# R News

The Newsletter of the R Project — Volume 4/2, September 2004


# Editorial

by Thomas Lumley

This edition of R News accompanies the release of R 2.0.0. Comparing the length of the pre-1.0.0 and post-1.0.0 NEWS files shows that R has certainly changed enough since 1.0.0 for a new major version. The base R language has added formal classes, namespaces, exception handling, connections, and many packaging and QA tools, and done away with the underscore as an assignment operator. A Macintosh version has been added and the Windows GUI extended. Even more dramatic is the change in the number of contributed packages, some of which (lattice, nlme, gam) were on the wishlist back at 1.0.0, but many more that show how the range of R users has expanded. Version 2.0.0 does not introduce many radical changes, but Brian Ripley's article describes one important and long planned improvement to R's memory usage and Paul Murrell's article explains new graphics features that give users more control over details of rendering.

Since the last release, R has also moved (for security reasons) from a CVS archive to one based on the Subversion system and hosted by Martin Mächler at ETH Zurich. To download an up-to-the-minute snapshot of R you just need a Subversion client, which are available for many operating systems. You can also browse individual files at `https://svn.r-project.org/R`. (Yes, the website uses a self-signed certificate that your browser doesn't recognize. If this bothers you, you may not understand what `https` is for).

Reminding people again that Bioconductor is not just about RNA we have an article using some of the graph packages. It is usual at this point to comment that 'graph' here means the mathematical abstraction, not the picture, except that in this case the article is actually about pictures (graphs of graphs) showing the structure of biological pathways.

Tools for reproducible research have been an important feature of the R project. Previous issues of this newsletter have described Sweave and the use of vignettes. In this issue Roger Peng and Leah Welty write about the use of R to distribute data and reproducible analyses from a major epidemiologic research project, the National Morbidity and Mortality Air Pollution Study.

In other articles Paul Gilbert writes about automating maintenance and testing of sets of packages, and Jun Yan describes using BUGS and R together.


*Thomas Lumley*
*Department of Biostatistics*
*University of Washington, Seattle*
`thomas.lumley@R-project.org`


## Contents of this issue:

# Lazy Loading and Packages in R 2.0.0

*by Brian D. Ripley*

## Lazy Loading

One of the basic differences between R and S is how objects are stored. S stores its objects as files on the file system, whereas R stores objects in memory, and uses *garbage collection* from time to time to clear out unused objects. This led to some practical differences:

1. R can access objects faster, particularly on first use (although the difference is not as large as one might think, as both S and the file system will do caching).
2. R slows down the more objects that there are in memory.
3. R's performance is more sensitive to the number and size of packages that are loaded.

These differences are blurred considerably by the advent of *lazy loading* in R 2.0.0. This is optional, but is used by all the standard and recommended packages, and by default when a package with more than 25Kb of R code is installed (about 45% of those currently on CRAN). This is 'lazy' in the same sense as *lazy evaluation*, that is objects are not loaded into memory until they are actually used. This leads to some immediate differences:

1. R uses much less memory on startup: on my 32-bit system, 4.2Mb rather than 12.5Mb. Such a gain would have been very valuable in the early days of R, but nowadays most of us have far more RAM than those numbers.
2. The start-up time is much shorter: 0.4s on my system. This is almost entirely because many fewer objects have been loaded into memory, and loading them takes time.
3. Tasks run a little faster, as garbage collection takes much less time (again, because there are many fewer objects to deal with).
4. There is much less penalty in loading up lots of packages at the beginning of a session. (There is some, and loading R with just the **base** package takes under 0.1s.)

### For data, too

Another R/S difference has been the use of `data()` in R. As I understand it this arose because data objects are usually large and not used very often. However, we can apply lazy-loading to datasets as well as to other R objects, and the **MASS** package has done so since early 2003. This is optional when installing packages in 2.0.0 (and not the default), but applies to all standard and most recommended packages. So for example to make use of the dataset `heart` in package **survival**, just refer to it by name in your code.

There is one difference to watch out for: `data(heart)` loaded a copy of `heart` into the workspace, from the package highest on the search path offering such a dataset. If you subsequently altered `heart`, you got the altered copy, but using `data(heart)` again gave you the original version. It still does, and is probably the only reason to continue to use `data` with an argument.

For packages with namespaces there is a subtle difference: data objects are in `package:foo` but not in `namespace:foo`. This means that data set `fig` **cannot** be accessed as `foo::fig`. The reason is again subtle: if the objects were in the namespace then functions in **foo** would find `fig` from the namespace rather than the object of that name first on the search path, and modifications to `fig` would be ignored by some functions but by not others.

### Under the bonnet (or 'hood')

The implementation of lazy loading makes use of *promises*, which are user-visible through the use of the `delay` function. When R wants to find an object, it looks along a search path determined by the scope rules through a series of environments until it encounters one with a pointer to an object matching the name. So when the name `heart` is encountered in R code, R searches until it finds a matching variable, probably in `package:survival`. The pointer it would find there is to an object of type `PROMSXP` which contains instructions on how to get the real object, and evaluating it follows those instructions. The following shows the pattern

```
> library(survival)
> dump("heart", "", evaluate = FALSE)
heart <- delay(lazyLoadDBfetch(key, datafile,
       compressed, envhook), <environment>)
Warning message: deparse may be incomplete
```

The actual objects are stored in a simple database, in a format akin to the `.rda` objects produced by `save(compress = TRUE, ascii = FALSE)`. Function `lazyLoadDBfetch` fetches objects from such databases, which are stored as two files with extensions `.rdb` and `.rdx` (an index file). Readers nay be puzzled as to how `lazyLoadDBfetch` knows which object to fetch, as `key` seems to be unspecified. The answer lies (literally) in the environment shown as `<environment>` which was not dumped. The code in function `lazyLoad` contains essentially

```
wrap <- function(key) {
    key <- key
    mkpromise(expr, environment())
}
for (i in along(vars))
    set(vars[i], wrap(map$variables[[i]]), envir)
```

so key is found from the immediate environment and the remaining arguments from the enclosing environment of that environment, the body of lazyLoad.

This happens from normal R code completely transparently, perhaps with a very small delay when an object is first used. We can see how much by a rather unrealistic test:

```
> all.objects <-
    unlist(lapply(search(), ls, all.names=TRUE))
> system.time(sapply(all.objects,
                     function(x) get(x); TRUE),
             gcFirst = TRUE)
[1] 0.66 0.06 0.71 0.00 0.00
> system.time(sapply(all.objects,
                     function(x) get(x); TRUE),
             gcFirst = TRUE)
[1] 0.03 0.00 0.03 0.00 0.00
```

Note the use of the new gcFirst argument to system.time. This tells us that the time saved in start up would be lost if you were to load all 2176 objects on the search path (and there are still hidden objects in namespaces that have not been accessed).

People writing C code to manipulate R objects may need to be aware of this, although we have only encountered a handful of examples where promises need to be evaluated explicitly, all in R's graphical packages.

