R FAQ

Frequently Asked Questions on R

Kurt Hornik
and the R Core Team

Table of Contents

1 Introduction 1

1.1 Legalese 1
1.2 Obtaining this document 1
1.3 Citing this document 1
1.4 Notation 1
1.5 Feedback 1

2 R Basics 2

2.1 What is R? 2
2.2 What machines does R run on? 3
2.3 What is the current version of R? 3
2.4 How can R be obtained? 3
2.5 How can R be installed? 3
2.5.1 How can R be installed (Unix-like) 4
2.5.2 How can R be installed (Windows) 5
2.5.3 How can R be installed (Mac) 5
2.6 Are there Unix-like binaries for R? 5
2.7 What documentation exists for R? 6
2.8 Citing R 7
2.9 What mailing lists exist for R? 7
2.10 What is CRAN? 8
2.11 Can I use R for commercial purposes? 9
2.12 Why is R named R? 9
2.13 What is the R Foundation? 9
2.14 What is R-Forge? 10

3 R and S 11

3.1 What is S? 11
3.2 What is S-PLUS? 11
3.3 What are the differences between R and S? 11
3.3.1 Lexical scoping 12
3.3.2 Models 15
3.3.3 Others 15
3.4 Is there anything R can do that S-PLUS cannot? 17
3.5 What is R-plus? 18

4 R Web Interfaces 19

5 R Add-On Packages 20

5.1 Which add-on packages exist for R? 20
5.1.1 Add-on packages in R 20

5.1.2 Add-on packages from CRAN 20
5.1.3 Add-on packages from Bioconductor 21
5.1.4 Other add-on packages 21
5.2 How can add-on packages be installed? 21
5.3 How can add-on packages be used? 22
5.4 How can add-on packages be removed? 23
5.5 How can I create an R package? 24
5.6 How can I contribute to R? 24

6 R and Emacs 25

6.1 Is there Emacs support for R? 25
6.2 Should I run R from within Emacs? 25
6.3 Debugging R from within Emacs 26

7 R Miscellanea 27

7.1 How can I set components of a list to NULL? 27
7.2 How can I save my workspace? 27
7.3 How can I clean up my workspace? 27
7.4 How can I get `eval()` and `D()` to work? 27
7.5 Why do my matrices lose dimensions? 28
7.6 How does autoloading work? 28
7.7 How should I set options? 28
7.8 How do file names work in Windows? 29
7.9 Why does plotting give a color allocation error? 29
7.10 How do I convert factors to numeric? 29
7.11 Are Trellis displays implemented in R? 29
7.12 What are the enclosing and parent environments? 30
7.13 How can I substitute into a plot label? 30
7.14 What are valid names? 31
7.15 Are GAMs implemented in R? 31
7.16 Why is the output not printed when I source() a file? 31
7.17 Why does outer() behave strangely with my function? 32
7.18 Why does the output from `anova()` depend on the order of factors in the model? 32
7.19 How do I produce PNG graphics in batch mode? 33
7.20 How can I get command line editing to work? 33
7.21 How can I turn a string into a variable? 33
7.22 Why do lattice/trellis graphics not work? 34
7.23 How can I sort the rows of a data frame? 34
7.24 Why does the help.start() search engine not work? 34
7.25 Why did my .Rprofile stop working when I updated R? 34
7.26 Where have all the methods gone? 35
7.27 How can I create rotated axis labels? 35
7.28 Why is read.table() so inefficient? 35
7.29 What is the difference between package and library? 36
7.30 I installed a package but the functions are not there 36

7.31 Why doesn't R think these numbers are equal? 36
7.32 How can I capture or ignore errors in a long simulation? 37
7.33 Why are powers of negative numbers wrong? 37
7.34 How can I save the result of each iteration in a loop into a separate file? 37
7.35 Why are p-values not displayed when using `lmer()`? 38
7.36 Why are there unwanted borders, lines or grid-like artifacts when viewing a plot saved to a PS or PDF file? 38
7.37 Why does backslash behave strangely inside strings? 38
7.38 How can I put error bars or confidence bands on my plot? 39
7.39 How do I create a plot with two y-axes? 40
7.40 How do I access the source code for a function? 40
7.41 Why does summary() report strange results for the R^2 estimate when I fit a linear model with no intercept? 40
7.42 Why is R apparently not releasing memory? 40
7.43 How can I enable secure https downloads in R? 41
7.44 How can I get CRAN package binaries for outdated versions of R? 41

8 R Programming 43

8.1 How should I write summary methods? 43
8.2 How can I debug dynamically loaded code? 43
8.3 How can I inspect R objects when debugging? 43
8.4 How can I change compilation flags? 43
8.5 How can I debug S4 methods? 43

9 R Bugs 44

9.1 What is a bug? 44
9.2 How to report a bug 44

Acknowledgments 47

1 Introduction

1.1 Legalese

Copyright © 1998–2020 Kurt Hornik
Copyright © 2021–2026 R Core Team

This document is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2, or (at your option) any later version.

This document is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

Copies of the GNU General Public License versions are available at

`https://www.R-project.org/Licenses/`

1.2 Obtaining this document

The latest version of this document is always available from

`https://CRAN.R-project.org/doc/manuals/`

From there, you can obtain versions converted to HTML and PDF.

1.3 Citing this document

In publications, please refer to this FAQ as Hornik and R Core Team (2026), "The R FAQ", and give the above, *official* URL:

```
@Misc{,
  author        = {Kurt Hornik and the R Core Team},
  title         = {{R} {FAQ}},
  year          = {2026},
  url           = {https://CRAN.R-project.org/doc/manuals/R-FAQ.html}
}
```

1.4 Notation

Everything should be pretty standard. '`R>`' is used for the R prompt, and a '`$`' for the shell prompt (where applicable).

1.5 Feedback

Feedback via email to `R-devel@R-project.org` is most welcome.

Features specific to the Windows and macOS ports of R are described in the "R for Windows FAQ" (`https://CRAN.R-project.org/bin/windows/base/rw-FAQ.html`) and the "R for Mac OS X FAQ" (`https://CRAN.R-project.org/bin/macosx/RMacOSX-FAQ.html`). If you have information on Mac or Windows systems that you think should be added to this document, please let us know.

2 R Basics

2.1 What is R?

R is a system for statistical computation and graphics. It consists of a language plus a run-time environment with graphics, a debugger, access to certain system functions, and the ability to run programs stored in script files.

The design of R has been heavily influenced by two existing languages: Becker, Chambers & Wilks' S (see Section 3.1 [What is S?], page 11) and Sussman's Scheme (`http://community.schemewiki.org/?scheme-faq`). Whereas the resulting language is very similar in appearance to S, the underlying implementation and semantics are derived from Scheme. See Section 3.3 [What are the differences between R and S?], page 11, for further details.

The core of R is an interpreted computer language which allows branching and looping as well as modular programming using functions. Most of the user-visible functions in R are written in R. It is possible for the user to interface to procedures written in the C, C++, or FORTRAN languages for efficiency. The R distribution contains functionality for a large number of statistical procedures. Among these are: linear and generalized linear models, nonlinear regression models, time series analysis, classical parametric and nonparametric tests, clustering and smoothing. There is also a large set of functions which provide a flexible graphical environment for creating various kinds of data presentations. Additional modules ("add-on packages") are available for a variety of specific purposes (see Chapter 5 [R Add-On Packages], page 20).

R was initially written by Ross Ihaka and Robert Gentleman at the Department of Statistics of the University of Auckland in Auckland, New Zealand. In addition, a large group of individuals has contributed to R by sending code and bug reports.

Since mid-1997 there has been a core group (the "R Core Team") who can modify the R source code archive, currently consisting of

John Chambers,
Peter Dalgaard,
Robert Gentleman,
Kurt Hornik,
Ross Ihaka,
Tomas Kalibera,
Michael Lawrence,
Uwe Ligges,
Thomas Lumley,
Martin Maechler,
Sebastian Meyer,
Paul Murrell,
Martyn Plummer,
Brian Ripley,
Deepayan Sarkar,
Duncan Temple Lang,
Luke Tierney,

Heather Turner, and
Simon Urbanek,

plus Heiner Schwarte up to October 1999, Guido Masarotto up to June 2003, Stefano Iacus up to July 2014, Seth Falcon up to August 2015, Duncan Murdoch up to September 2017, Martin Morgan up to June 2021, Douglas Bates up to March 2024, and Friedrich Leisch up to April 2024.

R has a home page at `https://www.R-project.org/`. It is free software (`https://www.gnu.org/philosophy/free-sw.html`) distributed under a GNU-style copyleft (`https://www.gnu.org/copyleft/copyleft.html`), and an official part of the GNU (`https://www.gnu.org/`) project ("GNU S").

2.2 What machines does R run on?

R is being developed for the Unix-like, Windows and Mac families of operating systems. Support for Mac OS Classic ended with R 1.7.1.

The current version of R will configure and build under a number of common Unix-like (e.g., `https://en.wikipedia.org/wiki/Unix-like`) platforms including *cpu*-linux-gnu for the i386, amd64/x86_64, alpha, arm, arm64, hppa, mips/mipsel, powerpc, s390x and sparc CPUs (e.g., `https://buildd.debian.org/build.php?&pkg=r-base`), 386-hurd-gnu, *cpu*-kfreebsd-gnu for i386 and amd64, i386-pc-solaris, rs6000-ibm-aix, sparc-sun-solaris, x86_64-apple-darwin, aarch64-apple-darwin, x86_64-unknown-freebsd and x86_64-unknown-openbsd.

If you know about other platforms, please drop us a note.

2.3 What is the current version of R?

R uses a 'major.minor.patchlevel' numbering scheme. Based on this, there are the current release version of R ('r-release') as well as two development versions of R, a patched version of the current release ('r-patched') and one working towards the next minor or eventually major ('r-devel') releases of R, respectively. New features are typically introduced in r-devel, while r-patched is for bug fixes mostly.

See `https://CRAN.R-project.org/sources.html` for the current versions of r-release, r-patched and r-devel.

2.4 How can R be obtained?

Sources, binaries and documentation for R can be obtained via CRAN, the "Comprehensive R Archive Network" (see Section 2.10 [What is CRAN?], page 8).

Sources are also available via `https://svn.R-project.org/R/`, the R Subversion repository, but currently not via anonymous rsync (nor CVS).

Tarballs with daily snapshots of the r-devel and r-patched development versions of R can be found at `https://cran.r-project.org/src/base-prerelease/`. An alternative source is `https://stat.ethz.ch/R/daily/`.

2.5 How can R be installed?

2.5.1 How can R be installed (Unix-like)

If R is already installed, it can be started by typing *R* at the shell prompt (of course, provided that the executable is in your path).

If binaries are available for your platform (see Section 2.6 [Are there Unix-like binaries for R?], page 5), you can use these, following the instructions that come with them.

Otherwise, you can compile and install R yourself, which can be done very easily under a number of common Unix-like platforms (see Section 2.2 [What machines does R run on?], page 3). The file `INSTALL` that comes with the R distribution contains a brief introduction, and the "R Installation and Administration" guide (see Section 2.7 [What documentation exists for R?], page 6) has full details.

Note that you need a FORTRAN 90 compiler as well as a C compiler to build R.

In the simplest case, untar the R source code, change to the directory thus created, and issue the following commands (at the shell prompt):

```
$ ./configure
$ make
```

If these commands execute successfully, the R binary and a shell script front-end called `R` are created and copied to the `bin` directory. You can copy the script to a place where users can invoke it, for example to `/usr/local/bin`. In addition, HTML versions of the R manuals (e.g., `R-exts.html`, the "Writing R Extensions" manual) are built in the `doc/manual` subdirectory (if a suitable `texi2any` program was found).

Use *make pdf* to build PDF (Portable Document Format) versions of the R manuals, including `fullrefman.pdf` (an R object reference index). Manuals written in the GNU Texinfo system can also be converted to `.info` files suitable for reading online with Emacs or standalone GNU Info; use *make info* to create these files.

Finally, use *make check* to find out whether your R system works correctly.

