で書かれたファイルにドキュメント化されている。これは LaTEX, HTML そして平文を含む様々な書式に変換できる。翻訳は ‘$R_HOME/bin’ 中にある Perl スクリプト Rdconv とパッケージのインストール用スクリプトにより実行される。

R の配布は 700 を越えるそうしたファイルを持ち、R のソースツリーの ‘src/library/*pkg*/man’ ディレクトリにある。ここで *pkg* は全ての標準的オブジェクトがあるパッケージ **base** や、R の配布物に含まれる **eda** や **mva** といった標準パッケージを表す。

例として R の関数 rle を説明するファイル ‘src/library/base/man/rle.Rd’ を眺めてみよう。

```
\name{rle}
\alias{rle}
\title{Run Length Encoding}
\description{
  Compute the lengths and values of runs of equal values in
  a vector.
}
\usage{
rle(x)
}
\arguments{
  \item{x}{a (numerical, logical or character) vector.}
}
\value{
  A list with components
  \item{lengths}{a vector containing the length of each run.}
  \item{values}{a vector of the same length as \code{lengths}
    with the corresponding values.}
}
\examples{
x <- rev(rep(6:10, 1:5))
rle(x)
## $lengths
## [1] 5 4 3 2 1
## $values
## [1] 10  9  8  7  6
z <- c(TRUE,TRUE,FALSE,FALSE,TRUE,FALSE,TRUE,TRUE,TRUE)
rle(z)
rle(as.character(z))
}
\keyword{manip}
```

一つの Rd ファイルは三つの部分からなる。ヘッダーはファイルの名前、説明されている話題、表題、説明されているオブジェクトに対する短い文章による記述と R の使用法情報に関する基本的な情報を与える。本体はそれ以上の情報 (例えば、上の例におけるように関数の引数や返り値) を与える。最後に、キーワード情報を与えるフッターがある。ヘッダーとフッターは不可欠である。

パッケージ作者に有用である Rd 書式でドキュメントを書くガイドラインについては “Guidelines for Rd files” (`http://developer.r-project.org/Rds.html`) を見よ。

2.1.1 ドキュメント用関数

R オブジェクト (特に関数) のドキュメントに使われる基本的マークアップ命令がこの副節で与えられる。

`\name{`*file*`}`
: *file* はファイルの基本名である。

`\alias{`*topic*`}`
: 項目 `\alias` はファイルが説明する全ての “話題” を指定する。この情報 (ファイル名とともに) はオンライン (平文や HTML) のヘルプシステムによる検索用の索引データベースに集められる。

  複数の `\alias` 項目があっても良い。しばしば複数の R オブジェクトを一つのファイルで説明するのが便利である。例えば、ファイル ‘`Normal.Rd`’ は正規分布の密度、分布関数、クォンタイル関数そして乱数の生成を説明しており、したがって次のように始まる

  ```
  \name{Normal}
  \alias{dnorm}
  \alias{pnorm}
  \alias{qnorm}
  \alias{rnorm}
  ```

  ファイル名は文章化された話題である必要はないことを注意しよう。

`\title{`*Title*`}`
: Rd ファイルに対するタイトル情報。これは大文字で始まり、最後にピリオッドをおかず、如何なるマークアップ命令も使わない (ハイパーテキスト検索に問題を起こす)。

`\description{...}`
: 関数が何をするのかに関する短い記述 (一段落、数行で良い)。(もし記述が “長すぎ” そして容易に短くできないならば、恐らくファイルは多くのことを一度に説明しようとしているのだろう。)

`\usage{`*fun*`(`*arg1*`, `*arg2*`, ...)}`
: ファイル中に説明されている関数と変数の用法を一行もしくは数行で示す。これらはタイプライターフォントを用いた verbatim モードで表示される。

  説明される使用情報は一般に (コードと文章間の一貫性の自動検査が可能になるように) 関数の定義と 正確に 一致する必要があるさもなければ実際の用法を示す `\synopsis` 節を含めよう。

  例えば `abline` はプロットに直線を加える関数で、名前つき引数に応じて色々に使える。したがって ‘`abline.Rd`’ は次の内容を含む

  ```
  \synopsis{
  abline(a = NULL, b = NULL, h = NULL, v = NULL, reg = NULL,
  ```

```
        coef = NULL, untf = FALSE, col = par("col"),
        lty = par("lty"), lwd = NULL, ...)
}
\usage{
abline(a, b, \dots)
abline(h=, \dots)
abline(v=, \dots)
...
}
```

`\arguments{...}`
: 関数の引数の記述で、次のような形式

  `\item{`*arg_i*`}{`*Description of arg_i*`.}`

  の項目を引数リストの各要素に対し使う。項目の前後にオプションでテキストをおいても良い。

`\details{...}`
: 提供される機能の詳細な、可能なら正確な、記述で、`\description` 項目中の基本情報を拡張する、

`\value{...}`
: 関数の返り値の記述。

  もし複数の返り値を含むリストが返されるならば、次の形式

  `\item{`*comp_i*`}{`*Description of comp_i*`.}`

  の項目を返されるリストの各項目に加えても良い。オプションのテキストが前に付いても良い (紹介例 `rle` を見よ)。

`\references{...}`
: 文献への参照の節。ウェッブへのポインターに対しては `\url{}` を使う。

`\note{...}`
: 必要な特別な注意にはこれを使う。

  例えば ‘`piechart.Rd`’ は以下を含む

```
\note{
  Pie charts are a very bad way of displaying information.
  The eye is good at judging linear measures and bad at
  judging relative areas.
  ...
}
```

`\author{...}`
: Rd ファイルの著者に関する情報。電子メイルアドレスを指定するには余計な区切り文字 (‘`( )`’ もしくは ‘`< >`’) 無しの `\email{}` を使う。ウェッブへのポインターは `\url{}` を使う。

`\seealso{...}`
: 関連 R オブジェクトへのポインターで、それらを参照するには `\code{\link{...}}` を使う (`\code` は R オブジェクト名に対する正しいマークアップであり、`\link` はこれをサポートする出力書式によるハイパーリンクを生成する。See Section 2.3 [マークされたテキスト], page 13 と Section 2.5 [相互参照], page 13)。

`\examples{...}`
: 関数の使用法の例。これらはタイプライターフォントを用いた verbatim モードで表示される。

  実例はドキュメントの目的に有用であるだけでなく、R の診断的チェックとして使われる検査コードを提供する。既定では、`\examples{}` 中のテキストはヘルプ頁の出力に表示され、*make check* により実行される。表示されるだけで実行されない命令に対しては `\dontrun{}` を使うことが出来、ユーザーには示されるべきではない R の検査用の追加命令には`\testonly{}` を用いる。

  例えば

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

  このように、`\dontrun` 中に含まれる以外の実例コードは実行できなければならない。加えて、システム依存の特徴や特殊な機能 (例えばインターネットへのアクセスや特定のディレクトリの書き込み許可) を使うべきではない。

  実例を実行可能にするために必要なデータは乱数により (たとえば `x <- rnorm(100)`)、もしくは `data()`を使って読み込み出来る標準的なデータセットより得ることが出来る (情報については `data()` を見よ)。

`\keyword{`*key*`}`
: 各 `\keyword` 項目は標準的キーワードの一つ (ファイル '`$R_HOME/doc/KEYWORDS`' に指示されたような) を指定しなければならない。少くとも一つの `\keyword` 項目が必要であるが、もし文章化される R オブジェクトが一つ以上のカテゴリーに分類されるならば一つ以上でも良い。

R 関数 `prompt` は R のオブジェクトのドキュメントファイルの作成を手助けする。もし `foo` が R の関数ならば、`prompt(foo)` は既に `foo` の適正な関数・引数名を持ち、情報で埋めることの出来る構造を持ったファイル '`prompt.Rd`' を作成する。

2.1.2 データセットのドキュメント化

R のデータセットのドキュメントである Rd ファイルは少々異なる。`\arguments` や `\value` といった節が不要な一方、データの出典と書式が説明されなければならない。

例えば、標準的な R のデータセット `rivers` のドキュメントである '`src/library/base/man/rivers.Rd`' を眺めてみよう。

```
\name{rivers}
\alias{rivers}
\title{Lengths of Major North American Rivers}
\description{
  This data set gives the lengths (in miles) of 141 ``major''
  rivers in North America, as compiled by the US Geological
  Survey.
}
\usage{data(rivers)}
\format{A vector containing 141 observations.}
\source{World Almanac and Book of Facts, 1975, page 406.}
\references{
  McNeil, D. R. (1977) \emph{Interactive Data Analysis}.
  New York: Wiley.
}
\keyword{datasets}
```

これは以下の補助的なマークアップ命令を使う。

`\format{...}`
: データセットの書式の記述 (ベクトル、行列、データフレーム、時系列等)。行列とデータフレームに対しては、これは各列の説明を与えるべきであり、できればリストか表の形が望ましい。より詳しくは See Section 2.4 [リストと表], page 13 を見よ。

`\source{...}`
: 原出典の詳細 (参考文献もしくは URL)。更に、節 `\references` は二次的出典と用法を与えることが出来る。

データセット *bar* を文章化する際にもこれを注意せよ。

- `\usage` 項目は常に `data(`*bar*`)`。
- `\keyword` 項目は常に '`datasets`'。
- `\keyword` 項目は常に '`datasets`'。

もし `bar` がデータフレームなら、それをデータセットとして文章化することは再び *`prompt(bar)`* で始めることが出来る。

2.2 節区分

新しいパラグラフを始めたり、例中に空白行を残したければ、単に空白行を挿入せよ ((La)TEX と同様に)。行換えをするためには `\cr` を使う。

予め定義された節 (`\description{}`, `\value{}` 等) 以外に、任意の節を `\section{`*section_title*`}{...}` で定義できる。例えば

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

既定義済み節との一貫性のために、節名 (`\section` への第一引数) は大文字化 (しかし全て大文字にしない) する。

追加の名前付きの節、それが入力のどこにあろうと、常に出力の一定の位置 (`\note`, `\seealso` そして実例の前) に挿入されることを注意しよう。

2.3 マークされたテキスト

以下の論理的マークアップ命令は特別な種類のテキストを指示するために利用できる。

\bold{*word*}	もし可能なら *word* を **bold** フォントにする
\emph{*word*}	もし可能なら *word* を **italic** フォントで強調する
\code{*word*}	コードの断片を、もし可能なら typewriter フォントにする
\file{*word*}	ファイル名用
\email{*word*}	電子メイル用
\url{*word*}	URL 用

最初の二つ、\bold と \emph は平文で強調のために使うべきである。

R オブジェクトの名前を含む、R コードの断片は\code を使ってマークアップされるべきである。\code 中では、バックスラッシュとパーセント記号だけが (バックスラッシュ記号を用い) エスケープされる必要がある。

2.4 リストと表

\itemize と \enumerate 命令は一つの引数を取り、その中では一つもしくは複数の \item 命令があって良い。各 \item に続くテキストは一つもしくは複数のパラグラフとして整形され、適切にインデントされ、最初のパラグラフは bullet 記号 (\itemize) もしくは番号 (\enumerate) でマークされる。

\itemize と \enumerate 命令は入れ子になっても良い。

\describe 命令は \itemize に似るが、最初のラベルを指定できる。\item は二つの引数、ラベルと項目本体、を \item の引数と値と全く同じ様に取る、\describe 命令は HTML の <DL> リストや LaTeX の \description リストに対応される。

\tabular 命令は二つの引数を取る。最初は各列に対して必要な整列法 (左揃えには 'l'、右揃えには 'r'、又は中央化には 'c') を与える。二つ目の引数は \cr で分離された任意個数の行からなり、各欄は \tab で分離される。例えば:

```
\tabular{rlll}{
  [,1] \tab Ozone   \tab numeric \tab Ozone (ppb)\cr
  [,2] \tab Solar.R \tab numeric \tab Solar R (lang)\cr
  [,3] \tab Wind    \tab numeric \tab Wind (mph)\cr
  [,4] \tab Temp    \tab numeric \tab Temperature (degrees F)\cr
  [,5] \tab Month   \tab numeric \tab Month (1--12)\cr
  [,6] \tab Day     \tab numeric \tab Day of month (1--31)
}
```

第一引数中の整列指示と同じ個数の欄が各行に存在する必要があり、空白であってはならない (しかし、空白文字だけであっても良い)。

2.5 相互参照

マークアップ \link{*foo*} (普通 \code{\link{*foo*}} と組み合わせて使う) はオブジェクト *foo* のヘルプ頁へのハイパーリンクを作り出す。\link の一つの主な利用はヘルプ頁の \seealso 節である、see Section 2.1 [Rd 書式], page 8。(これは、たとえば help.start() が利用する HTML 頁や参考マニュアルの PDF 版中のハイパーリンクの生成だけに影響する。)