Note that when we said that there were many fewer objects to garbage collect, that does not mean fewer *named* objects, since each named object is still there, perhaps as a promise. It is rather that we do not have in memory the components of a list, the elements of a character vector and the components of the parse tree of a function, each of which are R objects. We can see this *via*

```
> gc()
         used (Mb) gc trigger (Mb)
Ncells 140236  3.8     350000  9.4
Vcells  52911  0.5     786432  6.0
> memory.profile()
    NILSXP     SYMSXP    LISTSXP     CLOSXP
         1       4565      70606        959
    ENVSXP    PROMSXP    LANGSXP SPECIALSXP
      2416       2724      27886        143
BUILTINSXP    CHARSXP     LGLSXP
       912      13788       1080          0
               INTSXP    REALSXP    CPLXSXP
         0       2303       2986          0
    STRSXP     DOTSXP     ANYSXP     VECSXP
      8759          0          0       1313
   EXPRSXP   BCODESXP  EXTPTRSXP WEAKREFSXP
         0          0         10          0
```

```
> sapply(all.objects,
         function(x) get(x); TRUE) -> junk
> gc()
         used (Mb) gc trigger (Mb)
Ncells 429189 11.5     531268 14.2
Vcells 245039  1.9     786432  6.0
> memory.profile()
    NILSXP     SYMSXP    LISTSXP     CLOSXP
         1       7405     222887       3640
    ENVSXP    PROMSXP    LANGSXP SPECIALSXP
       822       2906     101110        208
BUILTINSXP    CHARSXP     LGLSXP
      1176      44308       4403          0
               INTSXP    REALSXP    CPLXSXP
         0        824      11710          9
    STRSXP     DOTSXP     ANYSXP     VECSXP
     24877          0          0       2825
   EXPRSXP   BCODESXP  EXTPTRSXP WEAKREFSXP
         0          0        106          0
```

Notice the increase in the number of LISTSXP and LANGSXP (principally storing parsed functions) and STRSXP and CHARSXP (character vectors and their elements), and in the sum (the number of objects has trebled to over 400,000). Occasionally people say on R-help that they 'have no objects in memory', but R starts out with hundreds of thousands of objects.

## Installing packages

We have taken the opportunity of starting the 2.x.y series of R to require all packages to be reinstalled, and to do more computation when they are installed. Some of this is related to the move to lazy loading.

- The 'DESCRIPTION' and 'NAMESPACE' files are read and parsed, and stored in a binary format in the installed package's 'Meta' subdirectory.

- If either lazy loading of R code or a saved image has been requested, we need to load the code into memory and dump the objects created to a database or a single file 'all.rda'. This means the code has to parse correctly (not normally checked during installation), and all the packages needed to load the code have to be already installed.

  This is simplified by accurate use of the 'Describe', 'Suggests' and 'Import' fields in the 'DESCRIPTION' file: see below.

- We find out just what data() can do. Previously there was no means of finding out what, say, data(sunspot) did without trying it (and in the base system it created objects sunspot.months and sunspot.years but not sunspot, but not after package **lattice** was loaded). So we do try loading all the possible datasets—this not only tests that they work but gives us an index of datasets which is stored in binary format and used by `data()` (with no argument).

We have always said any R code used to make datasets has to be self-sufficient, and now this is checked.

- If lazy loading of data is requested, the datasets found in the previous step are dumped into a database in the package directory `data`.

If we need to have one package installed to install another we have a dependency graph amongst packages. Fortuitously, installing CRAN packages in alphabetical order has worked (and still did at the time of writing), even though for example **RMySQL** required **DBI**. However, this is not true of BioConductor packages and may not remain true for CRAN, but `install.packages` is able to work out a feasible install order and use that. (It is also now capable of finding all the packages which need already to be installed and installing those first: just ask for its help!)

One problem with package **A** `require()`ing package **B** in `.First.lib`/`.onLoad` was that package **B** would get loaded after package **A** and so precede it on the search path. This was particularly problematic if **A** made a function in **B** into an S4 generic, and the file 'install.R' was used to circumvent this (but this only worked because it did not work as documented!).

We now have a much cleaner mechanism. All packages mentioned in the '`Depends`' field of the 'DESCRIPTION' file of a package are loaded in the order they are mentioned, both before the package is prepared for lazy-loading/save image and before it is loaded by `library`. Many packages currently have unneeded entries in their '`Depends`' field (often to packages that no longer exist) and will hopefully be revised soon. The current description from *'Writing R Extensions'* is

- Packages whose namespace only is needed to load the package using `library(`*pkgname*`)` must be listed in the '`Imports`' field.
- Packages that need to attached to successfully load the package using `library(`*pkgname*`)` must be listed in the '`Depends`' field.
- All packages that are needed to successfully run `R CMD check` on the package must be listed in one of '`Depends`' or '`Suggests`' or '`Imports`'.

## For Package Writers

The previous section ended with a plea for accurate 'DESCRIPTION' files. The 'DESCRIPTION' file is where a package writer can specify if lazy loading of code is to be allowed or mandated or disallowed (*via* the '`LazyLoad`' field), and similarly for lazy loading of datasets (*via* the '`LazyData`' field). Please make use of these, as otherwise a package can be installed with options to `R CMD INSTALL` that may override your intentions and make your documentation inaccurate.

Large packages that make use of saved images would benefit from being converted to lazy loading. It is possible to first save an image then convert the saved image to lazy-loading, but this is almost never necessary. The normal conversion route is to get the right '`Depends`' and '`Imports`' fields, add '`LazyLoad: yes`' then remove the 'install.R' file.

For a few packages lazy loading will be of little benefit. One is John Fox's **Rcmdr**, which accesses virtually all its functions on startup to build its menus.

## Acknowledgement

Lazy loading was (yet another) idea from Luke Tierney, who wrote the first implementation as a package **lazyload**.

*Brian D. Ripley*
*University of Oxford, UK*
`ripley@stats.ox.ac.uk`

# Fonts, Lines, and Transparency in R Graphics

*by Paul Murrell*

## Introduction

For R version 2.0.0, a number of basic graphics facilities have been added to allow finer control of fonts, line styles, and transparency in R graphics. At the user level, these changes involve the addition of graphical parameters in the **graphics** and **grid** (Murrell, 2002) packages for specifying fonts and line styles, and, in the **grDevices** package, some new functions for controlling font specifications and changes to some existing functions for specifying semitransparent colours. Summaries of the changes to the user interface and the availability of the changes on the standard devices are given in Tables 2 and 3 at the end of the article.

## Now you can choose your family

The specification of a particular font can be a complex task and it is very platform dependent. For example, the X11 specification for a Courier font is of the form `"-*-courier-%s-%s-*-*-%d-*-*-*-*-*-*-*"` while for Windows the specification is something like `"TT Courier"`.

To make things simpler and more standardised, R graphics has adopted a device-independent mechanism for specifying fonts which is loosely modelled on the specification of fonts in the Cascading Style Sheets specification (CSS, 1998).

### User interface

R graphics provides three basic parameters for specifying a font: the font family, the font face, and the font size.

The specification of font size and face have not changed. The font size is controlled both by an absolute pointsize, via `par(ps)` in the **graphics** packages or `gpar(fontsize)` in the **grid** package, and a relative multiplier, `par(cex)` or `gpar(cex)`. The font face is specified in the **graphics** package via `par(face)` as an integer between `1` and `5` and in the **grid** package via `gpar(fontface)` as a string: `"plain"`, `"bold"`, `"italic"` (or `"oblique"`), `"bold.italic"`, or `"symbol"`.

The specification of font families is where the changes have been made. All graphics devices define an initial or default font family when the device is created. This is typically a sans-serif font such as Helvetica or Arial. A new font family is specified via `par(family)` in the **graphics** package or `gpar(fontfamily)` in the **grid** package using a device-independent family name.

Four standard families, `"serif"`, `"sans"`, `"mono"`, and `"symbol"` are provided by default and more may be defined. Devices with support for font families provide a font database which is used to map the device-independent font family to a device-specific font family specification. For example, the standard mappings for the Quartz device are shown below.

```
> quartzFonts()

$serif
[1] "Times-Roman"
[2] "Times-Bold"
[3] "Times-Italic"
[4] "Times-BoldItalic"

$sans
[1] "Helvetica"
[2] "Helvetica-Bold"
[3] "Helvetica-Italic"
[4] "Helvetica-BoldOblique"

$mono
[1] "Courier"
[2] "Courier-Bold"
[3] "Courier-Oblique"
[4] "Courier-BoldOblique"

$symbol
[1] "Symbol" "Symbol" "Symbol"
[4] "Symbol"
```

For each of the standard devices there is a new function of the form `<dev>Font()` for defining new mappings and a new function of the form `<dev>Fonts()` for querying and modifying font family mappings and for assigning new mappings to the font database for the device (see Table 2).