You can also perform a "system-wide" installation using *make install*. By default, this will install to the following directories:

`${prefix}/bin`
: the front-end shell script

`${prefix}/man/man1`
: the man page

`${prefix}/lib/R`
: all the rest (libraries, on-line help system, . . .). This is the "R Home Directory" (`R_HOME`) of the installed system.

In the above, `prefix` is determined during configuration (typically `/usr/local`) and can be set by running `configure` with the option

```
$ ./configure --prefix=/where/you/want/R/to/go
```

(E.g., the R executable will then be installed into `/where/you/want/R/to/go/bin`.)

To install info and PDF versions of the manuals, use *make install-info* and *make install-pdf*, respectively.

2.5.2 How can R be installed (Windows)

The `bin/windows` directory of a CRAN site contains binaries for a base distribution and add-on packages from CRAN to run on 64-bit versions of Windows 10 and later on x86_64 chips (R 4.1.3 was the last version of R to support 32-bit Windows). The Windows version of R was created by Robert Gentleman and Guido Masarotto; Brian Ripley and Duncan Murdoch made substantial contributions and it is now being maintained by other members of the R Core team.

The same directory has links to snapshots of the r-patched and r-devel versions of R.

See the "R for Windows FAQ" (`https://CRAN.R-project.org/bin/windows/base/rw-FAQ.html`) for more details.

2.5.3 How can R be installed (Mac)

The `bin/macosx` directory of a CRAN site contains a standard Apple installer package to run on macOS 10.13 ('High Sierra') or later, and another which runs only on 'Apple Silicon' Macs under macOS 11 ('Big Sur') or later. Once downloaded and executed, the installer will install the current release of R and R.app, the macOS GUI. This port of R for macOS is maintained by Simon Urbanek (and previously by Stefano Iacus). The "R for macOS FAQ" (`https://CRAN.R-project.org/bin/macosx/RMacOSX-FAQ.html`) has more details.

Snapshots of the r-patched and r-devel versions of R are available as Apple installer packages at `https://mac.R-project.org`.

2.6 Are there Unix-like binaries for R?

Binary distributions of R are available on many Unix-like OSes: only some can be mentioned here so check your OS's search facilities to see if one is available for yours.

The `bin/linux` directory of a CRAN site contains R packages for Debian and Ubuntu.

Debian packages, maintained by Dirk Eddelbuettel, have long been part of the Debian distribution, and can be accessed through APT, the Debian package maintenance tool. Use e.g. `apt-get install r-base r-recommended` to install the R environment and recommended packages. If you also want to build R packages from source, also run `apt-get install r-base-dev` to obtain the additional tools required for this. So-called "backports" of the current R packages for at least the *stable* distribution of Debian are provided by Johannes Ranke, and available from CRAN. See `https://CRAN.R-project.org/bin/linux/debian/index.html` for details on R Debian packages and installing the backports, which should also be suitable for other Debian derivatives. Native backports for Ubuntu are provided by Michael Rutter, see `https://CRAN.R-project.org/bin/linux/ubuntu/index.html` for instructions.

R binaries for Fedora, maintained by Tom "Spot" Callaway and Iñaki Ucar, are provided as part of the Fedora distribution and can be accessed through `dnf`, the RPM installer/updater. The Fedora R RPM is a "meta-package" which installs all the user and developer components of R (available separately as `R-core` and `R-core-devel`), as well as `R-java` and `R-java-devel`, which ensures that R is configured for use with Java. The R RPM also installs the standalone R math library (`libRmath` and `libRmath-devel`), although this is not necessary to use R. When a new version of R is released, there may be a delay of up to 2 weeks until the Fedora RPM becomes publicly available, as it must pass through the Fedora

update process. The Extra Packages for Enterprise Linux (EPEL) project (`https://docs.fedoraproject.org/en-US/epel/`) provides ports of the Fedora RPMs for RedHat Enterprise Linux and compatible distributions (e.g., CentOS Stream, Scientific Linux, Oracle Linux, AlmaLinux, or Rocky Linux among others). RPMs for selection of R packages are also provided by Fedora. Additional RPMs for R packages are maintained by Iñaki Ucar on Fedora Copr. See `https://CRAN.R-project.org/bin/linux/fedora/` for further details and installation instructions.

No other binary distributions are currently publicly available via CRAN.

2.7 What documentation exists for R?

Online documentation for most of the functions and variables in R exists, and can be printed on-screen by typing *`help(name)`* (or *`?name`*) at the R prompt, where *name* is the name of the topic help is sought for. (In the case of unary and binary operators and control-flow special forms, the name may need to be quoted.)

This documentation can also be made available as one reference manual for on-line reading in HTML and PDF formats, and as hardcopy via LaTeX, see Section 2.5 [How can R be installed?], page 3. An up-to-date HTML version is always available for web browsing at `https://stat.ethz.ch/R-manual/`.

The R distribution also comes with the following manuals.

- "An Introduction to R" (`R-intro`) includes information on data types, programming elements, statistical modeling and graphics. This document is based on the "Notes on S-Plus" by Bill Venables and David Smith.
- "Writing R Extensions" (`R-exts`) currently describes the process of creating R add-on packages, writing R documentation, R's system and foreign language interfaces, and the R API.
- "R Data Import/Export" (`R-data`) is a guide to importing and exporting data to and from R.
- "The R Language Definition" (`R-lang`), a first version of the "Kernighan & Ritchie of R", explains evaluation, parsing, object oriented programming, computing on the language, and so forth.
- "R Installation and Administration" (`R-admin`).
- "R Internals" (`R-ints`) is a guide to R's internal structures. (Added in R 2.4.0.)

An annotated bibliography (BibTeX format) of R-related publications can be found at

`https://www.R-project.org/doc/bib/R.bib`

Books on R by R Core Team members include

John M. Chambers (2008), "Software for Data Analysis: Programming with R". Springer, New York, ISBN 978-0-387-75935-7, `https://johnmchambers.su.domains/Rbook/`.

Peter Dalgaard (2008), "Introductory Statistics with R", 2nd edition. Springer, ISBN 978-0-387-79054-1, `https://link.springer.com/book/10.1007/978-0-387-79054-1`.

Robert Gentleman (2008), "R Programming for Bioinformatics". Chapman & Hall/CRC, Boca Raton, FL, ISBN 978-1-420-06367-7, `https://bioconductor.org/help/publications/books/r-programming-for-bioinformatics/`.

Stefano M. Iacus (2008), "Simulation and Inference for Stochastic Differential Equations: With R Examples". Springer, New York, ISBN 978-0-387-75838-1.

Deepayan Sarkar (2007), "Lattice: Multivariate Data Visualization with R". Springer, New York, ISBN 978-0-387-75968-5.

W. John Braun and Duncan J. Murdoch (2007), "A First Course in Statistical Programming with R". Cambridge University Press, Cambridge, ISBN 978-0521872652.

P. Murrell (2005), "R Graphics", Chapman & Hall/CRC, ISBN 1-584-88486-X, `https://www.stat.auckland.ac.nz/~paul/RGraphics/rgraphics.html`.

William N. Venables and Brian D. Ripley (2002), "Modern Applied Statistics with S" (4th edition). Springer, ISBN 0-387-95457-0, `https://www.stats.ox.ac.uk/pub/MASS4/`.

Jose C. Pinheiro and Douglas M. Bates (2000), "Mixed-Effects Models in S and S-Plus". Springer, ISBN 0-387-98957-0.

Last, but not least, Ross' and Robert's experience in designing and implementing R is described in Ihaka & Gentleman (1996), "R: A Language for Data Analysis and Graphics", *Journal of Computational and Graphical Statistics*, **5**, 299–314 (doi: 10.1080/10618600.1996.10474713 (`https://doi.org/10.1080/10618600.1996.10474713`)).

2.8 Citing R

To cite R in publications, use

```
@Manual{,
  title        = {R: A Language and Environment for Statistical
                  Computing},
  author       = {{R Core Team}},
  organization = {R Foundation for Statistical Computing},
  address      = {Vienna, Austria},
  year         = YEAR,
  doi          = {10.32614/R.manuals},
  url          = {https://www.R-project.org}
}
```

where *YEAR* is the release year of the version of R used and can determined as `R.version$year`.

Citation strings (or BibTEX entries) for R and R packages can also be obtained by `citation()`.

2.9 What mailing lists exist for R?

Thanks to Martin Maechler, there are several mailing lists devoted to R, including the following:

`R-announce`
: A moderated list for major announcements about the development of R and the availability of new code.

`R-packages`
: A moderated list for announcements on the availability of new or enhanced contributed packages.

`R-help`
: The 'main' R mailing list, for discussion about problems and solutions encountered using R, including using R packages in the standard R distribution and on CRAN; announcements (not covered by 'R-announce' or 'R-packages'); the availability of new functionality for R and documentation of R; and for posting nice examples and benchmarks.

`R-devel`
: This list is for questions and discussion about code development in R.

`R-package-devel`
: A list which provides a forum for those developing R packages.

Please read the posting guide (`https://www.R-project.org/posting-guide.html`) *before* sending anything to any mailing list.

Note in particular that R-help is intended to be comprehensible to people who want to use R to solve problems but who are not necessarily interested in or knowledgeable about programming. Questions likely to prompt discussion unintelligible to non-programmers (e.g., questions involving C or C++) should go to R-devel.

Convenient access to information on these lists, subscription, and archives is provided by the web interface at `https://stat.ethz.ch/mailman/listinfo/`. One can also subscribe (or unsubscribe) via email, e.g. to R-help by sending '`subscribe`' (or '`unsubscribe`') in the *body* of the message (not in the subject!) to `R-help-request@lists.R-project.org`.

Send email to `R-help@lists.R-project.org` to send a message to everyone on the R-help mailing list. Subscription and posting to the other lists is done analogously, with '`R-help`' replaced by '`R-announce`', '`R-packages`', and '`R-devel`', respectively. Note that the R-announce and R-packages lists are gatewayed into R-help. Hence, you should subscribe to either of them only in case you are not subscribed to R-help.

It is recommended that you send mail to R-help rather than only to the R Core developers (who are also subscribed to the list, of course). This may save them precious time they can use for constantly improving R, and will typically also result in much quicker feedback for yourself.

Of course, in the case of bug reports it would be very helpful to have code which reliably reproduces the problem. Also, make sure that you include information on the system and version of R being used. See Chapter 9 [R Bugs], page 44, for more details.

See `https://www.R-project.org/mail.html` for more information on the R mailing lists.

2.10 What is CRAN?

The "Comprehensive R Archive Network" (CRAN) is a collection of sites which carry identical material, consisting of the R distribution(s), the contributed extensions, documentation for R, and binaries.

The CRAN main site at WU (Wirtschaftsuniversität Wien) in Austria can be found at the URL

```
https://CRAN.R-project.org/
```

and is mirrored daily to many sites around the world. See `https://CRAN.R-project.org/mirrors.html` for a complete list of mirrors. Please use the CRAN site closest to you to reduce network load.

From CRAN, you can obtain the latest official release of R, daily snapshots of R (copies of the current source trees), as gzipped and bzipped tar files, a wealth of additional contributed code, as well as prebuilt binaries for various operating systems (Linux, Mac OS Classic, macOS, and MS Windows). CRAN also provides access to documentation on R, existing mailing lists and the R Bug Tracking system.

Since March 2016, "old" material is made available from a central CRAN archive server (`https://CRAN-archive.R-project.org/`).

Please always use the URL of the master site when referring to CRAN.

2.11 Can I use R for commercial purposes?

R is released under the GNU General Public License (GPL). If you have any questions regarding the legality of using R in any particular situation you should bring it up with your legal counsel. We are in no position to offer legal advice.

It is the opinion of the R Core Team that one can use R for commercial purposes (e.g., in business or in consulting). The GPL, like all Open Source licenses, permits all and any use of the package. It only restricts distribution of R or of other programs containing code from R. This is made clear in clause 6 ("No Discrimination Against Fields of Endeavor") of the Open Source Definition (`https://opensource.org/osd`):

> The license must not restrict anyone from making use of the program in a specific field of endeavor. For example, it may not restrict the program from being used in a business, or from being used for genetic research.