それぞれ話題 (ファイル?) *foo* と *bar* へのリンク\link[*pkg*]{*foo*} と \link[*pkg*:*bar*]{*foo*} として指定されるオプションの引数がある。

2.6 数式

数学式は表示ドキュメントでは美しく整形されるべきであるが、テキストや HTML 形式のオンラインヘルプに対してはまだ適切なものがない。この目的のため、二つの命令`\eqn{`*latex*`}{`*ascii*`}` と `\deqn{`*latex*`}{`*ascii*`}` が使われる。ここで `\eqn` は “行中” 数式 (TEX の `$...$`)、`\deqn` は “数式行” (LaTEX の `displaymath` 環境、もしくは TEX の `$$...$$`) に使われる。

二つの命令はまた *latex* と *ascii* の双方で使える`\eqn{`*latexascii*`}` (引数一つだけ) の形式でも使うことが出来る。

次の例は `Poisson` のヘルプ頁から取った。

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

LaTEX マニュアルではこれは次のようになる

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

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

HTML とテキスト版オンラインヘルプでは次の様になる

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

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

2.7 挿入

R システムそれ自身には `\R` を使う (余分の ‘`{}`’ や ‘`\`’ は不要である)。関数引数リスト中の ‘`...`’ には `\dots` を使う。そして、通常テキスト中の省略記号 `...` には `\ldots` を使う。

‘`%`’ の後にはヘルプテキストに関するコメントをおくことが出来る。行のそれ以後は普通完全に無視されるであろう。従って、ヘルプの一部を完全に見えなくすることにそれを使うことが出来る。

バックスラッシュ(‘`\`’) はそれを別のバックスラッシュでエスケープすることによりえることが出来る。(行換えには `\cr` が使えることを注意しよう。)

“コメント” と “制御” 文字 ‘`%`’ と d ‘`\`’ は 常に エスケープされる必要がある。verbatim 風命令 (`\code` と `\examples`) の中では、その他の文字[1] は特別ではない。`\file` は verbatim 風命令ではないことを注意しよう。

“普通の” テキスト (verbatim, `\eqn`, ... でない) では、現在のところほとんどの LaTEX の特殊文字をエスケープする必要がある、例えば、‘`%`’, ‘`{`’ そして ‘`}`’ を除き、四つの特殊文字 ‘`$`’, ‘`#`’ そして ‘`_`’ は各々先頭に ‘`\`’ を付けて得られる。(‘`&`’ もエスケープ出来るが、必要はない。) 更に、‘`^`’ を `\eqn{\mbox{\textasciicircum}}{^}`, そして ‘`~`’ を`\eqn{\mbox{\textasciitilde}}{~}` もしくは `\eqn{\sim}{~}` (それぞれ短・長のチルダ) で入力する。又 ‘`<`’, ‘`>`’ そして ‘`|`’ は数式モードだけで使うべきである、つまり、`\eqn` もしくは `\deqn` 中。

[1] これは完全には真実ではない。対になっていない中括弧は問題を生じ、エスケープすべきである。例えばファイル ‘`Paren.Rd`’ 中の実例の節を見よ。

2.8 プラットフォーム固有の文章

しばしばドキュメントはプラットフォームにより異なる必要がある。現在のところ、三つの OS 固有のオプション、`unix`, `windows` そして `mac` が利用可能で、ヘルプのソースファイル中の行は次のようにして

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

もしくは

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

OS 固有の挿入や除外を指示できる。

もしプラットフォーム間の違いが甚だしいか、文章化された R オブジェクトが一つのプラットフォームにだけ関連するのであれば、プラットフォーム固有の Rd ファイルを '`unix`', '`windows`' もしくは '`mac`' サブディレクトリに置くことが出来る。

2.9 Rd 書式の処理

UNIX 版の R では Rd ファイルを処理するいくつかの命令がある。Windows での対応物はこの節の最後で説明される。これら全ては Perl がインストールされていることが前提である。

`R CMD Rdconv` を使って R のドキュメント書式を他の書式に変換したり、実行時検査のために実行可能例を取り出したりできる。現在のところ、平文、HTML、LaTeX そしてバージョン 3 の S のドキュメント書式への変換が可能である。

この低水準変換ツールに加えて、R の配布物は Rd 書式 を処理する二つのユーザーレベルのプログラムを提供する。`R CMD Rd2txt` は "清書" された平文出力を Rd ファイルから作り出し、特に Rd 書式ドキュメントを Emacs で書くときにプレビューするのに便利である。`R CMD Rd2dvi` は Rd ファイルから DVI (もしくは、もし option '`--pdf`' が与えられると PDF) 出力を生成し、 Rd ファイルは明示的に、またはパッケージのソースのディレクトリへのパスで指示できる。後者ではパッケージ中の全てのドキュメントオブジェクトに対する参考マニュアルが生成される。将来のバージョンでは同時に '`DESCRIPTION`' ファイル中の情報が加えられるであろう。

`R CMD Rdindex` を使って、引数に指定された Rd ファイルのタイトルと名前を表示する読みやすく整形された索引ファイルを作ることができる。これはアドオンパッケージの '`INDEX`' を作るのに使え、そいてもしそれがデータも含めば '`data`' ディレクトリ中に '`00Index`' データ索引を作る。パッケージ作成プログラム `R CMD build` はパッケージを作る際自動的にこれらの索引を作ることを注意しよう。

最後に `R CMD Sd2Rd` はバージョン 3 の S ドキュメントファイル (これは拡張された Nroff 書式を使う) を Rd 書式に変更する。これは最初 S システム用に書かれたパッケージを R に移植するのに便利である。

上の各命令の正確な用法と利用可能なオプションの詳細なリストは `R CMD` *command* `--help`、つまり *`R CMD Rdconv --help`* を実行して得られる。全ての利用可能な命令は *`R --help`* を用いて一覧できる。

これら全ての命令は Windows での対応物を持つ。多くの場合単に `RCMD` を `Rcmd` に置き換えるだけである。例外は `Rcmd Rd2dvi.sh` (そしてこれはソースからパッケージをインストールするツー

ルを必要とする)。これらの各々に対し *R CMD cmd* `--help` は使用法の詳細を与える。(ソースパッケージをインストールために R バイナリの Windows 配布物を必要とするであろう。)

3 R コードの整理とプロファイリング

パッケージとして保存し、またおそらく他の人が使えるようにすることに値する R コードはドキュメント化し、整理整頓し、そしておそらく最適化するに値する。最後の二つの行為がこの章の主題である。

3.1 R コードの整頓

R は、パッケージから読み込んだ関数コードと、使用者が入力したコードを違う扱いをする。使用者が入力したコードはある仕方で保存されたソースコードを持ち、関数を表示すると、元々のソースが再生される。パッケージから読み込まれたコードは（既定では）ソースコードを破棄し、関数の表示は関数のリストは関数の構文解析木から再構成される。

普通ソースコードを保存しておくことは良い考えであり、特に、注釈がソースから取り去られることを防ぐ。しかしながら、一貫したインデント、演算子周りの空白、好ましい付値演算子 `<-` の一貫した使用を持つ、簡潔な関数リストを構文解析木から再構成することができる。この簡潔な版は、標準的な書式に慣れた他の読者はいうに及ばず、より読みやすい。

ソースの保存を破棄する二つの方法がある。

1. コードが R に読み込まれる前に、オプション `keep.source` を `FALSE` に設定することができる。
2. 保管されたソースコードをその `source` 属性を取り除くことで破棄できる。例えば、

```
attr(myfun, "source") <- NULL
```

どちらの場合も、関数をリストすれば、標準的なレイアウトを得るであろう。

整頓したい関数ファイル '`myfuns.R`' があるとしよう。以下を含むファイル '`tidy.R`' を作る

```
options(keep.source = FALSE)
source("myfuns.R")
dump(ls(all = TRUE), file = "new.myfuns.R")
```

そして、R をこのソースファイルを用いて走らせる。これをソースコードとすると、例えば、`R --vanilla < tidy.R` (Unix) または `Rterm --vanilla < tidy.R` (Windows)、もしくは一つの R セッション中に取り込む。そうすると、ファイル '`new.myfuns.R`' はアルファベット順に並べられた標準レイアウトの関数を含むであろう。注釈をより適切な場所で取り除くことが必要になるかも知れない。

標準書式はそれ以降の整理整頓のための良い出発点を与える。ほとんどのパッケージ作者は R コードを編集するために、Emacs のバージョン（Unix もしくは Windows 上）を ESS Emacs パッケージの ESS[S] モードを使って編集している。R 自身のソースコード用に推薦される ESS[S] モード中のスタイルオプションについては Appendix B [R のコーディングの標準], page 57 を参照せよ。

3.2 R コードのプロファイル

R のバージョン 1.2.0 から、ほとんどの Unix 互換の R のバージョンで R コードをプロファイルすることができるようになった。プロファイル機能は既定の構築では無効にされているので、この機能を有効にするためには、R をオプション '`--enable-R-profiling`' 付きで構築する必要がある。残念なことに、プロファイルは Windows では使用できない OS の機能に依存している。

命令 `Rprof` を使いプロファイルを制御する。そのヘルプ頁が完全な詳細を与える。プロファイリングは決まった（既定では 20 ミリ秒）期間毎にどの R 関数が使われているかを記録し、結果をある

ファイル（既定では作業ディレクトリ中の ‘Rprof.out’）に書き込むことで動作する。そして命令 R CMD Rprof *Rprof.out* を使って活動を要約することができる。

例として、次のコード (Venables & Ripley, 1999) を考えよう。

```
library(MASS); library(boot); library(nls)
data(stormer)
storm.fm <- nls(Time ~ b*Viscosity/(Wt - c), stormer,
                start = c(b=29.401, c=2.2183))
st <- cbind(stormer, fit=fitted(storm.fm))
storm.bf <- function(rs, i) {
    st$Time <-  st$fit + rs[i]
    tmp <- nls(Time ~ (b * Viscosity)/(Wt - c), st,
               start = coef(storm.fm))
    tmp$m$getAllPars()
}
rs <- scale(resid(storm.fm), scale = F) # remove the mean
Rprof("boot.out")
storm.boot <- boot(rs, storm.bf, R = 4999) # pretty slow
Rprof()
```

これを走らせた後、結果を次の命令で要約できる

```
R CMD Rprof boot.out

Each sample represents 0.02 seconds.
Total run time: 153.72 seconds.

Total seconds: time spent in function and callees.
Self seconds: time spent in function alone.

   %       total       %       self
 total    seconds     self    seconds    name
100.00    153.72      0.21      0.32     "boot"
 99.67    153.22      0.57      0.88     "statistic"
 96.10    147.72      2.15      3.30     "nls"
 53.36     82.02      1.12      1.72     "<Anonymous>"
 49.92     76.74      0.88      1.36     "list"
 49.38     75.90      1.20      1.84     ".Call"
 21.35     32.82      2.35      3.62     "eval"
 18.87     29.00      0.77      1.18     "as.list"
 18.64     28.66      0.43      0.66     "switch"
 17.47     26.86      2.55      3.92     "nlsModel"
 16.82     25.86      0.42      0.64     "model.frame"
 16.41     25.22      1.14      1.76     "model.frame.default"
 15.86     24.38      1.42      2.18     "qr.qty"
 14.06     21.62      2.76      4.24     "assign"
 13.06     20.08      1.57      2.42     "qr.coef"
 10.76     16.54      2.81      4.32     "storage.mode<-"
  ...
```

```
  %       self        %       total
self     seconds    total    seconds    name
 5.80      8.92      6.61     10.16     "paste"
 4.25      6.54      8.13     12.50     "as.integer"
 4.07      6.26      7.62     11.72     "names"
 3.97      6.10      9.80     15.06     ".Fortran"
 3.36      5.16      4.74      7.28     "as.double"
 2.81      4.32     10.76     16.54     "storage.mode<-"
 2.76      4.24     14.06     21.62     "assign"
 2.55      3.92     17.47     26.86     "nlsModel"
 2.35      3.62     21.35     32.82     "eval"
 2.15      3.30     96.10    147.72     "nls"
 2.00      3.08      8.99     13.82     "lapply"
 1.99      3.06      1.99      3.06     "as.integer.default"
 ...
```