This approach means that it is now possible to modify the graphics font family on all of the core graphics devices (it was only previously possible on Windows), and the font family specification is now consistent across all devices (it is device-independent and the interface for modifying the font family and/or specifying new font families is consistent).

### Hershey fonts

It is possible to specify a Hershey vector font (Hershey, 1969) as the font family. These fonts are device-independent and are drawn by R so they are available on all devices. Table 1 lists the Hershey font families that are provided.

Not all families support the standard font faces and there are three non-standard font faces supported by the "HersheySerif" family: face 5 is a Cyrillic font, face 6 is an oblique Cyrillic font, and face 7 provides a set of Japanese characters (for more information, see help(Hershey)).

## Device support

All of the standard graphics devices provide device-independent font family mappings, however there are some limitations. For a start, the user is responsible for ensuring that all fonts are available and installed correctly on the relevant system.

The PostScript device also requires that you specify font metric files for the font you are mapping to. Furthermore, all fonts that are to be used on a PostScript device must be "declared" when the device is created (see the new fonts argument to postscript()). Finally, it is not possible to modify a PostScript font family mapping while the mapping is being used on a device.

The PDF device uses the PostScript font database (there is neither a pdfFont() nor a pdfFonts() function). Also, the PDF device does not embed fonts in the PDF file, which means that the only font families that can be mapped to are the "base 14" fonts that are assumed to be available in a PDF viewer: "Times" or "Times New Roman", "Helvetica" or "Arial", "Courier", "Symbol", and "ZapfDingbats".[1]

On the Windows graphics device, the font family mapping will override the mapping in the Rdevga file when the font family is not "" and the font face is between 1 ("plain") and 4 ("bold.italic").

## An example

The following code produces output demonstrating a mathematical annotation (see Figure 1). The text that shows the code used to produce the annotation is is drawn in a "mono" font and the text that shows what the annotation looks like is drawn with a "serif" font. This example runs unaltered on all of the core graphics devices.

```
> expr <- "z[i] == sqrt(x[i]^2 + y[i]^2)"
> grid.text(paste("expression(",
+     expr, ")", sep = ""),
+     y = 0.66, gp = gpar(fontfamily = "mono",
+         cex = 0.7))
> grid.text(parse(text = expr),
+     y = 0.33, gp = gpar(fontfamily = "serif"))
```

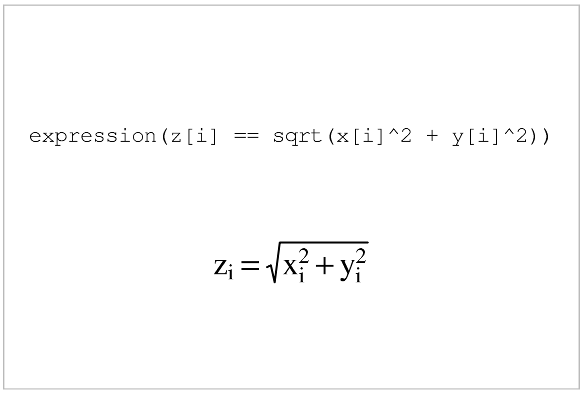



Figure 1: Simple R graphics output using "mono" and "serif" font families.

## The end of the line

The concepts of line ending style and line join style have been added to R graphics.

All lines are drawn using a particular style for line ends and joins, though the difference only becomes obvious when lines become thick. Figure 2 shows an extreme example, where three very wide lines (one black, one dark grey, and one light grey) have been drawn through exactly the same three locations. The locations are shown as black dots.

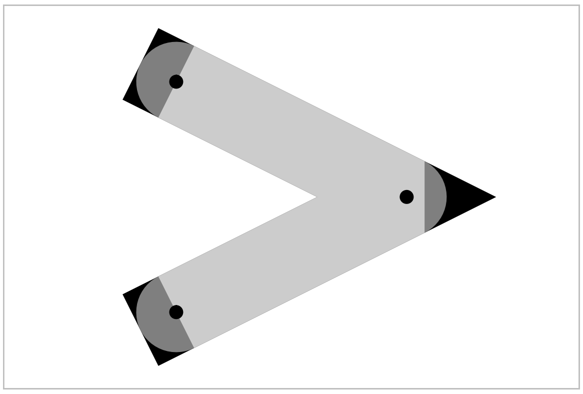


Figure 2: The different line end and line join styles.

[1]The "14" comes from the fact that there are four different faces for each of the Times, Helevetica/Arial, and Courier font families.

Table 1: The Hershey vector fonts available in R.

| Family | Description | Supported faces |
|---|---|---|
| "HersheySerif" | Serif font family | 1 to 7 |
| "HersheySans" | Sans serif font family | 1 to 4 |
| "HersheyScript" | Script (handwriting) font family | 1 and 2 |
| "HersheyGothicEnglish" | Various gothic font families | 1 |
| "HersheyGothicGerman" | | |
| "HersheyGothicItalian" | | |
| "HersheySymbol" | Serif symbol font family | 1 to 4 |
| "HersheySansSymbol" | Sans serif symbol font family | 1 and 3 |

The black line is drawn with `"square"` ends and a `"mitre"` join style, the dark grey line is drawn with `"round"` ends and a `"round"` join style, and the light grey line is drawn with `"butt"` ends and a `"bevel"` join style.

When the line join style is `"mitre"`, the join style will automatically be converted to `"bevel"` if the angle at the join is too small. This is to avoid ridiculously pointy joins. The point at which the automatic conversion occurs is controlled by a mitre limit parameter, which is the ratio of the length of the mitre divided by the line width. The default value is `10` which means that the conversion occurs for joins where the angle is less than 11 degrees. Other standard values are `2`, which means that conversion occurs at angles less than 60 degrees, and `1.414` which means that conversion occurs for angles less than 90 degrees. The minimum mitre limit value is `1`.

It is important to remember that line join styles influence the corners on rectangles and polygons as well as joins in lines.

### User interface

The current line end style, line join style, and line mitre limit can be queried and set in the **graphics** package via new `par()` settings: `lend`, `ljoin`, and `lmitre` respectively.

In the **grid** package, the parameters are `"lineend"`, `"linejoin"`, and `"linemitre"`, and they are settable via the `gp` argument of any viewport or graphics function using the `gpar()` function.

### Device support

Line end styles and line join styles are not available on the Windows graphics device and it is not possible to control the line mitre limit on the X11 device (it is fixed at 10).

## Fine tuning the alpha channel

It is now possible to define colours with a full alpha channel in R.

The alpha channel controls the transparency level of a colour; a colour with an alpha value of 0 is fully transparent and an alpha value of 1 (or 255, depending on the scale being used) means the colour is fully opaque. Anything in between is semitransparent.

### User interface

Colours may be specified with an alpha channel using the new `alpha` argument to the `rgb()` and `hsv()` functions. By default opaque colours are created.

It is also possible to specify a colour using a string of the form `"#RRGGBBAA"` where each pair of characters gives a hexadecimal value in the range 0 to 255 and the `AA` pair specify the alpha channel.

The function `col2rgb()` will report the alpha channel for colours if the new `alpha` argument is `TRUE` (even if the colour is opaque). Conversely, it will *not* print the alpha channel, even for semitransparent colours, if the `alpha` argument is `FALSE`.