It is also explicitly stated in clause 0 of the GPL, which says in part

> Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program.

Most add-on packages, including all recommended ones, also explicitly allow commercial use in this way. A few packages are restricted to "non-commercial use"; you should contact the author to clarify whether these may be used or seek the advice of your legal counsel.

None of the discussion in this section constitutes legal advice. The R Core Team does not provide legal advice under any circumstances.

2.12 Why is R named R?

The name is partly based on the (first) names of the first two R authors (Robert Gentleman and Ross Ihaka), and partly a play on the name of the Bell Labs language 'S' (see Section 3.1 [What is S?], page 11).

2.13 What is the R Foundation?

The R Foundation is a not for profit organization working in the public interest. It was founded by the members of the R Core Team in order to provide support for the R project

and other innovations in statistical computing, provide a reference point for individuals, institutions or commercial enterprises that want to support or interact with the R development community, and to hold and administer the copyright of R software and documentation. See `https://www.R-project.org/foundation/` for more information.

2.14 What is R-Forge?

R-Forge (`https://R-Forge.R-project.org/`) offers a central platform for the development of R packages, R-related software and further projects. It is based on GForge (`https://en.wikipedia.org/wiki/GForge`) offering easy access to the best in SVN, daily built and checked packages, mailing lists, bug tracking, message boards/forums, site hosting, permanent file archival, full backups, and total web-based administration. For more information, see the R-Forge web page and Stefan Theußl and Achim Zeileis (2009), "Collaborative software development using R-Forge", *The R Journal*, **1**(1):9–14.

3 R and S

3.1 What is S?

S is a very high level language and an environment for data analysis and graphics. In 1998, the Association for Computing Machinery (ACM) presented its Software System Award to John M. Chambers, the principal designer of S, for

> the S system, which has forever altered the way people analyze, visualize, and manipulate data ...
>
> S is an elegant, widely accepted, and enduring software system, with conceptual integrity, thanks to the insight, taste, and effort of John Chambers.

The evolution of the S language is characterized by four books by John Chambers and coauthors, which are also the primary references for S.

- Richard A. Becker and John M. Chambers (1984), "S. An Interactive Environment for Data Analysis and Graphics," Monterey: Wadsworth and Brooks/Cole.

  This is also referred to as the "*Brown Book*", and of historical interest only.
- Richard A. Becker, John M. Chambers and Allan R. Wilks (1988), "The New S Language," London: Chapman & Hall.

  This book is often called the "*Blue Book*", and introduced what is now known as S version 2.
- John M. Chambers and Trevor J. Hastie (1992), "Statistical Models in S," London: Chapman & Hall.

  This is also called the "*White Book*", and introduced S version 3, which added structures to facilitate statistical modeling in S.
- John M. Chambers (1998), "Programming with Data," New York: Springer, ISBN 0-387-98503-4 (`https://johnmchambers.su.domains/Sbook/`).

  This "*Green Book*" describes version 4 of S, a major revision of S designed by John Chambers to improve its usefulness at every stage of the programming process.

See `https://johnmchambers.su.domains/papers/96.7.ps` for further information on the "Evolution of the S Language".

3.2 What is S-PLUS?

S-PLUS is a value-added version of S sold by TIBCO Software Inc (`https://www.tibco.com/`) as 'TIBCO Spotfire S+'. See `https://en.wikipedia.org/wiki/S-PLUS` for more information.

3.3 What are the differences between R and S?

We can regard S as a language with three current implementations or "engines", the "old S engine" (S version 3; S-PLUS 3.x and 4.x), the "new S engine" (S version 4; S-PLUS 5.x and above), and R. Given this understanding, asking for "the differences between R and S" really amounts to asking for the specifics of the R implementation of the S language, i.e., the difference between the R and S *engines*.

For the remainder of this section, "S" refers to the S engines and not the S language.

3.3.1 Lexical scoping

Contrary to other implementations of the S language, R has adopted an evaluation model in which nested function definitions are lexically scoped. This is analogous to the evaluation model in Scheme.

This difference becomes manifest when *free* variables occur in a function. Free variables are those which are neither formal parameters (occurring in the argument list of the function) nor local variables (created by assigning to them in the body of the function). In S, the values of free variables are determined by a set of global variables (similar to C, there is only local and global scope). In R, they are determined by the environment in which the function was created.

Consider the following function:

```
cube <- function(n) {
  sq <- function() n * n
  n * sq()
}
```

Under S, `sq()` does not "know" about the variable `n` unless it is defined globally:

```
S> cube(2)
Error in sq():  Object "n" not found
Dumped
S> n <- 3
S> cube(2)
[1] 18
```

In R, the "environment" created when `cube()` was invoked is also looked in:

```
R> cube(2)
[1] 8
```

As a more "interesting" real-world problem, suppose you want to write a function which returns the density function of the r-th order statistic from a sample of size n from a (continuous) distribution. For simplicity, we shall use both the distribution and density functions distribution as explicit arguments. (Example compiled from various postings by Luke Tierney.)

The S-PLUS documentation for `call()` basically suggests the following:

```
dorder <- function(n, r, pfun, dfun) {
  f <- function(x) NULL
  con <- round(exp(lgamma(n + 1) - lgamma(r) - lgamma(n - r + 1)))
  PF <- call(substitute(pfun), as.name("x"))
  DF <- call(substitute(dfun), as.name("x"))
  f[[length(f)]] <-
    call("*", con,
         call("*", call("^", PF, r - 1),
              call("*", call("^", call("-", 1, PF), n - r),
                   DF)))
  f
}
```

Rather tricky, isn't it? The code uses the fact that in S, functions are just lists of special mode with the function body as the last argument, and hence does not work in R (one could make the idea work, though).

A version which makes heavy use of `substitute()` and seems to work under both S and R is

```
dorder <- function(n, r, pfun, dfun) {
  con <- round(exp(lgamma(n + 1) - lgamma(r) - lgamma(n - r + 1)))
  eval(substitute(function(x) K * PF(x)^a * (1 - PF(x))^b * DF(x),
                  list(PF = substitute(pfun), DF = substitute(dfun),
                       a = r - 1, b = n - r, K = con)))
}
```

(the `eval()` is not needed in S).

However, in R there is a much easier solution:

```
dorder <- function(n, r, pfun, dfun) {
  con <- round(exp(lgamma(n + 1) - lgamma(r) - lgamma(n - r + 1)))
  function(x) {
    con * pfun(x)^(r - 1) * (1 - pfun(x))^(n - r) * dfun(x)
  }
}
```

This seems to be the "natural" implementation, and it works because the free variables in the returned function can be looked up in the defining environment (this is lexical scope).

Note that what you really need is the function *closure*, i.e., the body along with all variable bindings needed for evaluating it. Since in the above version, the free variables in the value function are not modified, you can actually use it in S as well if you abstract out the closure operation into a function `MC()` (for "make closure"):

```
dorder <- function(n, r, pfun, dfun) {
  con <- round(exp(lgamma(n + 1) - lgamma(r) - lgamma(n - r + 1)))
  MC(function(x) {
       con * pfun(x)^(r - 1) * (1 - pfun(x))^(n - r) * dfun(x)
     },
     list(con = con, pfun = pfun, dfun = dfun, r = r, n = n))
}
```

Given the appropriate definitions of the closure operator, this works in both R and S, and is much "cleaner" than a substitute/eval solution (or one which overrules the default scoping rules by using explicit access to evaluation frames, as is of course possible in both R and S).

For R, `MC()` simply is

```
MC <- function(f, env) f
```

(lexical scope!), a version for S is

```
MC <- function(f, env = NULL) {
  env <- as.list(env)
  if (mode(f) != "function")
    stop(paste("not a function:", f))
  if (length(env) > 0 && any(names(env) == ""))
    stop(paste("not all arguments are named:", env))
  fargs <- if(length(f) > 1) f[1:(length(f) - 1)] else NULL
  fargs <- c(fargs, env)
  if (any(duplicated(names(fargs))))
    stop(paste("duplicated arguments:", paste(names(fargs)),
         collapse = ", "))
  fbody <- f[length(f)]
  cf <- c(fargs, fbody)
  mode(cf) <- "function"
  return(cf)
}
```

Similarly, most optimization (or zero-finding) routines need some arguments to be optimized over and have other parameters that depend on the data but are fixed with respect to optimization. With R scoping rules, this is a trivial problem; simply make up the function with the required definitions in the same environment and scoping takes care of it. With S, one solution is to add an extra parameter to the function and to the optimizer to pass in these extras, which however can only work if the optimizer supports this.

Nested lexically scoped functions allow using function closures and maintaining local state. A simple example (taken from Abelson and Sussman) is obtained by typing `demo("scoping")` at the R prompt. Further information is provided in the standard R reference "R: A Language for Data Analysis and Graphics" (see Section 2.7 [What documentation exists for R?], page 6) and in Robert Gentleman and Ross Ihaka (2000), "Lexical Scope and Statistical Computing", *Journal of Computational and Graphical Statistics*, **9**, 491–508 (doi: 10.1080/10618600.2000.10474895 (`https://doi.org/10.1080/10618600.2000.10474895`)).

Nested lexically scoped functions also imply a further major difference. Whereas S stores all objects as separate files in a directory somewhere (usually `.Data` under the current directory), R does not. All objects in R are stored internally. When R is started up it grabs a piece of memory and uses it to store the objects. R performs its own memory management of this piece of memory, growing and shrinking its size as needed. Having everything in memory is necessary because it is not really possible to externally maintain all relevant "environments" of symbol/value pairs. This difference also seems to make R *faster* than S.

The down side is that if R crashes you will lose all the work for the current session. Saving and restoring the memory "images" (the functions and data stored in R's internal memory at any time) can be a bit slow, especially if they are big. In S this does not happen, because everything is saved in disk files and if you crash nothing is likely to happen to them. (In fact, one might conjecture that the S developers felt that the price of changing their approach to persistent storage just to accommodate lexical scope was far too expensive.) Hence, when doing important work, you might consider saving often (see Section 7.2 [How can I save my workspace?], page 27) to safeguard against possible crashes. Other possibilities are logging

your sessions, or have your R commands stored in text files which can be read in using `source()`.

> **Note:** If you run R from within Emacs (see Chapter 6 [R and Emacs], page 25), you can save the contents of the interaction buffer to a file and conveniently manipulate it using `ess-transcript-mode`, as well as save source copies of all functions and data used.

3.3.2 Models

There are some differences in the modeling code, such as

- Whereas in S, you would use `lm(y ~ x^3)` to regress `y` on `x^3`, in R, you have to insulate powers of numeric vectors (using `I()`), i.e., you have to use `lm(y ~ I(x^3))`.
- The GLM family objects are implemented differently in R and S. The same functionality is available but the components have different names.
- Option `na.action` is set to `"na.omit"` by default in R, but not set in S.
- Terms objects are stored differently. In S a terms object is an expression with attributes, in R it is a formula with attributes. The attributes have the same names but are mostly stored differently.
- Finally, in R `y ~ x + 0` is an alternative to `y ~ x - 1` for specifying a model with no intercept. Models with no parameters at all can be specified by `y ~ 0`.

3.3.3 Others

Apart from lexical scoping and its implications, R follows the S language definition in the Blue and White Books as much as possible, and hence really is an "implementation" of S. There are some intentional differences where the behavior of S is considered "not clean". In general, the rationale is that R should help you detect programming errors, while at the same time being as compatible as possible with S.

Some known differences are the following.

- In R, if `x` is a list, then `x[i] <- NULL` and `x[[i]] <- NULL` remove the specified elements from `x`. The first of these is incompatible with S, where it is a no-op. (Note that you can set elements to `NULL` using `x[i] <- list(NULL)`.)
- In S, the functions named `.First` and `.Last` in the `.Data` directory can be used for customizing, as they are executed at the very beginning and end of a session, respectively.