これはしばしば驚くべき結果をもたらし、コンパイル済コードで置き換えることが望ましいボトルネックや R コードの小片を特定することに役立つ。

二つの警告：プロファイリングは少々余分の実行時間を課す。長時間の実行をプロファイルすると、出力ファイルは非常に長くなることがある。

4 システムと他言語間のインタフェイス

4.1 オペレーティングシステムへのアクセス

オペレーティングシステムへのアクセスは R 関数 `system` を経由する。詳細はプラットフォームにより異なる（オンラインヘルプを参照)。そして安全に仮定できるほとんど全ては、最初の引数は実行（必ずしもシェルによらない）のために引き渡される文字列 `command` で、第二引数は、もしそれが真なら、命令の出力を R の文字ベクトルに集める `internal` になるであろう、ことだけである。

関数 `system.time` が計時のために使える（しかしながら、利用できる情報は非ユニックスプラットフォームでは限られているかも知れない)。

4.2 インタフェイス関数 `.C` と `.Fortran`

この二つの関数は、作成時もしくは `dynload` により R にリンクされたコンパイル済みコードへの標準インタフェイスを提供する（see Section 4.3 [dyn.load と dyn.unload], page 21)。これらは元来、それぞれコンパイル済みの C と Fortran コードを対象としている。しかし、 `.C` 関数は C へのインタフェイスを生成できる他の言語に対しても使用できる、例えば C++ （see Section 4.5 [C++ コードとのインタフェイス], page 23)。

各々の関数への最初の引数は C や Fortran が理解できるシンボル名、つまり関数名やサブルーティン名、を与える文字列である（ロードテーブル中のシンボル名への対応は関数 `symbol.C` と `symbol.For` で与えられる；シンボルが読み込まれているかどうかは、例えば、`is.loaded(symbol.C("loglin"))` で検査できる。）

コンパイル済みコードに引き渡される R オブジェクトを与えるその他の引数は最大 65 個許される。通常これらは引き渡される前にコピーされ、コンパイル済みコードが値を返すとき、R リストに再びコピーされる。もし引数が名前を与えられているならば、これらは返されるリストオブジェクト中の成分の名前として使われる（しかしコンパイル済みコードには引き渡されない)。

次の表は R ベクトルのモードと C 関数もしくは Fortarn サブルーティンへの引数の型間の対応を与える。

R 保持モード	C での型	Fortran での型
`logical`	`int *`	`INTEGER`
`integer`	`int *`	`INTEGER`
`double`	`double *`	`DOUBLE PRECISION`
`complex`	`Rcomplex *`	`DOUBLE COMPLEX`
`character`	`char **`	`CHARACTER*255`

C の型 `Rcomplex` は `double` 型のメンバ `r` と `i` を持つ構造体であり、‘`R.h`’ により読み込まれるヘッダファイル ‘`Complex.h`’ 中で定義されている。単一の文字列だけが Fortran へ渡され、そして Fortran から戻される。これがうまく行くかどうかはコンパイラに依存する。他の R オブジェクトを `.C` へ引き渡すことができるが、他のインタフェイスのどれかを用いる方が好ましい。例外は R 関数を `call_R` とともに使うために引き渡す場合で、オブジェクトは `call_R` を用いて `void *` として処理される。この場合でも `.Call` の方が好ましい。

保持モード `double` の数値ベクトルを、属性 `Csinglw` を設定する（一番簡単には R の関数 `as.single`、 `single`、`storage.mode` を用いる）ことにより、`float *` として C へ、もしくは

REAL として Fortran へ引き渡すことが可能である。これは既存の C もしくは Fortarn コードへのインタフェイスを助けることだけを意図している。

形式的引数 NAOK が真でない限り、他の全ての引数は NA、そして IEEE 特殊値 NaN, Inf, -Inf があるかどうか検査され、こうした値が存在するとエラーになる。もし真ならば、これらの値は検査無しで引き渡される。

引数 DUP をコピーを抑制するために使うことができる。これは危険である：その使用に対する議論についてはオンラインヘルプを見よ。もし DUP=TRUE ならば、数値ベクトルを float * もしくは REAL として引き渡すことは不可能である。

最後に、引数 PACKAGE は特定の共用ライブラリのために、シンボル名を探すことを抑制する（もしくは R 中にコンパイルされるコードには "base" を用いよ）。これは極めて望ましい措置である。なぜなら、二つのパッケージ作成者が同じシンボル名を使うことを避ける方法は無く、そうした名前の衝突は R のクラッシュを引き起こすのに十分だからである。

コンパイル済みコードはその引数を通じて以外如何なるものも返すべきではないことを注意しよう：C 関数は void 型である必要があり、Fortran プログラムはサブルーティンでなければならない。

考えをまとめるために、二つの有限列を畳み込む非常に簡単な例を考えよう。（これはインタープリタ言語である R では高速に実行することが困難であるが C コードでは簡単である。）.C を用いて次のようにすることができるであろう

```
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];
}
```

これは R から次のように呼び出される

```
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
```

全ての R の保持モードを正しいものにするため、強制変換をする必要があることを注意しよう；型の対応が間違っていると把握困難な誤った結果に導く可能性がある。

4.3 dyn.load と dyn.unload

R と共に使われるコンパイル済みコードは共用ライブラリ（Unix, 詳細は see Section 4.4 [共用ライブラリの作成], page 22 ）、もしくは DLL (Windows) としてロードされる。

ライブラリもしくは DLL は dyn.load によりロードされ、dyn.unload により切り離される。切り離しは普通必要ではないが、Windows を含む幾つかのプラットフォームでは DLL を再構築するのに必要である。

二つの関数への最初の引数はライブラリへのパスを与える文字列である。利用者はライブラリに対する特定のファイル拡張子（例えば ‘.so’）を仮定すべきではなく、プラットフォームに依存せぬようにするため

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

という構文を使うべきである。Unix システムでは dyn.load に与えられるパスは、絶対パスでも、現ディレクトリに関する相対パスでも良く、もし ‘~’ で始まるのならユーザのホームディレクトリに関する相対パスでも良い。

ロードは最も普通にはパッケージの .First.lib 関数中の library.dynam の呼び出しで行われる。これは形式

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

を持ち、ここで libname は拡張子を省いたライブラリ/DLL の名前である。

幾つかの Unix システムではライブラリがロードされるとき、シンボル名をどのように解決するかに関し選択肢があり、引数 local と global によって制御される。これは真に必要なときだけ使うこと：特に、now=FALSE を用いた後に未解決のシンボルを呼び出すと R はぶしつけに停止するであろう。

もしあるライブラリや DLL が重複してロードされると、最新のものが使われる。より一般的には、同じシンボル名が複数のライブラリに現れると、最も最近にロードされたものが使われる。引数 PACKAGE はどのライブラリのものを使うかに関する曖昧さを避ける良い方法を提供する。

4.4 共用ライブラリの作成

Unix では R に読み込まれる共用ライブラリは R CMD SHLIB を用いて作り出される。これは引数としてオブジェクトファイル（拡張子 ‘.o’）もしくは C または Fortran のソース（それぞれ拡張子 .c または .o）であるファイルのリストを受け付ける。使用法については、命令 *R CMD SHLIB --help* を実行するか、SHLIB に対するオンラインヘルプを見よ。もしソースファイルのコンパイルがうまく行かなければ、コンパイルディレクトリ中のファイル ‘Makevars’ の中に幾つかの変数を設定するか、コンパイルディレクトリ中に必要な規則を書いた ‘Makefile’ を書くことにより（もしくは、勿論命令行から直接オブジェクトファイルを生成しても良い）、追加のフラグを指定することができる：例えば PKG_CPPFLAGS （C プリプロセッサ用、典型的には ‘-I’ フラグ）、PKG_CFLAGS と PKG_FFLAGS （それぞれ C と Fortran コンパイラ）。同様に、共用ライブラリを構築する際に、追加の ‘-l’ や ‘-L’ フラグをリンカに引き渡すのに、‘Makevars’ 中の変数 PKG_LIBS を使うことができる。

もし追加パッケージ *pkg* が C または Fortran コードをその ‘src’ サブディレクトリに含んでいたら、命令 R CMD INSTALL は共用ライブラリ（パッケージ中の .First.lib を用いて R 中にロードされる）を記述の R CMD SHLIB 機構を用いて自動的に作成するか、もしディレクトリ ‘src’ が Makefile を含めば make を用いて作成する。どちらの場合も、もし ‘Makefile’ が存在すれば、make を起動する場合それが最初に読み取られる。もし ‘Makefile’ が真に必要か用意されているならば、作成される共用ライブラリは R がリンクされている全ての Fortran ライブラリにたいしリンクされていることを保証する必要がある；make 変数 FLIBS がこの情報を含む。

Windows に於ける同値な機能は命令 Rcmd SHLIB である；もし存在すれば、ファイル ‘Makevars.win’ もしくは ‘Makefile.win’ が ‘Makevars’ もしくは ‘Makefile’ よりも好ましい。（これはソースファイルをインストールするのに Windows の R バイナリを必要としない。）

4.5 C++ コードとのインタフェイス

次のような仮想的な C++ ライブラリを考えよう：これは二つのファイル ‘X.hh’ と ‘X.cc’ からなり、R 中で使いたい二つのクラス X と Y をインプリメントする。

```
// 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; }
```

R と共に使うため、我々がしなければならない唯一つのことは、wrapper 関数を書き、それが以下に含まれることを保証することである

```
extern "C" {

}
```

例えば、

```
// X_main.cc:

#include "X.hh"

extern "C" {

void X_main () {
  X x;
}

}
```

コンパイルとリンクは（C のコンパイラとリンカ、もしくはリンカ自身ではなく）C++ のコンパイラとリンカを用いてなされなければならない；さもなければ、C++ の初期化コードが呼び出され（したがって静的変数 `Y` のコンストラクタが呼び出され）ない。適切に構築された Unix システム（バージョン 1.1 の C++ がサポートされた）では単に次のようにすれば

```
R CMD SHLIB X.cc X_main.cc
```

共用ライブラリ、典型的には ‘`X.so`’、が作られる（ファイルの拡張子名は使用プラットで異なるかもしれない）。ここで R を起動すれば次のようになるであろう。

```
R : Copyright 2000, The R Development Core Team
Version 1.1.0 Under development (unstable) (April 14, 2000)
...
Type    "q()" to quit R.

R> dyn.load(paste("X", .Platform$dynlib.ext, sep = ""))
constructor Y
R> .C("X_main")
constructor X
destructor X
list()
R> q()
Save workspace image? [y/n/c]: y
destructor Y
```

R for Windows FAQ (‘`rw-FAQ`’) は、この例を様々な Eindows 用コンパイラで如何にコンパイルするかに関する詳細を含む。

この例におけるような、C++ の iostream の使用は極力避ける方が良い。出力が R のコンソールに現れる保証は全くないし、実際 Windows 用の R のコンソールには現れない。可能な限り全ての I/O に対し R のコードを使うか、C の entry point (see Section 5.5 [表示], page 47) 機能を使うこと。

4.6 C 中で R のオブジェクトを扱う

R 関数の実行を加速するために C コードを使うことはしばしば非常に有効である。伝統的にはこれは R の `.C` 関数を経由して行われる。このインタフェイスの一つの制限は R オブジェクトは C 中で直接には処理できないことである。これは C コード中から R 関数を呼び出したいとき一層面倒になる。これを行うために `call_R` と呼ばれる C の関数（S との互換性のために同様に `call_S` という名前も使える）があるが、使うのは面倒であり、ここで説明される機構の方が使いやすいと同時に、より強力である。

内部的な R のデータ構造を使う C コードを本当に必要とするならば、.Call と .External 関数を用いてこれを実行することができる。それぞれの場合において、R からの関数呼び出しの構文は .C のそれと同様であるが、二つの関数はかなり異なる C インタフェイスを持つ。一般的に .Call インタフェイス（バージョン 4 の S の同じ名前のインタフェイスを真似た）は使用が少々単純であるが、.External は少々一般的である。

.Call の呼び出しは .C ととても良く似ている、例えば

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

第一引数は、既に R にロードされている C のコードのシンボル名を与える文字列でなければならない。最大 65 個の R のオブジェクトを引数として引き渡すことができる。インタフェイスの C 側は以下のようになる

```
#include <R.h>
#include <Rinternals.h>

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

.External の呼び出しはほとんど同じである

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

しかしインタフェイスの C 側は異なり、唯一つの引数を持つ

```
#include <R.h>
#include <Rinternals.h>

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

ここで args は、引数を取り出すことができる Lisp 風のリストである LISTSXP である。

どちらの例でも R オブジェクトは、ヘッダファイル ‘Rinternals.h’ 中で定義された一組の関数やマクロ、もしくは ‘Rdefines.h’ で定義されたより高水準のマクロを用いて操作することができる。.Call と .External に関する詳細は以下で更に説明される。