When colours are printed, anything with an alpha channel of 0 is printed as `"transparent"`. Known (opaque) colours are printed using their R colour name, e.g., `rgb(1, 0, 0)` is printed as `"red"`. Otherwise, opaque colours are printed in the form `"#RRGGBB"` and semitransparent colours are printed as `"#RRGGBBAA"`.

In the **grid** package, there is also an `alpha` graphical parameter (specified via `gpar()`), which controls a general alpha channel setting. This setting is combined with the alpha channel of individual colours by multiplying the two alpha values together (on the $[0, 1]$ scale). For example, if a viewport is pushed with `alpha=0.5` then everything drawn within that viewport will be semitransparent.

## Device support

Most graphics devices produce no output whatsoever for any colour that is not fully opaque. Only the PDF and Quartz devices will render semitransparent colour (and, for the PDF device, only when the new `version` argument to `pdf()` is set to `"1.4"` or higher).

## An example

A simple example of the application of transparency in a statistical graphic is to provide a representation of the density of data when points overlap. The following code plots 500 random values where each data symbol is drawn with a semitransparent blue colour. Where more points overlap, the blue becomes more saturated (see Figure 3).

```
> pdf("alpha.pdf", version = "1.4",
+     width = 4, height = 4)
> par(mar = c(5, 4, 2, 2))
> plot(rnorm(500), rnorm(500),
+     col = rgb(0, 0, 1, 0.2),
+     pch = 16)
> dev.off()
```

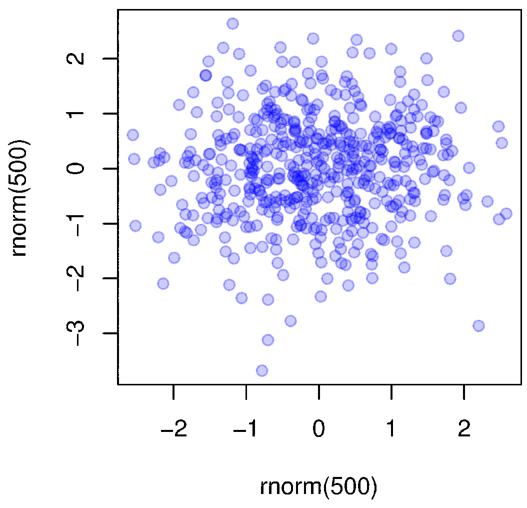



Figure 3: An application of alpha transparency.

# Bibliography

CSS2 specification. Technical report, World Wide Web Consortium, http://www.w3.org/TR/1998/REC-CSS2-19980512, May 1998. The latest version of CSS2 is available at http://www.w3.org/TR/REC-CSS2. 5

A. V. Hershey. Fortran IV programming for cartography and typography. Technical Report TR-2339, U. S. Naval Weapons Laboratory, Dahlgren, Virginia, 1969. 6

P. Murrell. The grid graphics package. *R News*, 2(2): 14–19, June 2002. URL `http://CRAN.R-project.org/doc/Rnews/`. 5

Table 2: Summary of the availability of new features on the standard graphics devices. A dot (•) means that the feature is supported on the device.

| Feature | PostScript | PDF | X11 | Windows | Quartz |
|---|---|---|---|---|---|
| Font family | • | • | • | • | • |
| Line end style | • | • | • | | • |
| Line join style | • | • | • | | • |
| Line mitre limit | • | • | | | • |
| Alpha channel | | • | | | • |

Table 3: Summary of the new and changed functions in R 2.0.0 relating to fonts, line styles, and transparency.

| Package | Function | Description |
|---|---|---|
| **grDevices** | `rgb()` | New `alpha` argument for specifying alpha channel. |
| | `hsv()` | New `alpha` argument for specifying alpha channel. |
| | `col2rgb()` | New `alpha` argument for reporting alpha channel. |
| | `postscriptFont()` | Generates a PostScript-specific font family description. |
| | `postscriptFonts()` | Defines and reports font mappings used by the PostScript and PDF devices. |
| | `windowsFont()` | Generates a Windows-specific font family description. |
| | `windowsFonts()` | Defines and reports font mappings used by the Windows device. |
| | `quartzFont()` | Generates a Quartz-specific font family description. |
| | `quartzFonts()` | Defines and reports font mappings used by the Quartz device. |
| | `X11Font()` | Generates an X11-specific font family description. |
| | `X11Fonts()` | Defines and reports font mappings used by the X11 device. |
| **graphics** | `par()` | New parameters `lend`, `ljoin`, and `lmitre` for controlling line style. New `family` parameter for controlling font family. |
| **grid** | `gpar()` | New graphical parameters, `lineend`, `linejoin`, and `linemitre` for controlling line style. The `alpha` parameter now affects the alpha channel of colours. The `family` parameter now affects graphical devices. |

# The NMMAPSdata Package

*by Roger D. Peng, Leah J. Welty*

The NMMAPSdata package for R contains daily mortality, air pollution, and weather data that were originally assembled for the National Morbidity, Mortality, and Air Pollution Study (NMMAPS). NMMAPS was a large multi-city time series study of the short-term effects of ambient air pollution on daily mortality and morbidity in the United States. The analyses of the original 90 city, 8 year database can be found in Samet et al. (2000a), Samet et al. (2000b), Dominici et al. (2003), and Daniels et al. (2004). The database has since been updated to contain data on 108 U.S. cities for 14 years (1987–2000). While the original study examined morbidity outcomes such as hospital admissions, the NMMAPSdata package does not include those data.

The NMMAPSdata package can be downloaded from the iHAPSS website.[1] The package does not currently reside on CRAN, although it passes all `R CMD check` quality control tests. A source package as well as a Windows binary package are available for download. All comments that follow pertain to version 0.3-4 of the package.

In this article we provide a very brief introduction to the data and to the R functions provided in the NMMAPSdata package. As an example of how one may use the package, we present a a small multi-city time series analysis of daily non-accidental mortality and $PM_{10}$. A more detailed description of the NMMAPSdata package and additional examples of time series models for air pollution and mortality are available in a technical report (Peng et al., 2004).

## A Brief Summary of the Data

The data are divided into 108 separate dataframes, one per city. Each dataframe has 15,342 rows and 291 columns. Although there are only 14 years of daily observations (5,114 days), the mortality data are split into three age categories, resulting in each of the weather and pollution variables being repeated three times. The dataframes are set up in this manner so that they can be used immediately in a regression analysis function such as `lm` or `glm` to fit models similar to those used in NMMAPS. Those not interested in using the separate age categories can collapse the outcome counts with the `collapseEndpoints` preprocessing function included in the package (see the next section for how to do this).

The measured pollutants in NMMAPSdata are $PM_{10}$, $PM_{2.5}$, $SO_2$, $O_3$, $NO_2$, and CO. These are the six "criteria pollutants" defined by the U.S. Environmental Protection Agency. Most cities have measurements for the gases ($SO_2$, $O_3$, $NO_2$, CO) every day and measurements for $PM_{10}$ once every six days. Only a handful of cities have daily measurements of $PM_{10}$. Beginning in 1999, most cities have daily $PM_{2.5}$ measurements.

The meteorological variables included in the database are temperature, relative humidity, and dew point temperature. We also include as separate variables in the dataframes three day running means of temperature and dew point temperature.

General information about the data and how they were assembled can be found in Samet et al. (2000a). Interested readers are also encouraged to visit the Internet-based Health and Air Pollution Surveillance System (IHAPSS) website at `http://www.ihapss.jhsph.edu/` which contains more details about how the data were originally processed.