  In R, the startup mechanism is as follows. Unless `--no-environ` was given on the command line, R searches for site and user files to process for setting environment variables. Then, R searches for a site-wide startup profile unless the command line option `--no-site-file` was given. This code is loaded in package **base**. Then, unless `--no-init-file` was given, R searches for a user profile file, and sources it into the user workspace. It then loads a saved image of the user workspace from `.RData` in case there is one (unless `--no-restore-data` or `--no-restore` were specified). Next, a function `.First()` is run if found on the search path. Finally, function `.First.sys` in the **base** package is run. When terminating an R session, by default a function `.Last` is run if found on the search path, followed by `.Last.sys`. If needed, the functions `.First()` and `.Last()` should be defined in the appropriate startup profiles. See the help pages for `.First` and `.Last` for more details.

- In R, `T` and `F` are just variables being set to `TRUE` and `FALSE`, respectively, but are not reserved words as in S and hence can be overwritten by the user. (This helps e.g. when you have factors with levels `"T"` or `"F"`.) Hence, when writing code you should always use `TRUE` and `FALSE`.
- In R, `dyn.load()` can only load *shared objects*, as created for example by *`R CMD SHLIB`*.
- In R, `attach()` currently only works for lists and data frames, but not for directories. (In fact, `attach()` also works for R data files created with `save()`, which is analogous to attaching directories in S.) Also, you cannot attach at position 1.
- Categories do not exist in R, and never will as they are deprecated now in S. Use factors instead.
- In R, `For()` loops are not necessary and hence not supported.
- In R, `assign()` uses the argument `envir=` rather than `where=` as in S.
- The random number generators are different, and the seeds have different length.
- R passes integer objects to C as `int *` rather than `long *` as in S.
- R has no single precision storage mode. However, as of version 0.65.1, there is a single precision interface to C/FORTRAN subroutines.
- By default, `ls()` returns the names of the objects in the current (under R) and global (under S) environment, respectively. For example, given

  ```
  x <- 1; fun <- function() {y <- 1; ls()}
  ```

  then `fun()` returns `"y"` in R and `"x"` (together with the rest of the global environment) in S.
- R allows for zero-extent matrices (and arrays, i.e., some elements of the `dim` attribute vector can be 0). This has been determined a useful feature as it helps reducing the need for special-case tests for empty subsets. For example, if `x` is a matrix, `x[, FALSE]` is not `NULL` but a "matrix" with 0 columns. Hence, such objects need to be tested for by checking whether their `length()` is zero (which works in both R and S), and not using `is.null()`.
- Named vectors are considered vectors in R but not in S (e.g., `is.vector(c(a = 1:3))` returns `FALSE` in S and `TRUE` in R).
- Data frames are not considered as matrices in R (i.e., if `DF` is a data frame, then `is.matrix(DF)` returns `FALSE` in R and `TRUE` in S).
- R by default uses treatment contrasts in the unordered case, whereas S uses the Helmert ones. This is a deliberate difference reflecting the opinion that treatment contrasts are more natural.
- In R, the argument of a replacement function which corresponds to the right hand side must be named '`value`'. E.g., `f(a) <- b` is evaluated as `a <- "f<-"(a, value = b)`. S always takes the last argument, irrespective of its name.
- In S, `substitute()` searches for names for substitution in the given expression in three places: the actual and the default arguments of the matching call, and the local frame (in that order). R looks in the local frame only, with the special rule to use a "promise" if a variable is not evaluated. Since the local frame is initialized with the actual arguments or the default expressions, this is usually equivalent to S, until assignment takes place.

- In S, the index variable in a `for()` loop is local to the inside of the loop. In R it is local to the environment where the `for()` statement is executed.
- In S, `tapply(simplify=TRUE)` returns a vector where R returns a one-dimensional array (which can have named dimnames).
- In S(-PLUS) the C locale is used, whereas in R the current operating system locale is used for determining which characters are alphanumeric and how they are sorted. This affects the set of valid names for R objects (for example accented chars may be allowed in R) and ordering in sorts and comparisons (such as whether `"aA" < "Bb"` is true or false). From version 1.2.0 the locale can be (re-)set in R by the `Sys.setlocale()` function.
- In S, `missing(`*arg*`)` remains `TRUE` if *arg* is subsequently modified; in R it doesn't.
- From R version 1.3.0, `data.frame` strips `I()` when creating (column) names.
- In R, the string `"NA"` is not treated as a missing value in a character variable. Use `as.character(NA)` to create a missing character value.
- R disallows repeated formal arguments in function calls.
- In S, `dump()`, `dput()` and `deparse()` are essentially different interfaces to the same code. In R from version 2.0.0, this is only true if the same `control` argument is used, but by default it is not. By default `dump()` tries to write code that will evaluate to reproduce the object, whereas `dput()` and `deparse()` default to options for producing deparsed code that is readable.
- In R, indexing a vector, matrix, array or data frame with `[` using a character vector index looks only for exact matches (whereas `[[` and `$` allow partial matches). In S, `[` allows partial matches.
- S has a two-argument version of `atan` and no `atan2`. A call in S such as `atan(x1, x2)` is equivalent to R's `atan2(x1, x2)`. However, beware of named arguments since S's `atan(x = a, y = b)` is equivalent to R's `atan2(y = a, x = b)` with the meanings of `x` and `y` interchanged. (R used to have undocumented support for a two-argument `atan` with positional arguments, but this has been withdrawn to avoid further confusion.)
- Numeric constants with no fractional and exponent (i.e., only integer) part are taken as integer in S-PLUS 6.x or later, but as double in R.

There are also differences which are not intentional, and result from missing or incorrect code in R. The developers would appreciate hearing about any deficiencies you may find (in a written report fully documenting the difference as you see it). Of course, it would be useful if you were to implement the change yourself and make sure it works.

3.4 Is there anything R can do that S-PLUS cannot?

Since almost anything you can do in R has source code that you could port to S-PLUS with little effort there will never be much you can do in R that you couldn't do in S-PLUS if you wanted to. (Note that using lexical scoping may simplify matters considerably, though.)

R offers several graphics features that S-PLUS does not, such as finer handling of line types, more convenient color handling (via palettes), gamma correction for color, and, most importantly, mathematical annotation in plot texts, via input expressions reminiscent of TEX constructs. See the help page for `plotmath`, which features an impressive on-line example.

More details can be found in Paul Murrell and Ross Ihaka (2000), "An Approach to Providing Mathematical Annotation in Plots", *Journal of Computational and Graphical Statistics*, **9**, 582–599 (doi: 10.1080/10618600.2000.10474900 (`https://doi.org/10.1080/10618600.2000.10474900`)).

3.5 What is R-plus?

See `https://en.wikipedia.org/wiki/R_programming_language#Commercialized_versions_of_R` for pointers to commercialized versions of R.

4 R Web Interfaces

Please refer to the CRAN task view on "Web Technologies and Services" (`https://CRAN.R-project.org/view=WebTechnologies`), specifically section "Web and Server Frameworks", for up-to-date information on R web interface packages.

Early references on R web interfaces include Jeff Banfield (1999), "Rweb: Web-based Statistical Analysis" (doi: 10.18637/jss.v004.i01 (`https://doi.org/10.18637/jss.v004.i01`)), David Firth (2003), "CGIwithR: Facilities for processing web forms using R" (doi: 10.18637/jss.v008.i10 (`https://doi.org/10.18637/jss.v008.i10`)), and Angelo Mineo and Alfredo Pontillo (2006), "Using R via PHP for Teaching Purposes: R-php" (doi: 10.18637/jss.v017.i04 (`https://doi.org/10.18637/jss.v017.i04`)).

5 R Add-On Packages

5.1 Which add-on packages exist for R?

5.1.1 Add-on packages in R

The R distribution comes with the following packages:

base Base R functions (and datasets before R 2.0.0).

compiler R byte code compiler (added in R 2.13.0).

datasets Base R datasets (added in R 2.0.0).

grDevices Graphics devices for base and grid graphics (added in R 2.0.0).

graphics R functions for base graphics.

grid A rewrite of the graphics layout capabilities, plus some support for interaction.

methods Formally defined methods and classes for R objects, plus other programming tools, as described in the Green Book.

parallel Support for parallel computation, including by forking and by sockets, and random-number generation (added in R 2.14.0).

splines Regression spline functions and classes.

stats R statistical functions.

stats4 Statistical functions using S4 classes.

tcltk Interface and language bindings to Tcl/Tk GUI elements.

tools Tools for package development and administration.

utils R utility functions.

5.1.2 Add-on packages from CRAN

The CRAN `src/contrib` area contains a wealth of add-on packages, including the following *recommended* packages which are to be included in all binary distributions of R.

KernSmooth
Functions for kernel smoothing (and density estimation) corresponding to the book "Kernel Smoothing" by M. P. Wand and M. C. Jones, 1995.

MASS Functions and datasets from the main package of Venables and Ripley, "Modern Applied Statistics with S".

Matrix Support for spares and dense matrices

boot Functions and datasets for bootstrapping from the book "Bootstrap Methods and Their Applications" by A. C. Davison and D. V. Hinkley, 1997, Cambridge University Press.

class Functions for classification (k-nearest neighbor and LVQ).

cluster Functions for cluster analysis.

codetools Code analysis tools.

foreign Functions for reading and writing data stored by statistical software like Minitab, S, SAS, SPSS, Stata, Systat, etc.

lattice Lattice graphics, an implementation of Trellis Graphics functions.

mgcv Routines for GAMs and other generalized ridge regression problems with multiple smoothing parameter selection by GCV or UBRE.

nlme Fit and compare Gaussian linear and nonlinear mixed-effects models.

nnet Software for single hidden layer perceptrons ("feed-forward neural networks"), and for multinomial log-linear models.

rpart Recursive PARTitioning and regression trees.

spatial Functions for kriging and point pattern analysis from "Modern Applied Statistics with S" by W. Venables and B. Ripley.

survival Functions for survival analysis, including penalized likelihood.

See the CRAN contributed packages page for more information.

Many of these packages are categorized into CRAN Task Views (`https://CRAN.R-project.org/web/views/`), allowing to browse packages by topic and providing tools to automatically install all packages for special areas of interest.

5.1.3 Add-on packages from Bioconductor

Bioconductor (`https://www.bioconductor.org/`) is an open source and open development software project for the analysis and comprehension of genomic data. Most Bioconductor components are distributed as R add-on packages. Initially most of the Bioconductor software packages (`https://bioconductor.org/packages/release/BiocViews.html#___Software`) focused primarily on DNA microarray data analysis. As the project has matured, the functional scope of the software packages broadened to include the analysis of all types of genomic data, such as SAGE, sequence, or SNP data. In addition, there are metadata (annotation, CDF and probe) and experiment data packages. See `https://bioconductor.org/install/` for available packages and a complete taxonomy via BioC Views.

5.1.4 Other add-on packages

Many more packages are available from places other than the default repositories discussed above (CRAN and Bioconductor). In particular, R-Forge provides a CRAN style repository at `https://R-Forge.R-project.org/`.

More code has been posted to the R-help mailing list, and can be obtained from the mailing list archive.

5.2 How can add-on packages be installed?

(Unix-like only.) The add-on packages on CRAN come as gzipped tar files named *`pkg_version`*`.tar.gz`. Let *path* be the path to such a package file. Provided that `tar` and `gzip` are available on your system, type

```
$ R CMD INSTALL path/pkg_version.tar.gz
```

at the shell prompt to install to the library tree rooted at the first directory in your library search path (see the help page for `.libPaths()` for details on how the search path is determined).

To install to another tree (e.g., your private one), use

```
$ R CMD INSTALL -l lib path/pkg_version.tar.gz
```

where *lib* gives the path to the library tree to install to.

Even more conveniently, you can install and automatically update packages from within R if you have access to repositories such as CRAN. See the help page for `available.packages()` for more information.

5.3 How can add-on packages be used?

To find out which additional packages are available on your system, type

```
library()
```

at the R prompt.