.Call や .External を使うことにする前に他の可能性を見ておくべきである。最初に、インタープリタ言語である R コードを使う；もしこれが十分早ければ、これが通常最良の選択肢である。同様に .C の使用で十分かどうか見るべきである。もし実行課題が単純で R の呼び出しが不要ならば .C で十分である。新しいインタフェイスは S と R に最近付け加えられたもので、それが使用可能になる前から、多量の有用なコードが .C だけを使って書かれてきた。.Call と .External インタフェイスはより多くの制御を可能にするが、同時により多くの責任を課し、注意深く使われなければならない。

より最近のやり方はバージョン 4.4 の S のインタフェイス .Call マクロと関数の R 版を使うことである。これはヘッダファイル ‘Rdefines.h’ 中に定義されている。これは多少より簡単な接近法であり、もしコードがあらゆる段階で S と共用されるのであれば、確かに好ましいやり方である。

R の本質的な部分がここで説明されている関数とマクロを用いて移植されている。したがって R ソースコードは豊富な例と「如何にすべきか」を提供する：実際、ここでの例の多くが、似たような課題を実行する R のシステム関数を注意深く調べることにより開発された。インスピレーションを得るために、ソースコードを利用しよう。

C コード中でどのように R オブジェクトを取り扱うかに付いて幾つかのことを知る必要がある。処理すべき全ての R オブジェクトは型 *SEXP*[1] を持つ、これは定義型 SEXPREC を持つ構造体への

[1] SEXP は *S*imple *EXP*ression, common in LISP-like language syntaxes の簡略形である。

ポインタである。これは R オブジェクトの通常の全てのタイプ、つまり様々なタイプのベクトル、関数、環境、を処理できる 可変な型 と考えると分かりやすい。詳細はこの節の後で与えられるが、多くの目的に取って、プログラマはそれを知っている必要はない。それよりも、Visual Basic で用いられるようなモデルを考えよう。そこでは R オブジェクトは可変型として C コード中で処理（R のインタプリタコードにおけるように）され、適当な部分が、例えば数値計算のために、必要なときだけ取り出される。R のインタプリタコードにおけるのと同様に、可変な型のオブジェクトを正しい型に直すために強制変換が頻繁に行われる。

4.6.1 ガベージコレクションの影響を処理する

R がメモリの割当を行う方法に付いて多少知っておく必要がある。R オブジェクトに割り当てられたメモリはユーザによって開放できない。代わりに、メモリは適宜ガベージコレクションされる. つまり、使用されていない全てのメモリは開放される。(R 1.2 以前では。オブジェクトは移動もされるか。)

R のオブジェクト型は 'Rinternals.h' 中の定義型 SEXPREC で定義される C の構造体で表現される。これは幾つかのものを含み、就中データブロックや他の SEXPREC へのポインタを含む。現在のメモリ処理スキームではデータブロックはガベージコレクション中に移動されるかもしれず、それらへのポインタは更新される。このために、あるオブジェクトの構造中のデータブロックへのポインタを保存し再使用することは安全ではない。SEXP は単に SEXPREC へのポインタである。

もし C コード中に R オブジェクトを作ったら、オブジェクトへのポインタに PROTECT マクロを用い R にオブジェクトを使用することを知らせる必要がある。これは R にそのオブジェクトが使用中であることを告げ、そのためそれが破壊されることはなくなる。保護されるのはオブジェクトでありポインタ変数ではないことを注意しよう。ある時点で PROTECT(p) を行えばそれ以降 p が保護されると信じることは良くある間違いであり、一旦新しいオブジェクトが p に割り当てられれば真ではなくなる。

一つの R オブジェクトを保護すれば、自動的に対応する SEXPREC 中のポインタが指す全ての R オブジェクトが保護される。

PROTECT の呼び出しに責任を負うのはプログラマだけである。対応してマクロ UNPROTECT があり、引数としてもはや不要になり保護を解除するオブジェクトの数を int として与える。保護機構はスタックに基づく。したがって UNPROTECT(n) は保護されている直近の n 個のオブジェクトを保護解除する。PROTECT と UNPROTECT への呼び出しは、ユーザのコードが終了する際釣合が取れていなければならない。R はもし管理がまずければ警告 "stack imbalance in .Call" （もしくは .External）を受ける。

以下は C コード中で R の数値ベクトルを作る小さな例である。最初に 'Rdefines.h' 中のマクロを使う：

```
#include <R.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);
```

そしてそれから 'Rinternals.h'中のマクロを使う：

```
#include <R.h>
#include <Rinternals.h>

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

ここで、読者はこうした操作の間にどうすれば R オブジェクトを除去できるか知りたいかも知れない。動作しているのは正に C コードだからである。結論としては、この場合は保護なしで行えば良い。しかし一般的にいって、我々は使用中の R のマクロや関数の背後に何があるか知らない（それどころか知りたくもない）し、それらの一つがメモリ割当を引き起こし、したがってガベージコレクションにより、我々のコード `ab` を取りさっ（移動し）てしまうかも知れない。通常注意深くしすぎる方が良く、R のマクロや関数がそのオブジェクトを除去（移動）するかも知れないことを仮定した方が良い。

ある場合には、保護が本当に必要かどうか慎重に監視する必要がある。特に大量のオブジェクトが生成される場合は注意が必要になる。ポインタの保護スタックは固定サイズ（既定値 10,000）であり、いっぱいになる可能性がある。したがって、見える限りの全てを `PROTECT` し、最後に数千のオブジェクトを `UNPROTECT` するのは良いやり方ではない。オブジェクトを他のオブジェクトの一部に割り当て（自動的に保護される）たり、使用後に直ちに保護解除することは、ほとんど必ず可能である。

既に使用中であることを R が知っているオブジェクトに対しては保護は不要である。特に、これは関数の引数について当てはまる。

より使われることの少ないマクロ `UNPROTECT_PTR(`s`)` があり、`SEXP` s によりポイントされているオブジェクトを、たとえそれがポインタ保護スタックの最上部になくても、保護解除する。これはパーサ以外ではほとんど必要でない（R のソースの ‘`plot3d.c`’ 中に一つの例がある）。

4.6.2 メモリ割り当ての保管

多くの目的に取って、R オブジェクトを割り当て、それらを操作するだけで十分である。ほんの少しの `alloc`*Xxx* 関数が ‘`Rinternals.h`’ 中に定義されている—それらを調べてみたくなるかも知れない。これらは様々な型の R オブジェクトを割り当て、標準的なベクトル型に対しては、‘`Rdefines.h`’ 中で定義された `NEW_`*XXX* マクロがある。

計算途中に C オブジェクトに対するメモリ割り当てが必要になったら、`R_alloc` を呼び出すのが最善である；see Section 4.6.2 [メモリ割り当ての保管], page 27。これら全てのメモリ割り当てルーチンはそれ自身のエラー検査機能を持つので、もしメモリが割り当て不能ならエラーが表示され呼び出しが戻らないことをプログラマは仮定して良い。

4.6.3 R の型の詳細

‘`Rinternals.h`’ 中で定義されたマクロを使うには R の型が内部 的に如何に知られるのかを知る必要がある：これはもし ‘`Rdefines.h`’ 中のマクロを使うのであれば、多かれ少なかれ完全に隠されている。

異なった R データタイプは C 中で *SEXPTYPE* により表現されている。幾つかは R でおなじみのものであり、幾つかは内部的なデータ型である。通常の R オブジェクトのモードが次の表に与えられている。

SEXPTYPE	R の対応型
REALSXP	型 double の数値
INTSXP	整数
CPLXSXP	複素数
LGLSXP	論理値
STRSXP	文字
VECSXP	リスト（総称的ベクトル）
LISTXP	“dotted-pair” リスト
DOTSXP	‘...’ オブジェクト
NILSXP	NULL
SYMSXP	名前/シンボル
CLOSXP	関数または関数の closure
ENVSXP	環境

重要な内部 SEXPTYPE には LANGSXP, CHARSXP 等がある。

引数の型に関し完全に得心していない限り、コードはデータ型を検査すべきである。時には C 中である R 表現式を評価することにより作られたオブジェクトのデータ型を検査することも必要になるかも知れない。型検査のためには isReal, isInteger そして isString といった関数を使うことができる。その他の類似の関数についてはヘッダファイル ‘Rinternals.h’ を見よ。これら全ては SEXP を引数に取り、*TRUE* もしくは *FALSE* を意味する 1 および 0 を返す。再びこれを行う二つの方法があり、‘Rdefines.h’ は IS_NUMERIC といったマクロを定義する。

もし SEXP が正しい型でないと何が起こるのだろうか？時折エラーを作り出す以外の選択肢がないことがある。このためには関数 error を使うことができる。通常オブジェクトを正しい型に強制変換することが望ましい。例えば、もし SEXP が INTEGER 型であることが分かったが、REAL 型のオブジェクトが必要だったとしよう。型を変換するには次の同値な方法がある

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

または

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

新しい オブジェクト が作られたときは保護が必要になる；SEXP により最初ポイントされていたオブジェクトは再使用されても依然として保護されているが、しかし今や使われていない。

全ての強制変換関数は自分自身でエラー検査を行い、必要に応じて警告と共に NA を生成したり、エラー停止を行う。

これまでのところ、如何に R オブジェクトを C コードから生成するか、そして如何に R の数値ベクトルから数値データを取り出すか、だけを見てきた。これらは我々が R オブジェクトを数値アルゴリズムにインタフェイスするのに十分である。しかし、有用なオブジェクトの返り値を作り出すにはもう少し知る必要がある。

4.6.4 属性

多くの R のオブジェクトは属性を持つ： 最も有用なものはクラスと、オブジェクトを行列や配列としてマークする dim と dimnames である。また、ベクトルの names 属性を用いて作業すると便利である。

このことを説明するために、二つのベクトルの外積を取るコードを書いてみよう (そのためには既に outer と %o% があるが)。 いつものように R コードは単順である。

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

ここで我々は x と y はひょっとすると名前を持つ数値ベクトルと仮定する。今回は R コードを呼び出すとき強制変換を行う。

計算を行う C コードは以下のようである

```
#include <R.h>
#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);
}
```

しかしながら結果に dimnames 属性を設定したいかも知れない。allocMatrix が近道を提供するが、dim 属性をどのように直接設定するかを紹介しよう。

```
#include <R.h>
#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(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];
  }
  PROTECT(dim = allocVector(INTSXP, 2));
  INTEGER(dim)[0] = nx; INTEGER(dim)[1] = ny;
  setAttrib(ans, R_DimSymbol, dim);

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

この例は幾つかの新しい特徴を紹介する。関数 `getAttrib` と `setAttrib` は個々の属性を得、そして設定する。第二引数は我々が要求する名前を属性のシンボル表に定義する `SEXP` である。これらおよびもっと多くのそうしたシンボルがヘッダファイル '`Rinternals.h`' に定義されている。

ここでも近道がある: 関数 `namesgets`, `dimgets` そして `dimnamesgets` は `names<-`, `dim<-` そして `dimnames<-` の内部形であり、また `GetMatrixDimnames` や `GetArrayDimnames` といった関数がある。

予め定義されていない属性を付加したら何が起こるであろうか？われわれはそれに対するシンボルを `install` への呼び出し経由で付け加える必要がある。例として値 `3.0` を持つ属性 `"version"` を付け加えたいとしよう。次のようにすることが考えられる

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

不要なときに `install` を使うのは害がなく、もしそれが既にシンボル表に定義されていたならばシンボルを検索する簡単な方法を提供する。

4.6.5 クラス

R ではクラスは単に名前 "class" を持つ属性であり、そうしたものとして扱うことが出来るが classgets という近道がある。我々の例の返り値に "mat" というクラスを与えたいとしよう。次のようにすれば良い

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

値は文字ベクトルであるから、C の文字配列からそれをどのように作り出すかをする必要がある。これには関数 mkChar を用いれば良い。

4.6.6 リストの処理

リストを扱うには少々注意がいる。なぜなら R は LISP 風のリスト（現在 “pairlists” と呼ばれる）の使用から S 風の総称的なベクトルの使用に移行したからである。結果として、モード list のオブジェクトに対する適切なテストは isNewList であり、allocList(n) では「なく」allocVector(VECSXP, n) を必要とする。

リストの成分は総称的ベクトルに直接アクセスすることにより、値を得たり設定したりできる。次のようなリストオブジェクトがあるとしよう

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

すると a$g は a[[2]] として次のようにアクセスできる

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

これはすぐに面倒になるかも知れない。そこで、次の関数（パッケージ **nls** 中の一つに基づく）が非常に有用になる：

```
/* str という名前のリスト成分を得る、さもなければ NULL を返す */

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_ELT(names, i)), str) == 0) {
      elmt = VECTOR_ELT(list, i);
      break;
    }
  return elmt;
```

```
}
```