## Overview of NMMAPSdata

The NMMAPSdata package can be loaded into R in the usual way.

```
> library(NMMAPSdata)

NMMAPS Data (version 0.3-4)
    Type '?NMMAPS' for a brief
    introduction to the NMMAPS
    database.  Type 'NMMAPScite()'
    for information on how to cite
    'NMMAPSdata' in publications.
    A short tutorial vignette is
    available and can be viewed by
    typing
    'vignette("NMMAPSdata")'
```

Some introductory material regarding the database can be found by typing `?NMMAPS` at the command line.

The primary function in NMMAPSdata is `buildDB`, which can be used to build custom versions of the full NMMAPS database. In particular, most users will not need to use the entire database (291 variables for each of 108 cities) at any given time. The custom versions of the database may also contain transformations of variables or newly created/derived variables. Possible new variables include:

- Exclusions: Remove days with extreme pollution, mortality, or temperature
- Fill in occasional/sporadic missing data
- Create seasonal indicators

[1] `http://www.ihapss.jhsph.edu/data/NMMAPS/R/`

- Compute running means of variables

There are, of course, many other possibilities.

The function `buildDB` has one required argument, `procFunc`, a processing function (or function name) which will be applied to the city dataframes. By default, `buildDB` applies the processing function to all cities in the NMMAPS package. However, if a character vector with abbreviated city names is supplied to argument `cityList`, the processing function will be applied only to the cities in that list.

```
> args(buildDB)

function (procFunc, dbName,
    path = system.file("db",
                  package = "NMMAPSdata"),
    cityList = NULL, compress = FALSE,
    verbose = TRUE, ...)
NULL
```

By default, `buildDB` builds a new database in the package installation directory. If installing the new database in this location is not desirable, one can specify another directory via the `path` argument.

The function specified in the `procFunc` argument should return a (possibly modified) dataframe or `NULL`. If `procFunc` returns `NULL` when it is applied to a particular city, `buildDB` will skip that city and not include the dataframe in the new database. This is useful for excluding cities that do not contain observations for a particular pollutant without having to directly specify a list of cities to include.

Once a database is constructed using `buildDB`, it is *registered* via call to `registerDB`. When `registerDB` is called with no arguments it sets the full (unprocessed) NMMAPS database as the currently registered database. The argument `dbName` can be used to register other previously built databases, however, only one database can be registered at a time. The processing function used to create the new database is always stored with the newly created database, ensuring that all of the transformations to the original data are documented with code.

```
> registerDB()
> showDB()

Currently using full NMMAPS database
```

Each of the city dataframes can be loaded, read, or attached using `loadCity`, `readCity`, or `attachCity`, respectively. For example we can load, read, or attach the full New York City dataframe.

```
> loadCity("ny")
> ny[1:5, 2:6]

      date dow agecat accident copd
1 19870101   5      1       10    3
2 19870102   6      1        4    4
3 19870103   7      1        5    0
4 19870104   1      1        5    1
5 19870105   2      1        2    2
```

```
> dframe <- readCity("ny")
> identical(dframe, ny)

[1] TRUE

> attachCity("ny")
> search()

 [1] ".GlobalEnv"
 [2] "ny"
 [3] "package:NMMAPSdata"
 [4] "package:tools"
 [5] "package:methods"
 [6] "package:stats"
 [7] "package:graphics"
 [8] "package:utils"
 [9] "Autoloads"
[10] "package:base"
```

We can print the first 10 days of death counts from cardiovascular disease and non-accidental deaths for people $< 65$ years old:

```
> cvd[1:10]

 [1] 22 20 17 18 14 18 17 16 25 20

> death[1:10]

 [1] 73 68 56 55 60 80 64 63 64 65
```

The function `attachCity` will mostly likely only be useful for interactive work. Furthermore, only one city dataframe can be usefully attached at a time since all of the variables in the most recently attached dataframe will mask the variables in previously attached dataframes.

## Example: Analysis of $\mathrm{PM}_{10}$ and Mortality

In this section we illustrate how to fit models similar to those used in Dominici et al. (2002a,b, 2003). The basic NMMAPS model for a single city is an overdispersed Poisson model of the following form

$$
\begin{aligned}
Y_t &\sim \text{Poisson}(\mu_t) \\
\log \mu_t &= \text{DOW}_t + \text{AgeCat} \\
&\quad + ns(\text{temp}_t, df = 6) \\
&\quad + ns(\text{temp}_{t,1-3}, df = 6) \\
&\quad + ns(\text{dewpt}_t, df = 3) \\
&\quad + ns(\text{dewpt}_{t,1-3}, df = 3) \\
&\quad + ns(t,\ df = 7 \times \#\,\text{years}) \\
&\quad + ns(t,\ df = 1 \times \#\,\text{years}) \times \text{AgeCat} \\
&\quad + \beta \text{PM}_t \\
\text{Var}(Y_t) &= \phi \mu_t
\end{aligned}
\tag{1}
$$

where $Y_t$ is the number of non-accidental deaths on day $t$ for a particular age category, AgeCat is an indicator for the age category, $\text{temp}_t$ is the average temperature on day $t$, $\text{temp}_{t,1-3}$ is a running mean of temperature for the previous 3 days, and $\text{PM}_t$ is the $\text{PM}_{10}$ level for day $t$. The variables $\text{dewpt}_t$ and $\text{dewpt}_{t,1-3}$ are current day and running mean of dew point temperature. The age categories used here are $\geq 75$ years old, 65–74, and $< 65$. Each of the temperature and dewpoint temperature variables are related to mortality in a flexible manner via the smooth function $ns(\cdot, df)$, which indicates a natural spline with $df$ degrees of freedom.

To process the data in preparation for fitting model (1) to $\text{PM}_{10}$ and mortality data, one can use the built-in `basicNMMAPS` function as the argument to `procFunc` in `buildDB`. The function first checks the dataframe to see if it contains any $\text{PM}_{10}$ data. If there is no $\text{PM}_{10}$ data, then `NULL` is returned and `buildDB` skips the city. For cities with $\text{PM}_{10}$ data, days with extreme mortality counts are set to `NA` (missing) using an indicator variable included in the dataframe. Then the function coerces the day-of-week and age category variables to `factor` type and creates some age category indicators. Finally, a subset of the pollution (seven lags of $\text{PM}_{10}$), weather (temperature and dewpoint), and mortality (total non-accidental deaths, deaths from cardiovascular disease, and deaths from respiratory diseases) variables are retained and the reduced dataframe is returned.