This produces something like

```
Packages in library '/home/me/lib/R':

mystuff       My own R functions, nicely packaged but not documented

Packages in library '/usr/local/lib/R/library':

KernSmooth    Functions for Kernel Smoothing Supporting Wand & Jones (1995)
MASS          Support Functions and Datasets for Venables and Ripley's MASS
Matrix        Sparse and Dense Matrix Classes and Methods
base          The R Base Package
boot          Bootstrap Functions (Originally by Angelo Canty for S)
class         Functions for Classification
cluster       "Finding Groups in Data": Cluster Analysis Extended
              Rousseeuw et al.
codetools     Code Analysis Tools for R
compiler      The R Compiler Package
datasets      The R Datasets Package
foreign       Read Data Stored by 'Minitab', 'S', 'SAS', 'SPSS', 'Stata',
              'Systat', 'Weka', 'dBase', ...
grDevices     The R Graphics Devices and Support for Colours and Fonts
graphics      The R Graphics Package
grid          The Grid Graphics Package
lattice       Trellis Graphics for R
methods       Formal Methods and Classes
mgcv          Mixed GAM Computation Vehicle with Automatic Smoothness
              Estimation
nlme          Linear and Nonlinear Mixed Effects Models
nnet          Feed-Forward Neural Networks and Multinomial Log-Linear
              Models
parallel      Support for Parallel Computation in R
rpart         Recursive Partitioning and Regression Trees
spatial       Functions for Kriging and Point Pattern Analysis
splines       Regression Spline Functions and Classes
stats         The R Stats Package
stats4        Statistical Functions using S4 Classes
survival      Survival Analysis
tcltk         Tcl/Tk Interface
tools         Tools for Package Development
utils         The R Utils Package
```

You can "load" the installed package *pkg* by

```
library(pkg)
```

You can then find out which functions it provides by typing one of

```
library(help = pkg)
help(package = pkg)
```

You can remove the loaded package *pkg* from the `search()` path by

```
detach("package:pkg")
```

(which does not by default unload the namespace, see `?detach`).

5.4 How can add-on packages be removed?

Use

```
$ R CMD REMOVE pkg_1 ... pkg_n
```

to remove the packages *pkg_1*, ..., *pkg_n* from the library tree rooted at the first directory given in `R_LIBS` if this is set and non-null, and from the default library otherwise.

To remove from library *lib*, do

```
$ R CMD REMOVE -l lib pkg_1 ... pkg_n
```

5.5 How can I create an R package?

A package consists of a subdirectory containing a file `DESCRIPTION` and the subdirectories `R`, `data`, `demo`, `exec`, `inst`, `man`, `po`, `src`, and `tests` (some of which can be missing). The package subdirectory may also contain files `INDEX`, `NAMESPACE`, `configure`, `cleanup`, `LICENSE`, `LICENCE`, `COPYING` and `NEWS`.

See Section "Creating R packages" in *Writing R Extensions* for details. This manual is included in the R distribution, see Section 2.7 [What documentation exists for R?], page 6, and gives information on package structure, the configure and cleanup mechanisms, and on automated package checking and building.

R version 1.3.0 has added the function `package.skeleton()` which will set up directories, save data and code, and create skeleton help files for a set of R functions and datasets.

See Section 2.10 [What is CRAN?], page 8, for information on uploading a package to CRAN.

5.6 How can I contribute to R?

R is in active development and there is always a risk of bugs creeping in. Also, the developers do not have access to all possible machines capable of running R. So, simply using it and communicating problems is certainly of great value.

The R Developer Page (`https://developer.R-project.org/`) acts as an intermediate repository for more or less finalized ideas and plans for the R statistical system. It contains (pointers to) TODO lists, RFCs, various other writeups, ideas lists, and SVN miscellanea.

6 R and Emacs

6.1 Is there Emacs support for R?

There is an Emacs package called ESS ("Emacs Speaks Statistics") which provides a standard interface between statistical programs and statistical processes. It is intended to provide assistance for interactive statistical programming and data analysis. Languages supported include: S dialects (R, S 3/4, and S-PLUS 3.x/4.x/5.x/6.x/7.x), LispStat dialects (XLispStat, ViSta), SAS, Stata, and BUGS.

ESS grew out of the need for bug fixes and extensions to S-mode 4.8 (which was a GNU Emacs interface to S/S-PLUS version 3 only). The current set of developers desired support for XEmacs, R, S4, and MS Windows. In addition, with new modes being developed for R, Stata, and SAS, it was felt that a unifying interface and framework for the user interface would benefit both the user and the developer, by helping both groups conform to standard Emacs usage. The end result is an increase in efficiency for statistical programming and data analysis, over the usual tools.

R support contains code for editing R source code (syntactic indentation and highlighting of source code, partial evaluations of code, loading and error-checking of code, and source code revision maintenance) and documentation (syntactic indentation and highlighting of source code, sending examples to running ESS process, and previewing), interacting with an inferior R process from within Emacs (command-line editing, searchable command history, command-line completion of R object and file names, quick access to object and search lists, transcript recording, and an interface to the help system), and transcript manipulation (recording and saving transcript files, manipulating and editing saved transcripts, and re-evaluating commands from transcript files).

The latest stable version of ESS is available via CRAN or the ESS web page (`https://ESS.R-project.org/`).

ESS comes with detailed installation instructions.

For help with ESS, send email to `ESS-help@r-project.org`.

Please send bug reports and suggestions on ESS to `ESS-bugs@r-project.org`. The easiest way to do this from is within Emacs by typing *`M-x ess-submit-bug-report`* or using the [ESS] or [iESS] pulldown menus.

6.2 Should I run R from within Emacs?

Yes, instead of just running it in a console, *definitely*. As an alternative to other IDEs such as RStudio, *possibly*, notably if you are interested to use Emacs for other computer interaction. You'd be using ESS, Emacs Speaks Statistics, see previous FAQ.

Inferior R mode provides a readline/history mechanism, object name completion, and syntax-based highlighting of the interaction buffer using Font Lock mode, as well as a very convenient interface to the R help system.

Of course, it also integrates nicely with the mechanisms for editing R source using Emacs. One can write code in one Emacs buffer and send whole or parts of it for execution to R; this is helpful for both data analysis and programming. One can also seamlessly integrate

with a revision control system, in order to maintain a log of changes in your programs and data, as well as to allow for the retrieval of past versions of the code.

In addition, it allows you to keep a record of your session, which can also be used for error recovery through the use of the transcript mode.

To specify command line arguments for the inferior R process, use *C-u M-x R* for starting R.

6.3 Debugging R from within Emacs

To debug R "from within Emacs", there are several possibilities. To use the Emacs GUD (Grand Unified Debugger) library with the recommended debugger GDB, type *M-x gdb* and give the path to the R *binary* as argument. At the `gdb` prompt, set `R_HOME` and other environment variables as needed (using e.g. *set env R_HOME /path/to/R/*, but see also below), and start the binary with the desired arguments (e.g., *run --quiet*).

If you have ESS, you can do *C-u M-x R* `RET - d SPC g d b RET` to start an inferior R process with arguments `-d gdb`.

A third option is to start an inferior R process via ESS (*M-x R*) and then start GUD (*M-x gdb*) giving the R binary (using its full path name) as the program to debug. Use the program `ps` to find the process number of the currently running R process then use the `attach` command in GDB to attach it to that process. One advantage of this method is that you have separate `*R*` and `*gud-gdb*` windows. Within the `*R*` window you have all the ESS facilities, such as object-name completion, that we know and love.

When using GUD mode for debugging from within Emacs, you may find it most convenient to use the directory with your code in it as the current working directory and then make a symbolic link from that directory to the R binary. That way `.gdbinit` can stay in the directory with the code and be used to set up the environment and the search paths for the source, e.g. as follows:

```
set env R_HOME /opt/R
set env R_PAPERSIZE letter
set env R_PRINTCMD lpr
dir /opt/R/src/appl
dir /opt/R/src/main
dir /opt/R/src/nmath
dir /opt/R/src/unix
```

7 R Miscellanea

7.1 How can I set components of a list to NULL?

You can use

```
x[i] <- list(NULL)
```

to set component `i` of the list `x` to `NULL`, similarly for named components. Do not set `x[i]` or `x[[i]]` to `NULL`, because this will remove the corresponding component from the list.

For dropping the row names of a matrix `x`, it may be easier to use `rownames(x) <- NULL`, similarly for column names.

7.2 How can I save my workspace?

`save.image()` saves the objects in the user's `.GlobalEnv` to the file `.RData` in the R startup directory. (This is also what happens after *`q("yes")`*.) Using `save.image(`*`file`*`)` one can save the image under a different name.

7.3 How can I clean up my workspace?

To remove all objects in the currently active environment (typically `.GlobalEnv`), you can do

```
rm(list = ls(all.names = TRUE))
```

(Without `all = TRUE`, only the objects with names not starting with a '`.`' are removed.)

7.4 How can I get `eval()` and `D()` to work?

Strange things will happen if you use `eval(print(x), envir = e)` or `D(x^2, "x")`. The first one will either tell you that `"x"` is not found, or print the value of the wrong `x`. The other one will likely return zero if `x` exists, and an error otherwise.

This is because in both cases, the first argument is evaluated in the calling environment first. The result (which should be an object of mode `"expression"` or `"call"`) is then evaluated or differentiated. What you (most likely) really want is obtained by "quoting" the first argument upon surrounding it with `expression()`. For example,

```
R> D(expression(x^2), "x")
2 * x
```

Although this behavior may initially seem to be rather strange, it is perfectly logical. The "intuitive" behavior could easily be implemented, but problems would arise whenever the expression is contained in a variable, passed as a parameter, or is the result of a function call. Consider for instance the semantics in cases like

```
D2 <- function(e, n) D(D(e, n), n)
```

or

```
g <- function(y) eval(substitute(y), sys.frame(sys.parent(n = 2)))
g(a * b)
```

See the help page for `deriv()` for more examples.

7.5 Why do my matrices lose dimensions?

When a matrix with a single row or column is created by a subscripting operation, e.g., `row <- mat[2, ]`, it is by default turned into a vector. In a similar way if an array with dimension, say, 2 x 3 x 1 x 4 is created by subscripting it will be coerced into a 2 x 3 x 4 array, losing the unnecessary dimension. After much discussion this has been determined to be a *feature*.

To prevent this happening, add the option `drop = FALSE` to the subscripting. For example,

```
rowmatrix <- mat[2, , drop = FALSE]  # creates a row matrix
colmatrix <- mat[, 2, drop = FALSE]  # creates a column matrix
a <- b[1, 1, 1, drop = FALSE]        # creates a 1 x 1 x 1 array
```

The `drop = FALSE` option should be used defensively when programming. For example, the statement

```
somerows <- mat[index, ]
```

will return a vector rather than a matrix if `index` happens to have length 1, causing errors later in the code. It should probably be rewritten as

```
somerows <- mat[index, , drop = FALSE]
```

7.6 How does autoloading work?

Autoloading is rarely used since packages became lazy-loaded.

R has a special environment called `.AutoloadEnv`. Using *`autoload(name, pkg)`*, where *name* and *pkg* are strings giving the names of an object and the package containing it, stores some information in this environment. When R tries to evaluate *name*, it loads the corresponding package *pkg* and reevaluates *name* in the new package's environment.

Using this mechanism makes R behave as if the package was loaded, but does not occupy memory (yet).

See the help page for `autoload()` for a very nice example.

7.7 How should I set options?

The function `options()` allows setting and examining a variety of global "options" which affect the way in which R computes and displays its results. The variable `.Options` holds the current values of these options, but should never directly be assigned to unless you want to drive yourself crazy—simply pretend that it is a "read-only" variable.

For example, given

```
test1 <- function(x = pi, dig = 3) {
  oo <- options(digits = dig); on.exit(options(oo));
  cat(.Options$digits, x, "\n")
}
test2 <- function(x = pi, dig = 3) {
  .Options$digits <- dig
  cat(.Options$digits, x, "\n")
}
```

we obtain:

```
R> test1()
```

```
3 3.14
R> test2()
3 3.141593
```

What is really used is the *global* value of `.Options`, and using *`options(OPT = VAL)`* correctly updates it. Local copies of `.Options`, either in `.GlobalEnv` or in a function environment (frame), are just silently disregarded.