そして次のように使うことができる

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

4.6.7 変数を見付ける・設定する

我々の C 計算で必要な全ての R オブジェクトは .Call もしくは .External 引数として渡されるのが普通であろう。しかしながら R オブジェクトの値を、それらの名前を与えることにより、C の内部から見い出すことが可能である。次のコードは get(name, envir = rho) と同値である。

```
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_ELT(name, 0))), rho);
  printf("first value is %f\n", REAL(ans)[0]);
  return(R_NilValue);
}
```

主たる仕事は findVar によってなされる。しかしながらそれを使うためには、name を名前としてシンボル表中に登録する必要がある。値は内部的にしか必要なかったから、値 NULL を返す。

次の構文を持つ同様の関数

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

を、R オブジェクトに値を代入することに使うことができる。defineVar は特定の環境フレーム中に新しい結合を作ったり、既存の結合の値を変更する；これは assign(symbol, value, envir = rho, inherits = FALSE) と同値である。setVar は rho 中もしくはそれを含む環境中で symbol に対する既存の結合を探索する。もしある結合が見付かれば、その値が value に変更される。さもなければ、グローバルな環境中で新しい結合が生成される。これは assign(symbol, value, envir = rho, inherits = TRUE) に対応する。

4.6.8 R のバージョン 1.2 に於ける変更点

バージョン 1.2.0 の R は新しい “generational” なガベッジコレクションを導入した。これは、文字列とベクトル (そして言語オブジェクト) が数値原子的タイプとは異なって処理されることを意味する。

これまでのコードは次のようなスタイルで書かれた

```
VECTOR(dimnames)[0] = getAttrib(x, R_NamesSymbol);
```

しかしながら、これはもはや許されない。関数 VECTOR_ELT と SET_VECTOR_ELT が総称的ベクトルの要素にアクセスし設定するために使われなければならない。これらは文字ベクトルに対する STRING_ELT と SET_STRING_ELT に類似の関数である。

既存のコードを変換するためには次の置き換えをせよ。

Original	**Replacement**
`foo = VECTOR(bar)[i]`	`foo = VECTOR_ELT(bar, i)`
`VECTOR(foo)[j] = bar`	`SET_VECTOR_ELT(foo, j, bar)`
`foo = STRING(bar)[i]`	`foo = STRING_ELT(bar, i)`
`STRING(foo)[j] = bar`	`SET_STRING_ELT(foo, j, bar)`

新しいガベージコレクション機能は SEXP を含む欄への全ての代入が SET_VECTOR_ELT といった関数経由でなされることを要求する。古いバージョンの R でコンパイルされたパッケージはこれらの欄を直接変更するバイナリコードを含むかも知れない。こうしたパッケージは再コンパイルし、再インストールされなければならない。そうしないと、メモリー管理機能を混乱させ、メモリーへの蓄積がおかしくなる結果を招く。

R の 1.x 系列において最大の互換性を達成するために、アドオンパッケージは次の形の条件式を使うか

```
#if R_VERSION >= R_Version(1, 2, 0)
  new-style-code
#else
  old-style-code
#endif
```

もしくは、不用なコードの重複を避けるため、新しいスタイルのインタフェイスと共に次のコードを含む個人的な ヘッダファイルを使うのがより好ましい：

```
#if R_VERSION < R_Version(1, 2, 0)
# define STRING_ELT(x,i)         (STRING(x)[i])
# define VECTOR_ELT(x,i)         (VECTOR(x)[i])
# define SET_STRING_ELT(x,i,v)   (STRING(x)[i] = (v))
# define SET_VECTOR_ELT(x,i,v)   (VECTOR(x)[i] = (v))
#endif
```

より一層の変更が言語オブジェクトを処理するコードで必要になる。

4.7 インタフェイス関数 .Call と .External

この節では R/C インタフェイスの詳細を議論する。

これら二つのインタフェイスはほぼ同じ機能を持つ。.Call はバージョン 4.0 の \S の同名のインタフェイスに準拠し、.External は .Internal に基づく。.External はより複雑であるが、個数が変わり得る引数を許す。

4.7.1 .Call の呼び出し

我々の有限畳み込みの例を、先ず 'Rdefines.h' マクロにより、.Call を使うように変更しよう。R における呼び出し関数は

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

これはこれ以上簡単にならない程簡単であるが、先で見るように全ての型の検査を C コードに次のように渡さなければならない

```
#include <R.h>
#include <Rdefines.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);
}
```

S のバージョン 4 と異なり、これらのマクロの R 版は強制変換が可能かどうかチェックし、それが失敗するとエラーを起こす。強制変換の結果欠損値が持ち込まれると警告が出される。ここでは強制変換が C コードで行われる例を示したが、必要な強制変換を R コードで行う方がしばしばより簡単である。

次に R の内部スタイルをあげる。C コードだけが変更される。

```
#include <R.h>
#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(REALSXP, nab));
  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);
}
```

これは前と全く同じ様に呼び出すことが出来る。

4.7.2 .External の呼び出し

同じ例で .External を説明する。R コードの変更点は.Call を .External に変えるだけである。

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

しかしながら、主な変更点は引数が C コードに引き渡される仕方であり、こんどは単一の SEXP だけである。C コードの唯一つの変更は引数をどう処理するかという点である。

```
#include <R.h>
#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));
    ...
}
```

再び引数を保護する必要は無い。インタフェイスの R 側ではそれらは既に使用されているオブジェクトだからである。次のマクロ

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

は最初の四つの引数にアクセスする簡便な方法を与える。より一般的には、CDR および CAR マクロを

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

の様に使うことが出来、これは明らかにどれだけでも多くの引数を取り出すことを可能にする (しかしながら、 .Call は 65 個という決して少ない数ではないが限界を持つ)。

もっと役に立つこととして、.External インタフェイスは可変個数の引数を持つ呼び出しを処理する簡単な方法を提供する。なぜなら length(args) が供給される引数の数 (そのうち、最初のものは無視される) を与えるから。実際の引数に与えられた名前 ('tags') を知りたくなるかも知れない。これは、TAG マクロと次の例のようなことを行えば可能になる。これは、引数のなまえと、もしそれががベクトル型であればその最初の値を印字する。

```
#include "R_ext/PrtUtil.h"

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:
        Rprintf("[%d] '%s' %f\n", i+1, name, REAL(CAR(args))[0]);
        break;
      case LGLSXP:
      case INTSXP:
        Rprintf("[%d] '%s' %d\n", i+1, name, INTEGER(CAR(args))[0]);
        break;
      case CPLXSXP:
        cpl = COMPLEX(CAR(args))[0];
        Rprintf("[%d] '%s' %f + %fi\n", i+1, name, cpl.r, cpl.i);
        break;
      case STRSXP:
        Rprintf("[%d] '%s' %s\n", i+1, name,
                CHAR(STRING_ELT(CAR(args), 0)));
        break;
      default:
        Rprintf("[%d] '%s' R type\n", i+1, name);
      }
    }
  }
  return(R_NilValue);
}
```

これは次の wrapper 関数を使って呼び出せる。

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

このスタイルのプログラミングは便利であるが必要ではないことを注意しよう。なぜならもう一つの次のスタイルが可能だからである。

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

4.7.3 欠損値と特殊値

.C 呼び出しが行うエラー検査の一つが (NAOK が真でない限り) 欠損値 (NA) と IEEE 特殊値 (Inf, -Inf そして NaN) の検査であり、もし一つでも見付かればエラーを与える。.Call インタフェイスによりこれらは我々のコードに引き渡される。この例では IEEE 算術が適正な処理を行うので特殊値は問題を引き起こさない。現在の移植では NA も問題を起こさない。なぜなら、それは NaN の一つの

型だから。しかし、そうした詳細に頼ることは賢明ではない。したがって、'R.h' により取り込まれる 'Arith.h' 中で定義されたマクロを用い NA を処理するようにコードを書き直してみよう。

コードの変更は convolve2 と convolveE の全てのバージョンでおなじである:

```
    ...
  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];
    ...
```

ISNA マクロと、同様のマクロ ISNAN (NaN と NA を検査) と R_FINITE (NA と全ての特殊値で偽となる) は double 型の数値にだけ適用されることを注意しよう。整数、論理値 そして文字列は定数 NA_INTEGER, NA_LOGICAL そして NA_STRING と等しいかどうかで検査できる。これらと NA_REAL は R のベクトルを NA に設定するのに使える。

定数 R_NaN, R_PosInf, R_NegInf そして R_NaReal は double を特殊値に設定することに使える。

4.8 R の表現式を C から評価する

先に call_R インタフェイスが R の表現式を C コードから評価するのに使えることを注意した。しかし、現在のインタフェイスはもっと使いやすい。我々が使う主な関数は次のようなものである

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

これは R インタプリタのコード eval(expr, envir = rho) と同値であるが、findVar, defineVar そして findFun (検索対象を関数に制限) を同じように使うことが出来る。

これがどのように使えるかを見るために、表現式に対する lapply の内部的な単純版を考えてみる。これは次のように使用でき

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

以下の C コードを持つ

```
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_ELT(list, i), rho);
    SET_VECTOR_ELT(ans, i, eval(expr, rho));
  }
  setAttrib(ans, R_NamesSymbol, getAttrib(list, R_NamesSymbol));
  UNPROTECT(1);
  return(ans);
}
```

もし表現式ではなく関数に使えたら、もっと lapply に近くなるであろう。これを行う一つの方法は次の例における R インタプリタコードを使うことである。しかし、(もし少々分かりにくいなら) これを C コード中で行うことも可能である。つぎは 'src/main/optimize.c' 中のコードに基づいている。

```
SEXP lapply2(SEXP list, SEXP fn, SEXP rho)
{
  int i, n = length(list);
  SEXP R_fcall, 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++) {
    SETCADR(R_fcall, VECTOR_ELT(list, i));
    SET_VECTOR_ELT(ans, i, eval(R_fcall, rho));
  }
  setAttrib(ans, R_NamesSymbol, getAttrib(list, R_NamesSymbol));
  UNPROTECT(2);
  return(ans);
}
```

これは次のように使う

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

関数 lang2 は二つの要素からなる実行可能な 'リスト' を作りだすが、しかしこれを理解するには LISP 風の言語の知識を必要とする。

4.8.1 零点を見付ける

この節では以前 call_Rの 'demos/dynload' (Becker, Chambers & Wilks (1988) 中の call_Sに基づく) にあった一変数関数の零点を見付ける例を再使用する。これは最早廃止された demo(dynload) の中で call_R の例として使われていたものである。R コードと例は以下のとおりである

```
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))
```

ここで今回は強制変換とエラー検査を R コードの中で行っている。C コードは次のようである

```
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;

    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;
        }
    }
}
```

C コードは本質的に `call_R` 版と同じで、単に `double` から `SEXP` への変換と `f.check` の評価のために幾つかの関数を使っているだけである。

4.8.2 数値微分の計算

`.External` の用法を説明するために少し長い例 (Saikat DebRoy による) を使おう。これは数値微分を計算するもので、インタープリタ R コードを使えば効率的に行えたであろうが、しかしより大規模な C 計算中で使えるかもしれない。

インタープリタ R 版と例は次のようになる

```
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"))
```

ここで expr は表現式、theta は変数名の文字ベクトル、そして rho は使用する環境である。

コンパイル版では R からの呼出しは次のようになるであろう

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

使用例は次のようになる

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

表現式を引用符で囲み、評価されるのを中止する必要があることを注意しよう。

以下は完全な C コードで節毎に説明される。

```
#include <R.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 = CADDDR(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_ELT(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));
    SET_VECTOR_ELT(dimnames, 1,  theta);
    dimnamesgets(gradient, dimnames);
    setAttrib(ans, install("gradient"), gradient);
    UNPROTECT(3); /* ans  gradient  dimnames */
    return ans;
  }
```

引数を処理するコードは

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

theta と rho が正しい型であることを検査したが、expr の型の検査をしないことを注意しよう。これは eval は EXPRSXP 以外の多くの R オブジェクトを処理できるからである。我々が行える有用な強制変換は無く、もし引数が正しいモードで無ければ実行はエラーメッセージとともに中断するであろう。

コードの最初のステップは環境 rho 中で次のように表現式を評価することである

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

次に計算された導関数用にメモリースペースを確保する

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

allocMatrix への最初の引数は行列の SEXPTYPE を与える。ここで必要なのは REALSXP である。他の二つの引数は行と列の数である。

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

次に for ループに入る。各引数毎にループを行う。for ループ中では、最初に STRSXP thetaの i 番目の要素に対応するシンボルを作り出す。ここで、STRING_ELT(theta, i) は STRSXP thetaの i 番目の要素にアクセスする。マクロ CHAR() はその実際の文字表現を取り出し、ポインターを返す。それから名前を登録しその値を見出すために findVar を使う。

```
    tt = REAL(par)[0];
    xx = fabs(tt);
    delta = (xx < 1) ? eps : xx*eps;
    REAL(par)[0] += delta;
    PROTECT(ans1 = coerceVector(eval(expr, rho), REALSXP));
```