In order to illustrate how `basicNMMAPS` works, we use it outside `buildDB` to build a customized dataframe for New York. After looking at the body of `basicNMMAPS`, we register the full NMMAPS database, load the database for New York specifically, then using `basicNMMAPS` create the customized dataframe called `ny2`.

```
> body(basicNMMAPS)

{
  if (all(is.na(dataframe[,"pm10tmean"])))
      return(NULL)
  is.na(dataframe[, "death"]) <-
    as.logical(dataframe[, "markdeath"])
  is.na(dataframe[, "cvd"]) <-
    as.logical(dataframe[, "markcvd"])
  is.na(dataframe[, "resp"]) <-
    as.logical(dataframe[, "markresp"])
  Age2Ind <-
    as.numeric(dataframe[,"agecat"] == 2)
  Age3Ind <-
    as.numeric(dataframe[,"agecat"] == 3)
  dataframe[, "dow"] <-
    as.factor(dataframe[, "dow"])
  dataframe[, "agecat"] <-
    as.factor(dataframe[, "agecat"])
  varList <- c("cvd", "death", "resp",
               "tmpd", "rmtmpd", "dptp",
               "rmdptp", "time", "agecat",
               "dow", "pm10tmean",
               paste("l", 1:7,"pm10tmean",
                     sep = ""))
  data.frame(dataframe[, varList],
             Age2Ind = Age2Ind,
             Age3Ind = Age3Ind)
}
```

```
> registerDB(NULL)
> loadCity("ny")
> ny2 <- basicNMMAPS(ny)
> str(ny2)

`data.frame':        15342 obs. of
    20 variables:
 $ cvd        : num  22 20 17 18 14  ...
 $ death      : num  73 68 56 55 60 ...
 $ resp       : num  6 5 3 3 4 3 5 2 ...
 $ tmpd       : num  34.5 36.5 35.8 ...
 $ rmtmpd     : num     NA    NA    NA ...
 $ dptp       : num  33.2 29.8 23.3 ...
 $ rmdptp     : num    NA   NA   NA 9.70 ...
 $ time       : num  -2556 -2556 <...>
 $ agecat     : Factor w/ 3 levels
                "1","2","3": 1 1 1 ...
 $ dow        : Factor w/ 7 levels
                "1","2","3","4",..: 5 6 ...
 $ pm10tmean  : num     NA    NA -17.1 ...
 $ l1pm10tmean: num     NA    NA    ...
 $ l2pm10tmean: num     NA    NA    ...
 $ l3pm10tmean: num  NA NA NA NA NA ...
 $ l4pm10tmean: num  NA NA NA NA NA ...
 $ l5pm10tmean: num  NA NA NA NA NA ...
 $ l6pm10tmean: num  NA NA NA NA NA ...
 $ l7pm10tmean: num  NA NA NA NA NA ...
 $ Age2Ind    : num  0 0 0 0 0 0 0 0 0 0 ...
 $ Age3Ind    : num  0 0 0 0 0 0 0 0 0 0 ...
```

For building a multi-city database, the steps above may be avoided by directly using `buildDB`.

As an example, we use `buildDB` with processing function `basicNMMAPS` to build a small four city database that includes New York City, Los Angeles, Chicago, and Seattle. Each of the city dataframes are processed with the `basicNMMAPS` function.

```
> buildDB(procFunc = basicNMMAPS,
+     cityList = c("ny", "la", "chic",
+         "seat"))

Creating directory
    /home/rpeng/R-local/lib/NMMAPSdata/<...>
Creating database: basicNMMAPS
Processing cities...
+ ny ---> /home/rpeng/R-local/lib/<...>
+ la ---> /home/rpeng/R-local/lib/<...>
+ chic ---> /home/rpeng/R-local/lib/<...>
+ seat ---> /home/rpeng/R-local/lib/<...>
Saving city information
Registering database location:
    /home/rpeng/R-local/lib/NMMAPSdata/<...>
```

```
> showDB()

basicNMMAPS in
    /home/rpeng/R-local/lib/NMMAPSdata/db
```

The database created with a given processing function need only be built once for each city. When `buildDB` is finished building the database it automatically calls `registerDB` to make the newly built database the currently registered one and therefore ready for analysis. To use a database for subsequent analyses not immediately following its creation, the database need only be registered using `registerDB`.

`buildDB` returns (invisibly) an object of class `NMMAPSdbInfo` which has a `show` method. This object is also stored with the database and can be retrieved with the `getDBInfo` function.

```
> getDBInfo()

NMMAPS Database with cities:
  ny la chic seat

Call:
buildDB(procFunc = basicNMMAPS,
    cityList = c("ny", "la", "chic",
    "seat"))
```

The `NMMAPSdbInfo` object currently contains slots for the processing function, the list of cities included in the database, the full path to the database, the environment of the processing function, and the original call to `buildDB`. A character vector containing the abbreviated names of the cities included in the new database can be retrieved with the `listDBCities` function. `listDBCities` always lists the names of the cities in the currently registered database.

```
> listDBCities()

[1] "chic" "la"   "ny"   "seat"
```

The file `simple.R` contains the code for fitting model (1) and can be downloaded from the IHAPSS website or sourced directly:

```
> source("http://www.ihapss.jhsph.edu/data/
          NMMAPS/R/scripts/simple.R")
```

It contains a function `fitSingleCity` which can be used for fitting NMMAPS-style time series models to air pollution and mortality data. There are number of arguments to `fitSingleCity`; the default values fit model (1) to a city dataframe.

```
> registerDB("basicNMMAPS")
> loadCity("la")
> fit <- fitSingleCity(data = la,
+     pollutant = "l1pm10tmean",
+     cause = "death")
```

One can examine the formula for `fit` to see the exact model fit to the data by `fitSingleCity`.

```
> formula(fit)

death ~ dow + agecat + ns(time, 98) +
    I(ns(time, 15) * Age2Ind) +
    I(ns(time, 15) * Age3Ind) +
    ns(tmpd, 6) + ns(rmtmpd, 6) +
    ns(dptp, 3) + ns(rmdptp, 3) +
    l1pm10tmean
```

The primary difference between using `fitSingleCity` and calling `glm` directly is that `fitSingleCity` will adjust the number of degrees of freedom for the smooth function of time if there are large contiguous blocks of missing data

The full summary output from the model fit is lengthy, but we can examine the estimate of the pollution effect (and its standard error) via:

```
> summary(fit)$coefficients["l1pm10tmean",
+     ]

    Estimate   Std. Error      t value
0.0003722357 0.0001874975 1.9852832959
    Pr(>|t|)
0.0472094754
```

The estimated effect is 0.0003722, which can be interpreted as approximately a 0.37% increase in mortality with a 10 $\mu$g/m$^3$ increase in PM$_{10}$.

For a single city analysis, returning the entire `glm` object from `fitSingleCity` is not too burdensome with respect to memory usage. However, in a multi-city analysis, with possibly up to 100 cities, it may not be desirable to store 100 `glm` objects at once, each of which can be 10–20 MB large. The function `fitSingleCity` has an argument `extractors`, which by default is `NULL`. One can pass a list of hook functions via the `extractors` argument and these functions will be applied to the object returned from the call to `glm`. This way, one can obtain relevant quantities (coefficients, standard errors, etc.) from the model fit and discard the rest.

```
> extractFun <-
+     list(coef = function(x)
+        summary(x)$coeff["l1pm10tmean",1],
+          std = function(x)
+        summary(x)$coeff["l1pm10tmean",2])
> fit <- fitSingleCity(data = la,
+     pollutant = "l1pm10tmean",
+     cause = "death",
+     extractors = extractFun)
> fit

$coef
[1] 0.0003722357

$std
[1] 0.0001874975
```

We can now run our multi-city analysis by calling `cityApply` with `fitSingleCity` and the list of extractor functions in `extractFun`.

```
> results <- cityApply(fitSingleCity,
+     extractors = extractFun)
```

By default, `cityApply` applies the function specified in the `FUN` argument on all of the city dataframes in the currently registered database.

The effect estimates from the 4 cities can be pooled using a simple fixed effects model:

```
> beta <- sapply(results, "[[", "coef")
> std <- sapply(results, "[[", "std")
> weighted.mean(beta, 1/std^2) *
+     1000

[1] 0.2005406

> sqrt(1/sum(1/std^2)) * 1000

[1] 0.07230552
```

## Future Directions

The NMMAPSdata package is a data package and we purposely omit any code for time series modeling. We are currently developing a separate package specifically designed for fitting time series models to air pollution and health data. For now, we hope that users will find the NMMAPSdata package useful for either reproducing results from previous studies or for implementing their own methods. Comments and suggestions are welcome.