7.8 How do file names work in Windows?

As R uses C-style string handling, '\' is treated as an escape character, so that for example one can enter a newline as '`\n`'. When you really need a '\', you have to escape it with another '\'.

Thus, in filenames use something like `"c:\\data\\money.dat"`. You can also replace '\' by '/' (`"c:/data/money.dat"`).

7.9 Why does plotting give a color allocation error?

This is about a problem rarely seen with modern X11 installations.

On an X11 device, plotting sometimes, e.g., when running `demo("image")`, results in "Error: color allocation error". This is an X problem, and only indirectly related to R. It occurs when applications started prior to R have used all the available colors. (How many colors are available depends on the X configuration; sometimes only 256 colors can be used.)

You could also set the `colortype` of `X11()` to `"pseudo.cube"` rather than the default `"pseudo"`. See the help page for `X11()` for more information.

7.10 How do I convert factors to numeric?

It may happen that when reading numeric data into R (usually, when reading in a file), they come in as factors. If `f` is such a factor object, you can use

```
as.numeric(as.character(f))
```

to get the numbers back. More efficient, but harder to remember, is

```
as.numeric(levels(f))[as.integer(f)]
```

In any case, do not call `as.numeric()` or their likes directly for the task at hand (as `as.numeric()` or `unclass()` give the internal codes).

7.11 Are Trellis displays implemented in R?

The recommended package **lattice** (`https://CRAN.R-project.org/package=lattice`) (which is based on base package **grid**) provides graphical functionality that is compatible with most Trellis commands.

You could also look at `coplot()` and `dotchart()` which might do at least some of what you want. Note also that the R version of `pairs()` is fairly general and provides most of the functionality of `splom()`, and that R's default plot method has an argument `asp` allowing to specify (and fix against device resizing) the aspect ratio of the plot.

(Because the word "Trellis" has been claimed as a trademark we do not use it in R. The name "lattice" has been chosen for the R equivalent.)

7.12 What are the enclosing and parent environments?

Inside a function you may want to access variables in two additional environments: the one that the function was defined in ("enclosing"), and the one it was invoked in ("parent").

If you create a function at the command line or load it in a package its enclosing environment is the global workspace. If you define a function `f()` inside another function `g()` its enclosing environment is the environment inside `g()`. The enclosing environment for a function is fixed when the function is created. You can find out the enclosing environment for a function `f()` using `environment(f)`.

The "parent" environment, on the other hand, is defined when you invoke a function. If you invoke `lm()` at the command line its parent environment is the global workspace, if you invoke it inside a function `f()` then its parent environment is the environment inside `f()`. You can find out the parent environment for an invocation of a function by using `parent.frame()` or `sys.frame(sys.parent())`.

So for most user-visible functions the enclosing environment will be the global workspace, since that is where most functions are defined. The parent environment will be wherever the function happens to be called from. If a function `f()` is defined inside another function `g()` it will probably be used inside `g()` as well, so its parent environment and enclosing environment will probably be the same.

Parent environments are important because things like model formulas need to be evaluated in the environment the function was called from, since that's where all the variables will be available. This relies on the parent environment being potentially different with each invocation.

Enclosing environments are important because a function can use variables in the enclosing environment to share information with other functions or with other invocations of itself (see the section on lexical scoping). This relies on the enclosing environment being the same each time the function is invoked. (In C this would be done with static variables.)

Scoping *is* hard. Looking at examples helps. It is particularly instructive to look at examples that work differently in R and S and try to see why they differ. One way to describe the scoping differences between R and S is to say that in S the enclosing environment is *always* the global workspace, but in R the enclosing environment is wherever the function was created.

7.13 How can I substitute into a plot label?

Often, it is desired to use the value of an R object in a plot label, e.g., a title. This is easily accomplished using `paste()` if the label is a simple character string, but not always obvious in case the label is an expression (for refined mathematical annotation). In such a case, either use `parse()` on your pasted character string or use `substitute()` on an expression. For example, if `ahat` is an estimator of your parameter a of interest, use

```
title(substitute(hat(a) == ahat, list(ahat = ahat)))
```

(note that it is '`==`' and not '`=`'). Sometimes `bquote()` gives a more compact form, e.g.,

```
title(bquote(hat(a) = .(ahat)))
```

where subexpressions enclosed in '`.()`' are replaced by their values.

There are more examples in the mailing list archives.

7.14 What are valid names?

When creating data frames using `data.frame()` or `read.table()`, R by default ensures that the variable names are syntactically valid. (The argument `check.names` to these functions controls whether variable names are checked and adjusted by `make.names()` if needed.)

To understand what names are "valid", one needs to take into account that the term "name" is used in several different (but related) ways in the language:

1. A *syntactic name* is a string the parser interprets as this type of expression. It consists of letters, numbers, and the dot and underscore characters, and starts with either a letter or a dot not followed by a number. Reserved words are not syntactic names.
2. An *object name* is a string associated with an object that is assigned in an expression either by having the object name on the left of an assignment operation or as an argument to the `assign()` function. It is usually a syntactic name as well, but can be any non-empty string if it is quoted (and it is always quoted in the call to `assign()`).
3. An *argument name* is what appears to the left of the equals sign when supplying an argument in a function call (for example, `f(trim=.5)`). Argument names are also usually syntactic names, but again can be anything if they are quoted.
4. An *element name* is a string that identifies a piece of an object (a component of a list, for example.) When it is used on the right of the '`$`' operator, it must be a syntactic name, or quoted. Otherwise, element names can be any strings. (When an object is used as a database, as in a call to `eval()` or `attach()`, the element names become object names.)
5. Finally, a *file name* is a string identifying a file in the operating system for reading, writing, etc. It really has nothing much to do with names in the language, but it is traditional to call these strings file "names".

7.15 Are GAMs implemented in R?

Package **gam** (`https://CRAN.R-project.org/package=gam`) from CRAN implements all the Generalized Additive Models (GAM) functionality as described in the GAM chapter of the White Book. In particular, it implements backfitting with both local regression and smoothing splines, and is extendable. There is a `gam()` function for GAMs in package **mgcv** (`https://CRAN.R-project.org/package=mgcv`), but it is not an exact clone of what is described in the White Book (no `lo()` for example). Package **gss** (`https://CRAN.R-project.org/package=gss`) can fit spline-based GAMs too. And if you can accept regression splines you can use `glm()`. For Gaussian GAMs you can use `bruto()` from package **mda** (`https://CRAN.R-project.org/package=mda`).

7.16 Why is the output not printed when I source() a file?

Most R commands do not generate any output. The command

```
1+1
```

computes the value 2 and returns it; the command

```
summary(glm(y~x+z, family=binomial))
```

fits a logistic regression model, computes some summary information and returns an object of class `"summary.glm"` (see Section 8.1 [How should I write summary methods?], page 43).

If you type '`1+1`' or '`summary(glm(y~x+z, family=binomial))`' at the command line the returned value is automatically printed (unless it is `invisible()`), but in other circumstances, such as in a `source()`d file or inside a function it isn't printed unless you specifically print it.

To print the value use

```
print(1+1)
```

or

```
print(summary(glm(y~x+z, family=binomial)))
```

instead, or use `source(`*`file`*`, echo=TRUE)`.

7.17 Why does outer() behave strangely with my function?

As the help for `outer()` indicates, it does not work on arbitrary functions the way the `apply()` family does. It requires functions that are vectorized to work elementwise on arrays. As you can see by looking at the code, `outer(x, y, FUN)` creates two large vectors containing every possible combination of elements of `x` and `y` and then passes this to `FUN` all at once. Your function probably cannot handle two large vectors as parameters.

If you have a function that cannot handle two vectors but can handle two scalars, then you can still use `outer()` but you will need to wrap your function up first, to simulate vectorized behavior. Suppose your function is

```
foo <- function(x, y, happy) {
  stopifnot(length(x) == 1, length(y) == 1) # scalars only!
  (x + y) * happy
}
```

If you define the general function

```
wrapper <- function(x, y, my.fun, ...) {
  sapply(seq_along(x), FUN = function(i) my.fun(x[i], y[i], ...))
}
```

then you can use `outer()` by writing, e.g.,

```
outer(1:4, 1:2, FUN = wrapper, my.fun = foo, happy = 10)
```

Scalar functions can also be vectorized using `Vectorize()`.

7.18 Why does the output from anova() depend on the order of factors in the model?

In a model such as `~A+B+A:B`, R will report the difference in sums of squares between the models `~1`, `~A`, `~A+B` and `~A+B+A:B`. If the model were `~B+A+A:B`, R would report differences between `~1`, `~B`, `~A+B`, and `~A+B+A:B` . In the first case the sum of squares for `A` is comparing `~1` and `~A`, in the second case it is comparing `~B` and `~B+A`. In a non-orthogonal design (i.e., most unbalanced designs) these comparisons are (conceptually and numerically) different.

Some packages report instead the sums of squares based on comparing the full model to the models with each factor removed one at a time (the famous 'Type III sums of squares' from SAS, for example). These do not depend on the order of factors in the model. The question of which set of sums of squares is the Right Thing provokes low-level holy wars on R-help from time to time.

There is no need to be agitated about the particular sums of squares that R reports. You can compute your favorite sums of squares quite easily. Any two models can be compared with `anova(`*`model1, model2`*`)`, and `drop1(`*`model1`*`)` will show the sums of squares resulting from dropping single terms.

7.19 How do I produce PNG graphics in batch mode?

Under a Unix-like, if your installation supports the `type="cairo"` option to the `png()` device there should be no problems, and the default settings should just work. This option is not available for versions of R prior to 2.7.0, or without support for cairo. From R 2.7.0 `png()` by default uses the Quartz device on macOS, and that too works in batch mode.

Earlier versions of the `png()` device used the X11 driver, which is a problem in batch mode or for remote operation. If you have Ghostscript you can use `bitmap()`, which produces a PostScript or PDF file then converts it to any bitmap format supported by Ghostscript. On some installations this produces ugly output, on others it is perfectly satisfactory. Many systems now come with Xvfb from X.Org (`https://www.x.org/`) (possibly as an optional install), which is an X11 server that does not require a screen.

7.20 How can I get command line editing to work?

The Unix-like command-line interface to R can only provide the inbuilt command line editor which allows recall, editing and re-submission of prior commands provided that the GNU readline library is available at the time R is configured for compilation. Note that the 'development' version of readline including the appropriate headers is needed: users of Linux binary distributions will need to install packages such as `libreadline-dev` (Debian) or `readline-devel` (Red Hat).

7.21 How can I turn a string into a variable?

If you have

```
varname <- c("a", "b", "d")
```

you can do

```
get(varname[1]) + 2
```

for

```
a + 2
```

or

```
assign(varname[1], 2 + 2)
```

for

```
a <- 2 + 2
```

or

```
eval(substitute(lm(y ~ x + variable),
                list(variable = as.name(varname[1]))))
```

for

```
lm(y ~ x + a)
```

At least in the first two cases it is often easier to just use a list, and then you can easily index it by name

```
vars <- list(a = 1:10, b = rnorm(100), d = LETTERS)
vars[["a"]]
```

without any of this messing about. This becomes especially true if you are finding yourself creating and trying to programmatically access groups of related variables such as `result1`, `result2`, `result3`, and so on: instead of fighting against the language to use

```
assign(paste0("result", i), process(get(paste0("dataset", i))))
```

it is much easier to put the related variables in lists and use

```
result[[i]] <- process(dataset[[i]])
```

and, eventually,

```
result <- lapply(dataset, process)
```

which is easy to replace with `parLapply` for parallel processing.

7.22 Why do lattice/trellis graphics not work?

The most likely reason is that you forgot to tell R to display the graph. Lattice functions such as `xyplot()` create a graph object, but do not display it (the same is true of **ggplot2** (`https://CRAN.R-project.org/package=ggplot2`) graphics, and Trellis graphics in S-PLUS). The `print()` method for the graph object produces the actual display. When you use these functions interactively at the command line, the result is automatically printed, but in `source()` or inside your own functions you will need an explicit `print()` statement.