最初にパラメータの値である実数値を取り出し、数値微分の近似に用いられる増分であるデルタ値を計算する。それから、par (環境 rho 中) 中に保管された値を delta で置き換え、再び環境 rho 中

で expr を評価する。ここでは本来の R のメモリー位置を直接操作しているので、R が変更されたパラメータ値に対する評価を行う。

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

次に、グラディエント行列の i 番目の列を計算する。どのようにそれがアクセスされるかを注意しよう。R は (FORTRAN の様に) 行列を列順に保管する。

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

先ずグラディエント行列の列名を加える。これは (VECSXPである) リストを保管することによりなされる。リストの最初の要素、行名、は NULL (既定) で、次の要素、列名、は theta と設定される。このリストはそれからシンボル R_DimNamesSymbol を持つ属性として設定される。最後にグラディエント行列を ans のグラディエント属性で設定し、その他の保護されたメモリー位置を保護解除し、答え ans を返す。

4.9 コンパイル済みのコードのデバッグ

遅かれ早かれプログラマーは R に読み込まれたコンパイル済みのコードのデバッグをする必要に迫られる。いくつかの“トリック”が役に立つ。

4.9.1 動的に読み込まれたコード中のエントリポイントを見付ける

ほとんどのコンパイル環境で、R に読み込まれたコンパイル済みコードは、それが実際に読み込まれるまでその中にセットされた中断点を持つことが出来ない。UNIX 下でそのような動的に読み込まれたコードに対するシンボリックデバッガーを使うためには以下を用いる

- 例えば *R -d gdb* により、R の実行プログラムに対してデバッガーを呼びだす。
- R を起動する。
- R のプロンプトに対し、あなたのライブラリーを読み込むために dyn.load か library を使う。library.
- 中断信号を送る。これによりデバッガーのプロンプトに戻ることが出来る。
- あなたのコードに中断点を設定する。
- R の実行を *signal 0*⟨RET⟩ をタイプすることにより続行する。

Windows 下では R の engine はそれ自身 DLL 中にあり、手順は

- WinMain に対する中断点を設定した後 R をデバッガー下で起動する。

```
gdb .../bin/Rgui.exe
(gdb) break WinMain
(gdb) run
```

```
[ R.dll を読み込んで中断 ]
(gdb) break R_ReadConsole
(gdb) continue
[ コンソールを動かしたまま停止する ]
(gdb) continue
```

- R のプロンプトに対し、ライブラリーを読み込むために dyn.load か library を使う。
- あなたのコード中に中断点を設定する。
- 設定された中断点を用いて実行を続行するためには次の様にする。

```
(gdb) clear R_ReadConsole
(gdb) continue
```

Windows は信号に対するサポートをほとんど持っておらず、デバッガー下でプログラムを実行し、それを中断するために一時停止信号を送り、デバッガーに制御を戻すことはある種のデバッガーでのみ可能である。

4.9.2 デバッグ時の R オブジェクトの精査

R オブジェクトをコンパイル済みコードから探す基本は関数 PrintValue(SEXP s) であり、これは s が指し示す R オブジェクトを表示する正規の R の表示機構を使う。もしくはより確実な方法としては 'objects' だけを表示する R_PV(SEXP s) を使う。

PrintValue をつかう一つの場合はデバッグ用にコードに適当な呼び出しを挿入することである。

もう一つの方法は R_PV をシンボリックデバッガーから呼び出すことである。(PrintValue は Rf_PrintValue として隠蔽されている。) 例えば、gdb から次のように使う

```
(gdb) p R_PV(ab)
```

ここでオブジェクト ab は畳み込みの例から取ったもので、適当な中断点を畳み込みの C コードに置く。

任意の R オブジェクトを検査するには、もう少し工夫がが必要になる。例えば次のようにしよう

```
R> DF <- data.frame(a = 1:3, b = 4:6)
```

中断点を do_get に置き、R のプロンプトに *get("DF")* とタイプすると、DF のメモリ中のアドレスを得ることが出来る。例えば

```
Value returned is $1 = (SEXPREC *) 0x40583e1c
(gdb) p *$1
$2 = {
  sxpinfo = {type = 19, obj = 1, named = 1, gp = 0,
    mark = 0, debug = 0, trace = 0, = 0},
  attrib = 0x40583e80,
  u = {
    vecsxp = {
      length = 2,
      type = {c = 0x40634700 "0>X@D>X@0>X@", i = 0x40634700,
        f = 0x40634700, z = 0x40634700, s = 0x40634700},
      truelength = 1075851272,
    },
    primsxp = {offset = 2},
    symsxp = {pname = 0x2, value = 0x40634700, internal = 0x40203008},
    listsxp = {carval = 0x2, cdrval = 0x40634700, tagval = 0x40203008},
```

```
      envsxp = {frame = 0x2, enclos = 0x40634700},
      closxp = {formals = 0x2, body = 0x40634700, env = 0x40203008},
      promsxp = {value = 0x2, expr = 0x40634700, env = 0x40203008}
    }
  }
```

(より可読性を高めたデバッガの出力)。

R_PV() を用いて SEXP の様々な要素の値を “吟味” することが出来る。例えば

```
(gdb) p R_PV($1->attrib)
$names
[1] "a" "b"

$row.names
[1] "1" "2" "3"

$class
[1] "data.frame"

$3 = void
```

対応する情報が正確にはどこに保管されているかを見出すためには、“より精妙に” 行う必要がある:

```
(gdb) set $a = $1->attrib
(gdb) p $a->u.listsxp.tagval->u.symsxp.pname->u.vecsxp.type.c
$4 = 0x405d40e8 "names"
(gdb) p $a->u.listsxp.carval->u.vecsxp.type.s[1]->u.vecsxp.type.c
$5 = 0x40634378 "b"
(gdb) p $1->u.vecsxp.type.s[0]->u.vecsxp.type.i[0]
$6 = 1
(gdb) p $1->u.vecsxp.type.s[1]->u.vecsxp.type.i[1]
$7 = 5
```

5 R の API: C コードのエントリポイント

R の実行プログラム/DLL にはコードから呼び出せる多く (そして FORTRAN コードから呼び出せる幾つか) のエントリポイントがある。ここで説明されたものだけが十分安定しており、それを変更するには細心の注意がいる。

これらを使用する推奨される手順は C コード中にヘッダファイル 'R.h' を次のように取り込むことである

```
#include <R.h>
```

これは ディレクトリ '$R_HOME/include/R_ext' から幾つかの他のヘッダファイルを取り込む。、他にも取り込めるヘッダファイルがあるが、それらが含む特徴は文章化されておらず不安定であると見なされるべきである。

もう一つの方法はヘッダファイル 'S.h' を取り込むことで、それは S からのコードを転用する際に有用である。これは 'R.h' よりも僅かなものを取り込み、幾つかの互換用の定義を持つ (例えば S からの S_complex タイプ)。

'R.h' から取り込まれる全てを含む、これらのほとんどのヘッダファイルは C++ コードからも使うことが出来る。

> 注意: R はユーザーコードとの衝突を避けるためその外部名の多くを再定義するため、これらのエントリーポイントを使う際は適当なヘッダファイルを取り込むことが 不可欠である。

5.1 メモリ割り当て

C プログラマーが使用できる二つのメモリー割当法があり、一つは R が掃除の面倒を見るもので、もう一つはユーザーが全体を制御できる (従って責任もとる) ものである。

5.1.1 一時的保管用メモリ割当

ここでは R は .C への呼び出しの終わりにメモリーを再利用する。次のように使う

```
char* R_alloc(long n, int size)
```

これは各々サイズ *size* の *n* 単位のメモリーを確保する。使用法の典型的な例 (パッケージ **mva** から) は

```
x = (int *) R_alloc(nrows(merge)+2, sizeof(int));
```

S との互換性のために同様な呼び出し S_alloc があり、これは確保されたメモリーを零で埋める点だけが異なる。

```
S_realloc(char *p, long new, long old, int size)
```

これは確保サイズを *old* から *new* 単位に変え、追加の単位を零で初期化する。

このメモリーはヒープメモリーから取られ、.C, .Call または .External 呼び出しの最後に開放される。ユーザーがそれを処理することも出来、vmaxget への呼び出しで現在の位置を確認し、続いて vmaxset の呼び出しで確保メモリーをクリアする。これは熟達者にのみ勧められる。

5.1.2 ユーザー制御モリ

もう一つのメモリ確保機能は R のエラー処理を提供するインタフェイスである malloc へのインタフェイスである。このメモリはユーザーが開放するまで存続し、R の作業空間用に確保されたメモリに加えられる。

インタフエイス関数は

```
type* Calloc(size_t n, type)
type* Realloc(any *p, size_t n, type)
void Free(any *p)
```

であり、calloc, realloc そして free の類似物を提供する。もしエラーが生じればそれは R により処理され、したがってもしこれらのルーティンから無事に戻ることができればメモリは成功裡に確保され開放される。Free はポインタ p を NULL に設定する。(S の全てのバージョンはそうするとは限らない。)

5.2 エラー処理

基本的なエラー処理ルーティンは R コードの stop と warning に同値であり、同じインタフェイスを使う。

```
void error(const char * format, ...);
void warning(const char * format, ...);
```

これらは printf への呼び出しと同じ呼び出し系列を持つが、最も単純な場合、エラーメッセージを与える単一の文字列とともに呼び出すことが出来る。(もし文字列が % を含む場合はこれを行ってはいけない。さもなくば一つの書式として解釈されるかもしれない。)

また S と互換なインタフェイスがあり、次のような呼び出しを用いる

```
PROBLEM ...... ERROR
MESSAGE ...... WARN
PROBLEM ...... RECOVER(NULL_ENTRY)
MESSAGE ...... WARNING(NULL_ENTRY)
```

最後の二つは S の全てのバージョンで使用可能である。ここで ..... は printf への引数の集まりであり、文字列や書式文字列の後にカンマで区切られた引数を伴う。

5.3 乱数生成

R の内部的な乱数生成の手順は

```
double unif_rand();
double norm_rand();
double exp_rand();
```

であり、一個の一様、正規、そして指数疑似乱数を与える。しかしながら、これらを使う前に、ユーザーは呼び出し

```
GetRNGstate();
```

を行い、そして全ての必要な変量が生成されたならば、次の呼び出しを実行する必要がある。

```
PutRNGstate();
```

これらは本質的に .Random.seed を読みだし (もしくは生成し)、そして使用後にそれを書き出している。

ファイル 'S.h' は S との互換性のため、GetRNGstate と PutRNGstate の代わりに、seed_in と seed_out を定義している。

乱数生成手順は R に固有なものである; R 関数の呼び出し評価を通じて以外、乱数生成手順を選択したり、乱数種を設定したりすることは出来ない。

R の rxxx 関数の背後にある C コードは、ヘッダファイル 'Rmath.h' を読み込むことによりアクセス出来る; See ⟨undefined⟩ [Distribution functions], page ⟨undefined⟩。これらの呼び出しは一つの変量を生成し、GetRNGstate と PutRNGstate への呼び出しと同時に使われるべきである。

5.4 欠損値と IEEE 特殊値

IEEE 754-互換算術を欠くプラットフォーム上で R をコンパイルすることが出来、ユーザーはそれが利用可能であることを仮定する必要はない。むしろ NA, Inf, -Inf (すべてのプラットフォーム上で存在する) そして NaN を検査する関数の一組が存在する。これらの関数は次のマクロを用いてアクセスできる:

ISNA(*x*)	R の NA に対してだけ真
ISNAN(*x*)	R の NA と IEEE の NaN に対し真
R_FINITE(*x*)	Inf, -Inf, NA, NaN に対し偽

そして関数 R_IsNaN は NaN に対しては真だが、NA に対しては偽である。isnan や finite の代わりにこれらを使おう; 特に後者はしばしば誤った結果を与える。

Inf や -Inf の検査を R_PosInf や R_NegInf への等号性を検査することで得ることが出来る。そして NA を NA_REAL として設定 (しかし検査はしない) することが出来る。

上の全ては 倍精度実数 変数に対してだけ当てはまる。整数変数に対してはマクロ NA_INTEGER によりアクセス出来る変数があり、非整数性を検査したり設定したり出来る。

これらの特殊値は暴走した通常の計算中に生じる可能性のある極端な値として表現されている可能性があり、従ってそれらが不注意で生じたのではないことを検査する必要があるかもしれない。