## Bibliography

M. J. Daniels, F. Dominici, S. L. Zeger, and J. M. Samet. *The National Morbidity, Mortality, and Air Pollution Study, Part III: Concentration-Response Curves and Thresholds for the 20 Largest US Cities*. Health Effects Institute, Cambridge MA, 2004. 10

F. Dominici, M. Daniels, S. L. Zeger, and J. M. Samet. Air pollution and mortality: Estimating regional and national dose-response relationships. *Journal of the American Statistical Association*, 97:100–111, 2002a. 11

F. Dominici, A. McDermott, M. Daniels, S. L. Zeger, and J. M. Samet. Mortality among residents of 90 cities. In *Revised Analyses of Time-Series Studies of Air Pollution and Health*, pages 9–24. The Health Effects Institute, Cambridge MA, 2003. 10, 11

F. Dominici, A. McDermott, S. L. Zeger, and J. M. Samet. On the use of generalized additive models in time-series studies of air pollution and health. *American Journal of Epidemiology*, 156(3):193–203, 2002b. 11

R. D. Peng, L. J. Welty, and A. McDermott. The National Morbidity, Mortality, and Air Pollution Study database in R. Technical Report 44, Johns Hopkins University Department of Biostatistics, 2004. `http://www.bepress.com/jhubiostat/paper44/`. 10

J. M. Samet, F. Dominici, S. L. Zeger, J. Schwartz, and D. W. Dockery. *The National Morbidity, Mortality, and Air Pollution Study, Part I: Methods and Methodological Issues*. Health Effects Institute, Cambridge MA, 2000a. 10

J. M. Samet, S. L. Zeger, F. Dominici, F. Curriero, I. Coursac, D. W. Dockery, J. Schwartz, and A. Zanobetti. *The National Morbidity, Mortality, and Air Pollution Study, Part II: Morbidity and Mortality from Air Pollution in the United States*. Health Effects Institute, Cambridge MA., 2000b. 10

# Laying Out Pathways With Rgraphviz

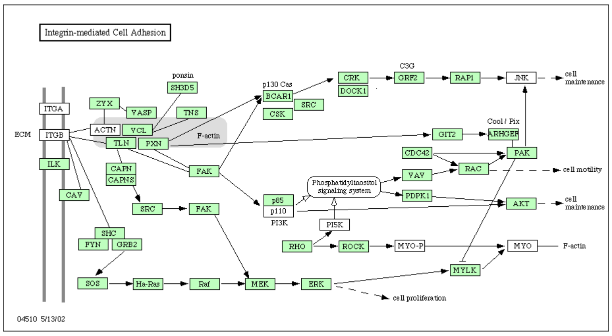
*by Jeff Gentry, Vincent Carey, Emden Gansner and Robert Gentleman*

## Overview

Graphviz `http://www.graphviz.org` is a flexible tool for laying out and rendering graphs. We have developed an R interface to the Graphviz functionality. In this article we demonstrate the use of *Rgraphviz* to layout molecular pathways, but note that the tool is much more general and can be used to layout any graph.

In this article, we will use the *hsa041510* pathway from KEGG (`http://www.genome.ad.jp/kegg/pathway/hsa/hsa04510.html`), which is available as a *graph* object from the *graph* package as the *integrinMediatedCellAdhesion* dataset. This dataset contains the graph as well as a list of attributes that can be used for plotting. The pathway graph as rendered by KEGG is seen here:

 ISSN 1609-3631




# Obtaining the initial graph

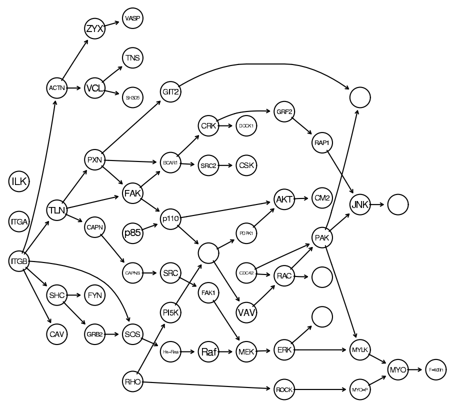
At this time, there is no automated way to extract the appropriate information from KEGG (or other sites) and construct a graph. If one wishes to layout their own pathways, it requires manual construction of a graph, creating each node and then recording the edges. Likewise, for any basic attributes (such as the green/white coloration in the hsa041510 graph), they too must be collected by hand. For instance, this would be a good time to take advantage of edge weights by putting in desired values (which can be changed later, if necessary) while constructing the edges of the graph. We have manipulated some of the weights, such as the weight between the p85 and p110 nodes, as they are intended to be directly next to each other. Once constructed, the graph can be saved with the `save` command and stored for later use (which has been done already as part of the *integrinMediatedCellAdhesion* dataset).

```
> library("Rgraphviz")

Loading required package: graph
Loading required package: cluster
Loading required package: Ruuid
Creating a new generic function for "print" in
"Ruuid"
Loading required package: Biobase
Welcome to Bioconductor
      Vignettes contain introductory material.
      To view, simply type: openVignette()
      For details on reading vignettes, see
      the openVignette help page.

> data("integrinMediatedCellAdhesion")
> IMCAGraph

A graph with  directed  edges
Number of Nodes = 55
Number of Edges = 62
```

# Laying out the graph

Laying out a pathway graph is much like dealing with any other graph, except that typically we want to as closely emulate the officially laid out graph (or at least make it look like an actual pathway - the Graphviz layout methods were not designed with this task in mind). A lot of experimentation comes into play, in order to find the right combination of attributes, although there are some general tips that can help out. The first thing to know is that we will almost always want to use the *dot* layout, as that will provide the closest base to work off. Likewise, the *rankdir* attribute should be set to *LR*, to give us the left to right look of the graph. To see our starting point, here is the *IMCAGraph* with just those settings.

```
> plot(IMCAGraph,
+       attrs = list(graph =
+          list(rankdir = "LR")))
```




Note that *IMCAAttrs$defAttrs* is simply the *rankdir* attribute for *graph*, so we will be using that in place of the `list` call from now on.

This plot is not terrible, it does convey the proper information, but the layout is quite different from the layout at KEGG, and can be difficult to interpret. Furthermore, smaller things like the coloration of the nodes and the shape of the phosphatidylinositol signaling system are not handled.

Using other attributes can have a positive effect. We can set the color of each node (this must be entered manually) and change the shape of the phosphyatidylinositol signaling system node to be an ellipse. We have done this for this graph in the *IMCAAttrs$nodeAttrs* data:

```
> IMCAAttrs$nodeAttrs$shape

Phosphatidylinositol signaling system
                            "ellipse"

> IMCAAttrs$nodeAttrs$fillcolor[1:10]
```

 ISSN 1609-3631

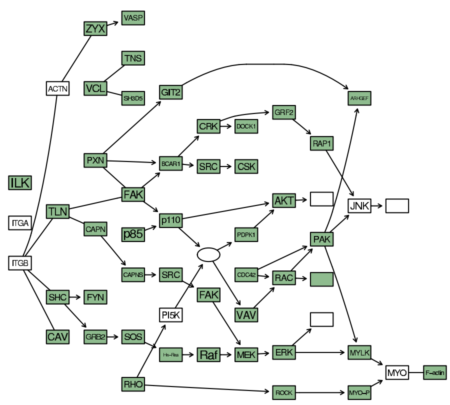
```
  ITGB                                         ITGA
  "white"                                      "white"
  ACTN                                         JNK
  "white"                                      "white"
  MYO                                          MYOP
  "white"                                      "white"
  PI5K  Phosphatidylinositol signaling system
  "white"                                      "white"
  cell maintenance                             CM2
  "white"                                      "white"
```