7.23 How can I sort the rows of a data frame?

To sort the rows within a data frame, with respect to the values in one or more of the columns, simply use `order()` (e.g., `DF[order(DF$a, DF[["b"]]), ]` to sort the data frame `DF` on columns named `a` and `b`).

From R 4.4.0, `sort_by()` provides a less verbose alternative with a formula interface (e.g., `sort_by(DF, ~a + b)`).

7.24 Why does the help.start() search engine not work?

Since R 2.10.0, the browser-based search engine in `help.start()` is an HTML interface to `help.search()`, and should always work. Before that, the engine utilized a Java applet. In order for this to function properly, one needed a compatible version of Java installed on the system and linked to the browser, and both Java *and* JavaScript enabled in the browser.

7.25 Why did my .Rprofile stop working when I updated R?

Did you read the `NEWS` file? For functions that are not in the **base** package you need to specify the correct package namespace, since the code will be run *before* the packages are loaded. E.g.,

```
ps.options(horizontal = FALSE)
help.start()
```

needs to be

```
grDevices::ps.options(horizontal = FALSE)
utils::help.start()
```

7.26 Where have all the methods gone?

Many functions, particularly S3 methods, are now hidden in namespaces. This has the advantage that they cannot be called inadvertently with arguments of the wrong class, but it makes them harder to view.

To see the code for an S3 method (e.g., `[.terms`) use

```
getS3method("[", "terms")
```

To see the code for an unexported function `foo()` in the namespace of package `"bar"` use `bar:::foo`. Don't use these constructions to call unexported functions in your own code—they are probably unexported for a reason and may change without warning.

7.27 How can I create rotated axis labels?

To rotate axis labels (using base graphics), you need to use `text()`, rather than `mtext()`, as the latter does not support `par("srt")`.

```
## Increase bottom margin to make room for rotated labels
par(mar = c(7, 4, 4, 2) + 0.1)
## Create plot with no x axis and no x axis label
plot(1 : 8, xaxt = "n",  xlab = "")
## Set up x axis with tick marks alone
axis(1, labels = FALSE)
## Create some text labels
labels <- paste("Label", 1:8, sep = " ")
## Plot x axis labels at default tick marks
text(1:8, par("usr")[3] - 0.25, srt = 45, adj = 1,
     labels = labels, xpd = TRUE)
## Plot x axis label at line 6 (of 7)
mtext(1, text = "X Axis Label", line = 6)
```

When plotting the x axis labels, we use `srt = 45` for text rotation angle, `adj = 1` to place the right end of text at the tick marks, and `xpd = TRUE` to allow for text outside the plot region. You can adjust the value of the `0.25` offset as required to move the axis labels up or down relative to the x axis. See `?par` for more information.

Also see Figure 1 and associated code in Paul Murrell (2003), "Integrating grid Graphics Output with Base Graphics Output", *R News*, **3/2**, 7–12.

7.28 Why is read.table() so inefficient?

By default, `read.table()` needs to read in everything as character data, and then try to figure out which variables to convert to numerics or factors. For a large data set, this takes considerable amounts of time and memory. Performance can substantially be improved by using the `colClasses` argument to specify the classes to be assumed for the columns of the table.

7.29 What is the difference between package and library?

A *package* is a standardized collection of material extending R, e.g. providing code, data, or documentation. A *library* is a place (directory) where R knows to find packages it can use (i.e., which were *installed*). R is told to use a package (to "load" it and add it to the search path) via calls to the function `library`. I.e., `library()` is employed to load a package from libraries containing packages.

See Chapter 5 [R Add-On Packages], page 20, for more details. See also Uwe Ligges (2003), "R Help Desk: Package Management", *R News*, **3/3**, 37–39.

7.30 I installed a package but the functions are not there

To actually *use* the package, it needs to be *loaded* using `library()`.

See Chapter 5 [R Add-On Packages], page 20, and Section 7.29 [What is the difference between package and library?], page 36, for more information.

7.31 Why doesn't R think these numbers are equal?

The only numbers that can be represented exactly in R's numeric type are integers and fractions whose denominator is a power of 2. All other numbers are internally rounded to (typically) 53 binary digits accuracy. As a result, two floating point numbers will not reliably be equal unless they have been computed by the same algorithm, and not always even then. For example

```
R> a <- sqrt(2)
R> a * a == 2
[1] FALSE
R> a * a - 2
[1] 4.440892e-16
R> print(a * a, digits = 18)
[1] 2.00000000000000044
```

The function `all.equal()` compares two objects using a numeric tolerance of `.Machine$double.eps ^ 0.5`. If you want much greater accuracy than this you will need to consider error propagation carefully.

A discussion with many easily followed examples is in Appendix G "Computational Precision and Floating Point Arithmetic", pages 753–771 of *Statistical Analysis and Data Display: An Intermediate Course with Examples in R*, Richard M. Heiberger and Burt Holland (Springer 2015, second edition). This appendix is a free download from `https://link.springer.com/content/pdf/bbm:978-1-4939-2122-5/1.pdf`.

For more information, see e.g. David Goldberg (1991), "What Every Computer Scientist Should Know About Floating-Point Arithmetic", *ACM Computing Surveys*, **23/1**, 5–48, also available via `https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html`.

Here is another example, this time using addition:

```
R> .3 + .6 == .9
[1] FALSE
R> .3 + .6 - .9
[1] -1.110223e-16
```

```
R> print(matrix(c(.3,  .6,  .9, .3 + .6)), digits = 18)
                      [,1]
[1,] 0.299999999999999989
[2,] 0.599999999999999978
[3,] 0.900000000000000022
[4,] 0.899999999999999911
```

7.32 How can I capture or ignore errors in a long simulation?

Use `try()`, which returns an object of class `"try-error"` instead of an error, or preferably `tryCatch()`, where the return value can be configured more flexibly. For example

```
beta[i,] <- tryCatch(coef(lm(formula, data)),
                     error = function(e) rep(NaN, 4))
```

would return the coefficients if the `lm()` call succeeded and would return `c(NaN, NaN, NaN, NaN)` if it failed (presumably there are supposed to be 4 coefficients in this example).

7.33 Why are powers of negative numbers wrong?

You are probably seeing something like

```
R> -2^2
[1] -4
```

and misunderstanding the precedence rules for expressions in R. Write

```
R> (-2)^2
[1] 4
```

to get the square of -2.

The precedence rules are documented in `?Syntax`, and to see how R interprets an expression you can look at the parse tree

```
R> as.list(quote(-2^2))
[[1]]
`-`

[[2]]
2^2
```

7.34 How can I save the result of each iteration in a loop into a separate file?

One way is to use `paste()` (or `sprintf()`) to concatenate a stem filename and the iteration number while `file.path()` constructs the path. For example, to save results into files `result1.rda`, ..., `result100.rda` in the subdirectory `Results` of the current working directory, one can use

```
for(i in 1:100) {
  ## Calculations constructing "some_object" ...
  fp <- file.path("Results", paste0("result", i, ".rda"))
  save(list = "some_object", file = fp)
}
```

7.35 Why are p-values not displayed when using `lmer()`?

Doug Bates has kindly provided an extensive response in a post to the r-help list, which can be reviewed at `https://stat.ethz.ch/pipermail/r-help/2006-May/094765.html`.

7.36 Why are there unwanted borders, lines or grid-like artifacts when viewing a plot saved to a PS or PDF file?

This can occur when using functions such as `polygon()`, `filled.contour()`, `image()` or other functions which may call these internally. In the case of `polygon()`, you may observe unwanted borders between the polygons even when setting the `border` argument to `NA` or `"transparent"`.

The source of the problem is the PS/PDF viewer when the plot is anti-aliased. The details for the solution will be different depending upon the viewer used, the operating system and may change over time. For some common viewers, consider the following:

Acrobat Reader (cross platform)
: There are options in Preferences to enable/disable text smoothing, image smoothing and line art smoothing. Disable line art smoothing.

Preview (macOS)
: There is an option in Preferences to enable/disable anti-aliasing of text and line art. Disable this option.

GSview (cross platform)
: There are settings for Text Alpha and Graphics Alpha. Change Graphics Alpha from 4 bits to 1 bit to disable graphic anti-aliasing.

gv (Unix-like X)
: There is an option to enable/disable anti-aliasing. Disable this option.

Evince (Linux/GNOME)
: There is not an option to disable anti-aliasing in this viewer.

Okular (Linux/KDE)
: There is not an option in the GUI to enable/disable anti-aliasing. From a console command line, use:

```
$ kwriteconfig --file okularpartrc --group 'Dlg Performance' \
               --key GraphicsAntialias Disabled
```

Then restart Okular. Change the final word to '`Enabled`' to restore the original setting.

7.37 Why does backslash behave strangely inside strings?

This question most often comes up in relation to file names (see Section 7.8 [How do file names work in Windows?], page 29) but it also happens that people complain that they cannot seem to put a single '\' character into a text string unless it happens to be followed by certain other characters.

To understand this, you have to distinguish between character strings and *representations* of character strings. Mostly, the representation in R is just the string with a single or double

quote at either end, but there are strings that cannot be represented that way, e.g., strings that themselves contain the quote character. So

```
> str <- "This \"text\" is quoted"
> str
[1] "This \"text\" is quoted"
> cat(str, "\n")
This "text" is quoted
```

The *escape sequences* '`\"`' and '`\n`' represent a double quote and the newline character respectively. Printing text strings, using `print()` or by typing the name at the prompt will use the escape sequences too, but the `cat()` function will display the string as-is. Notice that '`"\n"`' is a one-character string, not two; the backslash is not actually in the string, it is just generated in the printed representation.

```
> nchar("\n")
[1] 1
> substring("\n", 1, 1)
[1] "\n"
```

So how do you put a backslash in a string? For this, you have to escape the escape character. I.e., you have to double the backslash. as in

```
> cat("\\n", "\n")
\n
```

Some functions, particularly those involving regular expression matching, themselves use metacharacters, which may need to be escaped by the backslash mechanism. In those cases you may need a *quadruple* backslash to represent a single literal one.

In versions of R up to 2.4.1 an unknown escape sequence like '`\p`' was quietly interpreted as just '`p`'. Current versions of R emit a warning.

7.38 How can I put error bars or confidence bands on my plot?

Some functions will display a particular kind of plot with error bars, such as the `bar.err()` function in the **agricolae** (`https://CRAN.R-project.org/package=agricolae`) package, the `plotCI()` function in the **gplots** (`https://CRAN.R-project.org/package=gplots`) package, the `plotCI()` and `brkdn.plot()` functions in the **plotrix** (`https://CRAN.R-project.org/package=plotrix`) package and the `error.bars()`, `error.crosses()` and `error.bars.by()` functions in the **psych** (`https://CRAN.R-project.org/package=psych`) package. Within these types of functions, some will accept the measures of dispersion (e.g., `plotCI`), some will calculate the dispersion measures from the raw values (`bar.err`, `brkdn.plot`), and some will do both (`error.bars`). Still other functions will just display error bars, like the dispersion function in the **plotrix** (`https://CRAN.R-project.org/package=plotrix`) package. Most of the above functions use the `arrows()` function in the base **graphics** package to draw the error bars.

The above functions all use the base graphics system. The grid and lattice graphics systems also have specific functions for displaying error bars, e.g., the `grid.arrow()` function in the **grid** package, and the `geom_errorbar()`, `geom_errorbarh()`, `geom_pointrange()`, `geom_linerange()`, `geom_crossbar()` and `geom_ribbon()` functions in the **ggplot2** (`https://`

CRAN.R-project.org/package=ggplot2) package. In the lattice system, error bars can be displayed with Dotplot() or xYplot() in the **Hmisc** (https://CRAN.R-project.org/package=Hmisc) package and segplot() in the **latticeExtra** (https://CRAN.R-project.org/package=latticeExtra) package.

7.39 How do I create a plot with two y-axes?

Creating a graph with two y-axes, i.e., with two sorts of data that are scaled to the same vertical size and showing separate vertical axes on the left and right sides of the plot that reflect the original scales of the data, is possible in R but is not recommended. The basic approach for constructing such graphs is to use par(new=TRUE) (see ?par); functions twoord.plot() (in the **plotrix** (https://CRAN.R-project.org/package=plotrix) package) and doubleYScale() (in the **latticeExtra** (https://CRAN.R-project.org/package=latticeExtra) package) automate the process somewhat.