5.5 表示

R へコンパイルされた C ルーティンから表示するための最も有用な関数は Rprintf である。これは printf と全く同じように使えるが、R の出力 (それはファイルではなく GUI コンソールかもしれない) に書き出すことが保証されている。R に戻る前に ("\n" を含む) 完全な行を書き込むことが賢明である。

関数 REprintf は良く似ているがエラーストリーム (stderr) に書き出し、これは標準的な出力ストリームと同じかもしれないし異なるかもしれない。関数 Rvprintf と REvprintf は vprintf インタフェイスを使用する類似物である。

5.5.1 FORTRAN からの表示

理論的には FORTRAN の write と print 文を使うことが出来るが、しかし出力は C のそれとうまく混ざり合わないかもしれず、GUI インタフェイス上で見えなくなる可能性がある。使用をさけるのがベストである。

FORTRAN コードから上法を出力することを容易にするために三つのサブルーティンが用意されている。

```
subroutine dblepr(label, nchar, data, ndata)
subroutine realpr(label, nchar, data, ndata)
subroutine intpr (label, nchar, data, ndata)
```

ここで *label* か最大 255 文字の文字ラベルであり、*nchar* はその長さ (もし全てのラベルを使うのなら `-1` であっても良い) であり、*data* は適当な型 (それぞれ `double precision`, `real` そして `integer`) の少くとも長さ *ndata* の文字配列である。これらのルーティンはラベルを一行に表示し、そして *data* をそれらがあたかも R のベクトルであるかのように以降の行に出力する。 これらは *ndata* が零でも動作し、従ってラベルだけを表示することも出来る。

5.6 FORTRAN から C を呼ぶ、またその逆

FORTRAN が生成するシンボルに対する命名規則はプラットフォームにより異なる。プラットフォーム固有の差異をカバーすることを助けるために、使われるべき一連のマクロがある。

`F77_SUB(`*name*`)`
: FORTRANto から呼ばれる関数を C 中で定義する

`F77_NAME(`*name*`)`
: C 中で使用する前に FORTRAN ルーチンを宣言する

`F77_CALL(`*name*`)`
: C から FORTRAN ルーチンを呼び出す

`F77_COMDECL(`*name*`)`
: C 中で FORTRAN の common block を宣言する

`F77_COM(`*name*`)`
: C から FORTRAN の common block をアクセスする

ほとんどの現在のプラットフォーム上でこれらは同じであるが、過信しない方が賢明である。

5.7 数値解析サブルーチン

R はそれ自身で使用するため多数の数学関数を持つ。例えば数値的線形代数計算や特殊関数である。

ヘッダファイル '`R_ext/Linpack.h`' には R が含む LINPACK と EISPACK 線形代数関数の詳細がある。これらは FORTRAN サブルーチンへの呼び出しとして表現されており、またユーザーの FROTRAN コードからも使用可能である。API の公式な一部ではないが、このサブルーチンのセットは変更されることはありそうもない (しかし増補されるかもしれない)。

ヘッダファイル '`Rmath.h`' には他の多くの理用可能な関数の一覧があり、以下の副節で説明される。これらの多くは R 関数の背後にあるコードへの C インタフェイスであり、R 関数のドキュメントが更なる詳細を与えるであろう。

5.7.1 分布関数

標準的な統計分布に対する密度関数、累積分布関数そしてクオンタイル関数を計算するルーティンがエントリポイントとして理用可能である。

エントリポイントへの引数は正規分布に対するそれらの形式を踏襲する:

```
double dnorm(double x, double mu, double sigma, int give_log);
double pnorm(double x, double mu, double sigma, int lower_tail,
             int give_log);
double qnorm(double p, double mu, double sigma, int lower_tail,
             int log_p);
double rnorm(double mu, double sigma);
```

つまり、最初の引数は密度関数、累積分布関数そしてクオンタイル関数の位置を与え、以下分布のパラメータを与える。引数 *lower_tail* は普通 `TRUE` (もしくは `1`) であるべきであるが、もし分布の上側が必要か指定されるなら `FALSE` (もしくは `0`) であっても良い。

累積 (もしくは "integrated") *hazard* 関数 $H(t) = -\log(1 - F(t))$ は直接以下のように得られることを注意しよう

```
- pdist(t, ..., /*lower_tail = */ FALSE, /* give_log = */ TRUE)
```

もしくは単に (そしてより暗号的に) `-` p*dist*`(t, ..., 0, 1)` でも良い。

最後に *give_log* は結果を対数スケールで欲しければ零以外の値に取り、そして p が対数スケールで指定されていれば *log_p* は零以外の値に取る必要がある。

乱数生成ルーチン `rnorm` は一つの正規乱数を返す。乱数生成ルーチンの使用のための手順は See Section 5.3 [乱数], page 46 を見よ。

これらの引き数列は (名前と `rnorm` が n を持たないことを除き) 同じ名前の対応する R 関数と全く同じであり、従って R 関数のドキュメントが使用可能である。

参考として、以下の表は基本名 (例外を除き先頭に 'd', 'p', 'q' もしくは 'r' が付く) と分布固有の引数の完全な組を与える。

beta	`beta`	`a, b`
non-central beta	`nbeta`	`a, b, lambda`
binomial	`binom`	`n, p`
Cauchy	`cauchy`	`location, scale`
chi-squared	`chisq`	`df`
non-central chi-squared	`nchisq`	`df, lambda`
exponential	`exp`	`scale`
F	`f`	`n1, n2`
non-central F	`{p,q}nf`	`n1, n2, ncp`
gamma	`gamma`	`shape, scale`
geometric	`geom`	`p`
hypergeometric	`hyper`	`NR, NB, n`
logistic	`logis`	`location, scale`
lognormal	`lnorm`	`logmean, logsd`
negative binomial	`nbinom`	`n, p`
normal	`norm`	`mu, sigma`
Poisson	`pois`	`lambda`
Student's t	`t`	`n`
non-central t	`{p,q}nt`	`df, delta`
Studentized range	`{p,q}tukey`	`rr, cc, df`
uniform	`unif`	`a, b`
Weibull	`weibull`	`shape, scale`
Wilcoxon rank sum	`wilcox`	`m, n`
Wilcoxon signed rank	`signrank`	`n`

引数名は R のそれと全く同じではない。

5.7.2 数学関数

`double` **gammafn** (`double` x) Function
`double` **lgammafn** (`double` x) Function
`double` **digamma** (`double` x) Function
`double` **trigamma** (`double` x) Function
`double` **tetragamma** (`double` x) Function
`double` **pentagamma** (`double` x) Function
ガンマ関数、その自然対数、そしてその最初の 4 つの導関数。

`double` **beta** (`double` a, `double` b) Function
`double` **lbeta** (`double` a, `double` b) Function
(完全) ベータ関数とその自然対数。

`double` **choose** (`double` n, `double` k) Function
`double` **lchoose** (`double` n, `double` k) Function
n 個から k 個を選ぶ組合せの数とその自然対数。n と k は一番近い整数に丸められる。

`double` **bessel_i** (`double` x, `double` nu, `double` $expo$) Function
`double` **bessel_j** (`double` x, `double` nu) Function
`double` **bessel_k** (`double` x, `double` nu, `double` $expo$) Function
`double` **bessel_y** (`double` x, `double` nu) Function
インデックス nu でタイプが I, J, K そして Y のベッセル関数。`bessel_i` と `bessel_k` に対してはもし $expo$ `==2` `exp(-x) I(x;nu)` もしくは `exp(x) K(x;nu)` を返すオプションがある。(スケール化しない値には $expo$ `== 1` を使う。)

5.7.3 小道具

他にも利用可能な少しの数値計算用関数がエントリーポイントの形である。

`double` **R_pow** (`double` x, `double` y) Function
`double` **R_pow_di** (`double` x, `int` i) Function
`R_pow(`x`, `y`)` と `R_pow_di(`x`, `i`)` はそれぞれ x`^`y と x`^`i を計算する。x, y もしくは i が零だったり、欠損値、無限大、又は `NaN` でないかを `R_FINITE` を用いて検査し、x, y もしくは i が零だったり、欠損値、無限大、又は `NaN` の場合は適正な結果 (R と同じ) を返す。

`double` **pythag** (`double` a, `double` b) Function
`pythag(`a`, `b`)` は `sqrt(`a`^2 + `b`^2)` を桁溢れや破滅的な桁落ち無しに計算する: 例えば a と b の双方が `1e200` と `1e300` (IEEE 倍精度で) の間にあっても依然として動作する。

`double` **log1p** (x) Function
`log(1 + `x`)` (*log 1* **p***lus* x) を小さな x, つまり $|x| \ll 1$ でも正確に計算する。

`int` **imax2** (`int` *x*, `int` *y*) Function
`int` **imin2** (`int` *x*, `int` *y*) Function
`double` **fmax2** (`double` *x*, `double` *y*) Function
`double` **fmin2** (`double` *x*, `double` *y*) Function
それぞれ二つの整数、もしくは倍精度実数の大きな方 (`max`) もしくは小さな方 (`min`) を返す。

`double` **sign** (`double` *x*) Function
signum 関数を計算する。sign(x) は x が正、零、負に応じて 1, 0, 又は -1 となる。

`double` **fsign** (`double` *x*, `double` *y*) Function
"符号の交換" を実行する。$|x| * \text{sign}(y)$ と定義される。

`double` **fprec** (`double` *x*, `double` *digits*) Function
x を 10 進法で (少数点以下) *digits* 桁に丸めた数値を返す。
これは R の `round()` で用いられる関数である。

`double` **fround** (`double` *x*, `double` *digits*) Function
x の値を 10 進法で有向数字 *digits* 桁に丸めた数値を返す。
これは R の `signif()` で用いられる関数である。

`double` **fmod** (`double` *x*) Function
x の値の絶対値を返す。

`double` **ftrunc** (`double` *x*) Function
x の値を零に向かって (整数値に) 切り捨てた値を返す。

5.7.4 数学定数

R は普通 '`math.h`' に含まれる定数と、統計計算で使われる幾つかの定数を含む、良く使われる数学定数のセットを持つ。これら全ては (少くとも) 30 桁の精度で '`Rmath.h`' 中にある。以下の定義では `ln(x)` は自然対数 (R における `log(x)`) を表す。

Name	**Definition** (`ln` = `log`)	**round**(*value*, **7**)
`M_E`	= e	2.7182818
`M_LOG2E`	= log2(e)	1.4426950
`M_LOG10E`	= log10(e)	0.4342945
`M_LN2`	= ln(2)	0.6931472
`M_LN10`	= ln(10)	2.3025851
`M_PI`	= pi	3.1415927
`M_PI_2`	= pi/2	1.5707963
`M_PI_4`	= pi/4	0.7853982
`M_1_PI`	= 1/pi	0.3183099
`M_2_PI`	= 2/pi	0.6366198
`M_2_SQRTPI`	= 2/sqrt(pi)	1.1283792
`M_SQRT2`	= sqrt(2)	1.4142136
`M_SQRT1_2`	= 1/sqrt(2)	0.7071068

M_SQRT_3	= sqrt(3)	1.7320508
M_SQRT_32	= sqrt(32)	5.6568542
M_LOG10_2	= log10(2)	0.3010300
M_SQRT_PI	= sqrt(pi)	1.7724539
M_1_SQRT_2PI	= 1/sqrt(2pi)	0.3989423
M_SQRT_2dPI	= sqrt(2/pi)	0.7978846
M_LN_SQRT_PI	= ln(sqrt(pi))	0.5723649
M_LN_SQRT_2PI	= ln(sqrt(2*pi))	0.9189385
M_LN_SQRT_PId2	= ln(sqrt(pi/2))	0.2257914

インクルードされるヘッダー 'R_ext/Constants.h' 中で定義されている定数 (PI, DOUBLE_EPS) がある。後者は主に S との互換性のためである。

さらにインクルードされるヘッダー 'R_ext/Boolean.h' は C で論理値を首尾一貫して使用する方法を提供するために Rboolean 型の定数 TRUE と FALSE = 0 を持つ。

5.8 小道具関数

R はユーザーの C コードへ利用可能なかなり完全なソートのためのルーティンを持つ。以下がそうである。

void **R_isort** (int* *x*, int *n*) Function
void **R_rsort** (double* *x*, int *n*) Function
void **R_csort** (Rcomplex* *x*, int *n*) Function
void **rsort_with_index** (double* *x*, int* *index*, int *n*) Function
最初の三つはそれぞれ整数、実数 (倍精度) そして複素数をソートする。(複素数は最初に実部、それから虚部でソートされる。)

rsort_with_index は *x* をソートし、次に同じ並べ換えを *index* に行う。

void **revsort** (double* *x*, int* *index*, int *n*) Function
rsort_with_index に類似するが降順にソートする。

void **iPsort** (int* *x*, int *n*, int *k*) Function
void **rPsort** (double* *x*, int *n*, int *k*) Function
void **cPsort** (Rcomplex* *x*, int *n*, int *k*) Function
これら全ては (極めて) 部分的なソートを行う。*x* をソートし、*x*[*k*] がより小さな値は左に、より大きな値は右にくるようにする。

void **R_max_col** (double* *matrix*, int* *nr*, int* *nc*, int* *maxes*) Function
行 ("FORTRAN") 順の *nr* 掛ける *ny* の行列 matrix を与えたとき、R_max_col() は *maxes*[*i*-1] に *i*-番目の行の最大要素の列数を返す (R の max.col() 関数と同様)。

また幾つかの R 関数中でファイル名を展開する内部関数があり、path.expand を用いて直接呼び出せる。

char ***R_ExpandFileName** (char* *fn*) Function
パス名 *fn* の先頭のチルダをユーザーのホームディレクトリ (もし存在すれば) に置き換える。厳密な意味はプラットフォームに依存する; 普通もし定義されていれば環境変数 HOME から取られる。

5.9 プラットフォームとバージョン情報

ヘッダファイルは USING_R を定義し、これはコードが実際に R で使われているかどうかを検査するのに使える。ヘッダファイル 'Rconfig.h' ('R.h' により取り込まれる) は主に他のヘッダファイル中にあるプラットフォーム固有のマクロを定義するのに使われる。マクロ WORDS_BIGENDIAN は big-endian システムに対して定義され (例えば sparc-sun-solaris2.6)、little-endian システム (Linux や Windows 下の i686 のように) では定義されていない。これはバイナリファイルを操作する際に便利である。

ヘッダファイル 'Rversion.h' ('R.h' により取り込まれる) は整数としてコード化されたバージョン情報を与えるマクロ R_VERSION とコード化を行うマクロ R_Version を含む。これは R のバージョンが十分新しいか検査したり、古いバージョンとの互換的特徴を取り込むのに使うことが出来る。このマクロを持たない初期のバージョンに対する備えとして以下のような構成を使おう

```
#if defined(R_VERSION) && R_VERSION >= R_Version(0, 99, 0)
  ...
#endif
```

マクロ R_MAJOR, R_MINOR, R_YEAR, R_MONTH そして R_DAY に関する詳細な情報が利用可能である: それらの書式についてはヘッダファイル 'Rversion.h' を見よ。(99.0 のような) マイナーなバージョンは patchlevel を含むことを注意しよう。

5.10 これらの関数を自分自身の C コードで使う

'Rmath.h' に文章化されている R の数学関数のセットである Mathlib を Unix や Windows で使える単独使用可能なライブラリ 'libRmath' として構築できる。(これはこのヘッダファイルから取られた Section 5.7 [数値解析サブルーチン], page 48 に文章化されている関数を含む。)

このライブラリは R を移植する際には自動的に作成されず、ディレクトリ 'src/nmath/standalone' に作成することが出来る。そこにあるファイル 'README' を見よ。コードを自分の C プログラムで使うためには以下を取り込む

```
#define MATHLIB_STANDALONE
#include <Rmath.h>
```

そして '-lRmath' に対してリンクする。参考例ファイル 'test.c' がある。

乱数発生ルーティンを使うには少々注意がいる。一様乱数発生プログラムを準備する必要がある。

```
double unif_rand(void)
```

または用意されたものを使う (用意されたものを使うための共用ライブラリや DLL とともに、これは Marsaglia-multicarry 法でありその乱数種を設定するためのエントリポイント point

```
set_seed(unsigned int, unsigned int)
```

を持つ)。

Appendix A R の（内部の）プログラミングに関する雑多なこと

A.1 .Internal と .Primitive

構築時に R にコンパイルされた C コードは “直接” もしくは.Internal インタフェイス経由で呼び出すことが出来、これは構文を除けば.External インタフェイスに非常に良く似ている。取り詳しくのべると、R は R 関数の名前と対応する呼び出し C 関数の表を維持しており、それは慣例ですべて ‘do_’ ではじまり SEXP を返す。R_FunTab in ファイル ‘src/main/names.c’) one can この表 (ファイル ‘src/main/names.c’ 中の R_FunTab) 経由で、また関数の引数が幾つ必要か、許されるか、引数は呼び出し前に評価されるべきかどうか、関数は .Internal インタフェイス経由でアクセスされねばならないという意味で “内部的” かどうか、直接アクセスでき R 中で .Primitive として表示されるべきか、等を指定できる。

R の機能はまた対応する C コードを用意しこの関数表に加えることにより拡張することが出来る。

一般的に、そうした全ての関数は .Internal() を使用するのが安全であり、特に名前と既定引数の分かりやすい処理を可能にする。例えば axis は次のように定義される

```
axis <- function(side, at = NULL, labels = NULL, ...)
    .Internal(axis(side, at, labels, ...))
```

しかしながら、便宜と効率性 (.Internal インタフェイスの使用はいくらかの負荷を生じる) のために、直接アクセス出来る例外がある。これらの関数は R コードを使用せず、従って通常のインタプリタ関数と非常に異なる。特に、args と body はそうしたオブジェクトに対し NULL を返す。

これらの “primitive” な関数は以下のように完全に指定できる。

1. 真に 言語 の一部であるが R の “primitive” 関数として存在するもの。

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

2. 一部分特定、付値、算術・論理演算等の基本的 演算子 (つまり通常 foo(a, b, ...) として呼び出されないもの)。これらは以下の 1-, 2-, そして N-引数関数である:

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

3. 以下の関数グループのどれかに属する “低水準” の 0- または 1-引数関数:

   a. 単一引数の基本数学関数、つまり

   ```
   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
```

しかしながら R 関数 `log` は名前付きのオプション引数 `base` を持ち、従って次のように定義する

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

ことにより `log(x = pi, base = 2)` が `log(base = 2, x = pi)` に一致することが保証される。

b. 次のような“プログラミング”以外では滅多に使われない (つまり、主に他の関数の内部で使われる) 関数

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

(ここで xxx は約 30 個の異なった意味を持つ、例えば `function`, `vector`, `numeric`, 等)。

c. プログラミングとセッションの管理をする機能

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

4. 以下の基本的付値と抽出関数

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

5. 以下の僅かな N-引数関数は効率性のために“primitive”とされている。名前付き引数を適正に扱うためには注意がいる:

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

A.2 R コードの検査

(R 開発者として) R の base (R とともに配布される全てのパッケージ) に新しい関数を付け加えるときは、`make test-Specific` もしくは特に `cd tests; make no-segfault.Rout` が依然として動作 (対話型のユーザーの介入無しで、また単独で動く計算機上で) するかどうかを慎重に検査す

る必要がある。もし新しい関数がインタネットにアクセスしたり、または GUI 型対話を必要とするならば、その名前を ‘`tests/make-no-segfault.R`’ 中の “stop list” に付け加えて欲しい。

Appendix B R のコーディングの標準

R は Linux、ほとんどの Unix の変種、32-bit 版の Windows、そしてついには (再び) Power Mac といった多くのプラットフォーム上で動くことを目指している。従って、R の基本配布物への追加やアドオンパッケージを提供することにより R を拡張するときは、少数のプラットフォームだけに固有の特性に依存することは、もしそれが可能なら避けなければならない。特に、多くの R 開発者は GNU ツールを使っているが、標準的なツールへの GNU 拡張を利用すべきではない。幾つかの他のソフトウェアパッケージは公然と、例えば GNU make や GNU C++ コンパイラーを使っているが、R は使用しない。しかしながら R は GNU プロジェクトであり、GNU のコーディング標準は出来る限り守られるべきである。

以下のツールは R の拡張で “安全に仮定できる”。

- ANSI C コンパイラー。もし ANSI 標準が得られないときは Brian W. Kernighan & Dennis M. Ritchie, *The C Programming Language* の第二版を見よ。POSIX 等の如何なる拡張も、典型的には Autoconf (see Section 1.2 [コンフィギュアと後片付け], page 5) を用いて検査されるべきである。
- FORTRAN 77 コンパイラーもしくは FORTRAN-to-C 変換プログラムである `f2c`。
- バージョン 4.2 の BSD システムの `make` を基本とする単純な `make`。

  ‘`%`’ を用いたパターン規則、自動変数 ‘`$^`’、変数に値を追加する ‘`+=`’ 構文、エラー無しの make ファイルの (“safe”) 取り込み、条件付き実行、その他大勢を含む GNU もしくは他の拡張は使うべきではない (より詳細は GNU *Make Manual* の “Features” の章を見よ)。他方で、もし `make` が `VPATH` 機構をサポートしていれば、R を別個 (ソースを含まない) のディレクトリに構築することが出来るはずである。

  Windows 固有の makefile は GNU `make` 3.75 版もしくはそれ以降を仮定しても良い。なぜならそれ以外の `make` はこのプラットフォームでは永続しそうもないからである。
- Bourne シェルや、`grep`, `sed`, そして `awk` を含む “伝統的な” Unix プログラミングツール。

  こうしたツールには POSIX 標準があるが、これらは完全にはサポートされないかもしれず、正確な標準は典型的にアクセスしにくい。基本となる特性は Brian W. Kernighan & RobPike による *The UNIX Programming Environment* といった本から得られる。特に正規表現に置ける ‘`|`’ は拡張正規表現であり、全ての版の `grep` や `sed` でサポートされていない。

  Windows ではこうしたツールを仮定でき、必要なバージョン (特に、`basename`, `cat`, `comm`, `cp`, `cut`, `diff`, `echo`, `egrep`, `expr`, `find`, `gawk`, `grep`, `ls`, `mkdir`, `mv`, `rm`, `sed`, `sort`, `tar`, `touch`, `unzip`, `wc` そして `zip`) は `http://www.stats.ox.ac.uk/pub/Rtools/tools.zip` に提供されている。しかしながら、これは単独稼働のシェル (Bourne シェルはいうまでもなく) を利用していないので、リダイレクションは `system` 経由では利用可能と仮定できない。

加えて、以下のツールがある種の作業に必要になる。

- バージョン 5 の Perl が Rd 書式でかかれたドキュメントを平文、HTML, LaTeX に変換し、例を取り出すために必要になる。更に、`check` や `build` (see Section 1.3 [パッケージの検査と構築], page 6) といった他のツールは Perl を必要とする。

  R のコアチームは R をソースから構築したり、アドオンパッケージを構築し検査したり、アドオンパッケージをソースから移植するのに、Perl (第 5 版) が安全に仮定できることを決定した。他方で、アドオンパッケージの *binary* (構築済み) 版をインストールしたり、実行時には Perl は全く仮定することは出来ない。

- バージョン 4 の Makeinfo が GNU Texinfo システムで書かれた R のマニュアルの Info ファイルを作るのに必要である。(将来の R の配布は Info ファイルを含むであろう。)

またコードが他人にも理解可能な風に書かれていることが重要である。これは特に問題を解決し、それだけで理解可能な変数名を利用し、コードに注釈を施し、適正に書式化するのに有用である。R のコアチームは R と C (そして同様に Perl) コードに 4 文字分、Rd 書式には 2 文字分の基本インデントを施すことを推奨する。Emacs のユーザーは以下を初期化ファイルの一つに置くことによりこのインデントスタイルを移植できる。

```
;;; C
(add-hook 'c-mode-hook
          (lambda () (c-set-style "bsd")))
;;; ESS
(add-hook 'ess-mode-hook
          (lambda ()
            (ess-set-style 'C++)
            ;; Because
            ;;                                DEF GNU BSD K&R C++
            ;; ess-indent-level                 2   2   8   5   4
            ;; ess-continued-statement-offset   2   2   8   5   4
            ;; ess-brace-offset                 0   0  -8  -5  -4
            ;; ess-arg-function-offset          2   4   0   0   0
            ;; ess-expression-offset            4   2   8   5   4
            ;; ess-else-offset                  0   0   0   0   0
            ;; ess-close-brace-offset           0   0   0   0   0
            (add-hook 'local-write-file-hooks
                      (lambda ()
                        (nuke-trailing-whitespace)))))
;;; Perl
(add-hook 'perl-mode-hook
          (lambda () (setq perl-indent-level 4)))
```

(Emacs の C と R モードに対する 'GNU' スタイルは 2 文字の基本インデントを利用し、これは狭い幅のフォントを使ったとき構造を十分明瞭には表示しない様に決められた。)

関数と変数の索引

*

.

\

B

C

D

E

F

G

I

L

M

N

P

R

S

T

U

V

概念の索引