7.40 How do I access the source code for a function?

In most cases, typing the name of the function will print its source code. However, code is sometimes hidden in a namespace, or compiled. For a complete overview on how to access source code, see Uwe Ligges (2006), "Help Desk: Accessing the sources", *R News*, **6/4**, 43–45 (https://CRAN.R-project.org/doc/Rnews/Rnews_2006-4.pdf).

7.41 Why does summary() report strange results for the R^2 estimate when I fit a linear model with no intercept?

As described in ?summary.lm, when the intercept is zero (e.g., from y ~ x - 1 or y ~ x + 0), summary.lm() uses the formula $R^2 = 1 - \sum_i R_i^2 / \sum_i y_i^2$, which is different from the usual $R^2 = 1 - \sum R_i^2 / \sum_i (y_i - \text{mean}(y))^2$. There are several reasons for this:

- Otherwise the R^2 could be negative (because the model with zero intercept can fit *worse* than the constant-mean model it is implicitly compared to).
- If you set the slope to zero in the model with a line through the origin you get fitted values y*=0
- The model with constant, non-zero mean is not nested in the model with a line through the origin.

All these come down to saying that if you know *a priori* that $E[Y] = 0$ when $x = 0$ then the 'null' model that you should compare to the fitted line, the model where x doesn't explain any of the variance, is the model where $E[Y] = 0$ everywhere. (If you don't know a priori that $E[Y] = 0$ when $x = 0$, then you probably shouldn't be fitting a line through the origin.)

7.42 Why is R apparently not releasing memory?

This question is often asked in different flavors along the lines of "I have removed objects in R and run gc() and yet ps/top still shows the R process using a lot of memory", often on Linux machines.

This is an artifact of the way the operating system (OS) allocates memory. In general it is common that the OS is not capable of releasing all unused memory. In extreme cases it is possible that even if R frees almost all its memory, the OS can not release any of it due to its design and thus tools such as `ps` or `top` will report substantial amount of resident RAM used by the R process even though R has released all that memory. In general such tools do *not* report the actual memory usage of the process but rather what the OS is reserving for that process.

The short answer is that this is a limitation of the memory allocator in the operating system and there is nothing R can do about it. That space is simply kept by the OS in the hope that R will ask for it later. The following paragraph gives more in-depth answer with technical details on how this happens.

Most systems use two separate ways to allocate memory. For allocation of large chunks they will use `mmap` to map memory into the process address space. Such chunks can be released immediately when they are completely free, because they can reside anywhere in the virtual memory. However, this is a relatively expensive operation and many OSes have a limit on the number of such allocated chunks, so this is only used for allocating large memory regions. For smaller allocations the system can expand the data segment of the process (historically using the `brk` system call), but this whole area is always contiguous. The OS can only move the end of this space, it cannot create any "holes". Since this operation is fairly cheap, it is used for allocations of small pieces of memory. However, the side-effect is that even if there is just one byte that is in use at the end of the data segment, the OS cannot release any memory at all, because it cannot change the address of that byte. This is actually more common than it may seem, because allocating a lot of intermediate objects, then allocating a result object and removing all intermediate objects is a very common practice. Since the result is allocated at the end it will prevent the OS from releasing any memory used by the intermediate objects. In practice, this is not necessarily a problem, because modern operating systems can page out unused portions of the virtual memory so it does not necessarily reduce the amount of real memory available for other applications. Typically, small objects such as strings or pairlists will be affected by this behavior, whereas large objects such as long vectors will be allocated using `mmap` and thus not affected. On Linux (and possibly other Unix-like systems) it is possible to use the `mallinfo` system call (also see the mallinfo (`https://rforge.net/mallinfo`) package) to query the allocator about the layout of the allocations, including the actually used memory as well as unused memory that cannot be released.

7.43 How can I enable secure https downloads in R?

From R 4.2.0, `"libcurl"` download method is always available and used for HTTPS by default on all platforms. It has been used since R 3.3.0 everywhere but Windows where the default method `"wininet"` also supported HTTPS.

So nothing needs to be done to access '`https://`' websites in recent versions of R.

7.44 How can I get CRAN package binaries for outdated versions of R?

Since March 2016, Windows and macOS binaries of CRAN packages for old versions of R (released more than 5 years ago) are made available from a central CRAN archive server

instead of the CRAN mirrors. To get these, one should set the CRAN "mirror" element of the `repos` option accordingly, by something like

```
local({r <- getOption("repos")
       r["CRAN"] <- "http://CRAN-archive.R-project.org"
       options(repos = r)
      })
```

(see `?options` for more information).

8 R Programming

8.1 How should I write summary methods?

Suppose you want to provide a summary method for class `"foo"`. Then `summary.foo()` should not print anything, but return an object of class `"summary.foo"`, *and* you should write a method `print.summary.foo()` which nicely prints the summary information and invisibly returns its object. This approach is preferred over having `summary.foo()` print summary information and return something useful, as sometimes you need to grab something computed by `summary()` inside a function or similar. In such cases you don't want anything printed.

8.2 How can I debug dynamically loaded code?

Roughly speaking, you need to start R inside the debugger, load the code, send an interrupt, and then set the required breakpoints.

See Section "Finding entry points in dynamically loaded code" in *Writing R Extensions*. This manual is included in the R distribution, see Section 2.7 [What documentation exists for R?], page 6.

8.3 How can I inspect R objects when debugging?

The most convenient way is to call `R_PV` from the symbolic debugger.

See Section "Inspecting R objects when debugging" in *Writing R Extensions*.

8.4 How can I change compilation flags?

Suppose you have C code file for dynloading into R, but you want to use `R CMD SHLIB` with compilation flags other than the default ones (which were determined when R was built).

Starting with R 2.1.0, users can provide personal `Makevars` configuration files in `$HOME/.R` to override the default flags. See Section "Add-on packages" in *R Installation and Administration*.

8.5 How can I debug S4 methods?

Use the `trace()` function with argument `signature=` to add calls to the browser or any other code to the method that will be dispatched for the corresponding signature. See `?trace` for details.

9 R Bugs

9.1 What is a bug?

If R executes an illegal instruction, or dies with an operating system error message that indicates a problem in the program (as opposed to something like "disk full"), then it is certainly a bug. If you call `.C()`, `.Fortran()`, `.External()` or `.Call()` (or `.Internal()`) yourself (or in a function you wrote), you can always crash R by using wrong argument types (modes). This is not a bug.

Taking forever to complete a command can be a bug, but you must make certain that it was really R's fault. Some commands simply take a long time. If the input was such that you *know* it should have been processed quickly, report a bug. If you don't know whether the command should take a long time, find out by looking in the manual or by asking for assistance.

If a command you are familiar with causes an R error message in a case where its usual definition ought to be reasonable, it is probably a bug. If a command does the wrong thing, that is a bug. But be sure you know for certain what it ought to have done. If you aren't familiar with the command, or don't know for certain how the command is supposed to work, then it might actually be working right. For example, people sometimes think there is a bug in R's mathematics because they don't understand how finite-precision arithmetic works. Rather than jumping to conclusions, show the problem to someone who knows for certain. Unexpected results of comparison of decimal numbers, for example `0.28 * 100 != 28` or `0.1 + 0.2 != 0.3`, are not a bug. See Section 7.31 [Why doesn't R think these numbers are equal?], page 36, for more details.

Finally, a command's intended definition may not be best for statistical analysis. This is a very important sort of problem, but it is also a matter of judgment. Also, it is easy to come to such a conclusion out of ignorance of some of the existing features. It is probably best not to complain about such a problem until you have checked the documentation in the usual ways, feel confident that you understand it, and know for certain that what you want is not available. If you are not sure what the command is supposed to do after a careful reading of the manual this indicates a bug in the manual. The manual's job is to make everything clear. It is just as important to report documentation bugs as program bugs. However, we know that the introductory documentation is seriously inadequate, so you don't need to report this.

If the online argument list of a function disagrees with the manual, one of them must be wrong, so report the bug.

See also "Making sure it's a bug" in Bug Reporting in R (`https://www.r-project.org/bugs.html`) for more information.

9.2 How to report a bug

When you decide that there is a bug, it is important to report it and to report it in a way which is useful. What is most useful is an exact description of what commands you type, starting with the shell command to run R, until the problem happens. Always include the version of R, machine, and operating system that you are using; type `version` in R to print this.

The most important principle in reporting a bug is to report *facts*, not hypotheses or categorizations. It is always easier to report the facts, but people seem to prefer to strain to posit explanations and report them instead. If the explanations are based on guesses about how R is implemented, they will be useless; others will have to try to figure out what the facts must have been to lead to such speculations. Sometimes this is impossible. But in any case, it is unnecessary work for the ones trying to fix the problem.

For example, suppose that on a data set which you know to be quite large the command

```
R> data.frame(x, y, z, monday, tuesday)
```

never returns. Do not report that `data.frame()` fails for large data sets. Perhaps it fails when a variable name is a day of the week. If this is so then when others got your report they would try out the `data.frame()` command on a large data set, probably with no day of the week variable name, and not see any problem. There is no way in the world that others could guess that they should try a day of the week variable name.

Or perhaps the command fails because the last command you used was a method for `"["()` that had a bug causing R's internal data structures to be corrupted and making the `data.frame()` command fail from then on. This is why others need to know what other commands you have typed (or read from your startup file).

It is very useful to try and find simple examples that produce apparently the same bug, and somewhat useful to find simple examples that might be expected to produce the bug but actually do not. If you want to debug the problem and find exactly what caused it, that is wonderful. You should still report the facts as well as any explanations or solutions. Please include an example that reproduces (e.g., `https://en.wikipedia.org/wiki/Reproducibility`) the problem, preferably the simplest one you have found.

Invoking R with the `--vanilla` option may help in isolating a bug. This ensures that the site profile and saved data files are not read.

Before you actually submit a bug report, you should check whether the bug has already been reported and/or fixed. First, try the "Show open bugs new-to-old" or the search facility on `https://bugs.R-project.org/`. Second, consult `https://svn.R-project.org/R/trunk/doc/NEWS.Rd`, which records changes that will appear in the *next* release of R, including bug fixes that do not appear on the Bug Tracker. Third, if possible try the current r-patched or r-devel version of R. If a bug has already been reported or fixed, please do not submit further bug reports on it. Finally, check carefully whether the bug is with R, or a contributed package. Bug reports on contributed packages should be sent first to the package maintainer, and only submitted to the R-bugs repository by package maintainers, mentioning the package in the subject line.

A bug report can be generated using the function `bug.report()`. For reports on R this will open the R Bugzilla page at `https://bugs.R-project.org/`: for a contributed package it will open the package's bug tracker Web page or help you compose an email to the maintainer. Since 2016, only "members" (including all who have previously submitted bugs) can submit new bugs on the R Bugzilla. See "Where to submit bug reports and patches" on Bug Reporting in R (`https://www.r-project.org/bugs.html`) for more information.

There is a section of the bug repository for suggestions for enhancements for R labelled '`wishlist`'. Suggestions can be submitted in the same ways as bugs, but please ensure that the subject line makes clear that this is for the wishlist and not a bug report, for example by starting with '`Wishlist:`'.

Comments on and suggestions for the Windows port of R should be sent to `R-windows@R-project.org`.

Corrections to and comments on message translations should be sent to the last translator (listed at the top of the appropriate '`.po`' file) or to the translation team as listed at `https://developer.R-project.org/TranslationTeams.html`.

Acknowledgments

Of course, many many thanks to Robert and Ross for the R system, and to the package writers and porters for adding to it.

Special thanks go to Doug Bates, Peter Dalgaard, Paul Gilbert, Stefano Iacus, Fritz Leisch, Jim Lindsey, Thomas Lumley, Martin Maechler, Brian D. Ripley, Anthony Rossini, and Andreas Weingessel for their comments which helped me improve this FAQ.

More to come soon . . .