We have set up a few other attributes. You'll notice on the original plot that there are some nodes that have the same label, there are two *cell maintenance* nodes, 2 *FAK* nodes, and 2 *SRC* nodes. In the internal structure of the graph we have given these nodes different names but we set their labels to be the same as the original. Also, we have defined some edges that do not exist in the original graph for structural reasons and make their color transparent so that they are not displayed. We also change some of the arrowheads:

```
> IMCAAttrs$nodeAttrs$label

                CM2                  FAK1
"cell maintenance"                  "FAK"
               SRC2
              "SRC"

> IMCAAttrs$edgeAttrs$color

     ITGB~SOS       ACTN~VCL        TLN~PXN
"transparent" "transparent" "transparent"

> IMCAAttrs$edgeAttrs$arrowhead

 ITGB~ACTN   ITGB~TLN   ACTN~ZYX    VCL~TNS
    "none"     "none"     "none"     "none"
 VCL~SH3D5   TLN~CAPN    TLN~FAK  PAK~ARGHEF
    "none"     "none"     "none"     "none"
   PXN~FAK   ITGB~CAV   ITGB~SHC MYO~F-actin
    "none"     "none"     "none"     "none"
```

Using these attributes to plot the graph will get us a bit closer to our goal:

```
> plot(IMCAGraph, attrs = IMCAAttrs$defAttrs,
+      nodeAttrs = IMCAAttrs$nodeAttrs,
+      edgeAttrs = IMCAAttrs$edgeAttrs)
```




Now the color scheme is the same as KEGG and using an ellipse helps with the rendering of the phosphatidylinositol signaling system node. However, we're still left with the issue that the layout itself is not the same as the original and is harder to interpret. The output nodes are scattered, there is no clear sense of where the membrane nodes are, and many nodes that are intended to be close to each other are not. This is where the use of subgraphs and clusters can help. In Graphviz, a subgraph is an organizational method to note that a set of nodes and edges belong in the same conceptual space, and share attributes. Subgraphs have no impact on layouts themselves, but are used to group elements of the graph for assigning attributes. A Graphviz cluster is a subgraph which is laid out as a separate graph and then introduced into the main graph. This provides a mechanism for clustering the nodes in the layout. For a description of how to specify subgraphs in *Rgraphviz*, please see the vignette `HowTo Render A Graph Using Rgraphviz` from the *Rgraphviz* package.

Here we define four subgraphs: One will be the membrane nodes, one will be the output nodes, one will be the F-actin block and the last will be the combination of the *PAK*, *JNK* and *ARHGEF* nodes to get the verticle stacking. It would be possible to specify more subgraphs to try to help keep things more blocked together like the original graph, but for the purposes of this document, these are what will be used.

```
> sg1 <- subGraph(c("ILK", "ITGA",
                  "ITGB"), IMCAGraph)
> sg2 <- subGraph(c("cell maintenance",
+                 "cell motility",
+                 "cell proliferation",
+                 "F-actin"), IMCAGraph)
> sg3 <- subGraph(c("ACTN", "VCL", "TLN",
+                "PXN"), IMCAGraph)
```

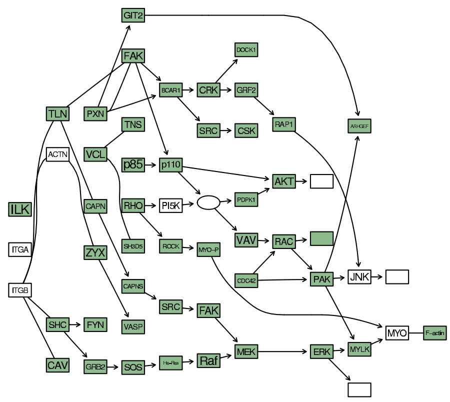
```
> sg4 <- subGraph(c("PAK", "JNK", "ARHGEF"),
+                 IMCAGraph)
```

We have defined the subgraphs. We can use these as subgraphs or clusters in Graphviz. Ideally, we would like to use clusters, as that guarantees that the nodes will be laid out close together. However, we want to use the *rank* attribute for the membrane, output nodes and the *ARHGEF* block, specifically using the values *min* and *max* and *same*, respectively. That will help with the verticle alignment that we see in the KEGG graph and create more of the left to right orientation. The problem is that *rank* currently only works with subgraphs and not clusters. So for these three subgraphs, we will be defining them as Graphviz subgraphs, and the F-actin block will be defined as a cluster. We have already prepared all of this as *IMCAAttrs$subGList*:

```
> IMCAAttrs$subGList

[[1]]
[[1]]$graph
A graph with  undirected  edges
Number of Nodes = 3
Number of Edges = 0

[[1]]$cluster
[1] FALSE

[[1]]$attrs
[[1]]$attrs$rank
[1] "min"

[[2]]
[[2]]$graph
A graph with  undirected  edges
Number of Nodes = 4
Number of Edges = 0

[[2]]$cluster
[1] FALSE

[[2]]$attrs
[[2]]$attrs$rank
[1] "max"

[[3]]
[[3]]$graph
A graph with  undirected  edges
Number of Nodes = 4
Number of Edges = 1

[[4]]
[[4]]$graph
A graph with  undirected  edges
Number of Nodes = 3
Number of Edges = 1

[[4]]$cluster
[1] FALSE

[[4]]$attrs
[[4]]$attrs$rank
[1] "same"
```

You can see that we have set the *rank* attribute on the three subgraphs and that the F-actin subgraph has been defined as a cluster. Using this subgraph list, we now get:

```
> plot(IMCAGraph, attrs = IMCAAttrs$defAttrs,
+      nodeAttrs = IMCAAttrs$nodeAttrs,
+      edgeAttrs = IMCAAttrs$edgeAttrs,
+      subGList = IMCAAttrs$subGList)
```

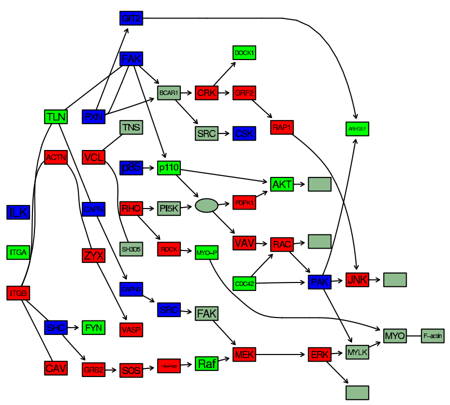



While this is still not identical to the image on KEGG (and for most graphs, it will be hard to do so), this layout is now easier to interpret. We can see the output nodes are now to the right side of the graph, and the membrane nodes are stacked on the left of the graph. We can also see the F-actin group in the upper left portion of the graph, representing the cluster.

## Working with the layout

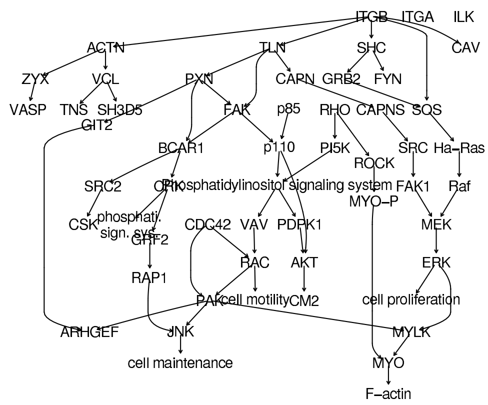
One of the benefits of using *Rgraphviz* to perform your layout as opposed to using the static layouts provided by sites like KEGG, is the ability to work with outside data and visualize it using your graph. The `plotExpressionGraph` function in *geneplotter* can be used to take expression data and then color nodes based on the level of expression. By default, this function will color nodes blue, green or red, corresponding to expression levels of 0-100, 101-500, and 501+ respectively. Here we will use this function along with the *fibroEset* and *hgu95av2* data packages and the *IMCAAttrs$IMCALocuLink* data which maps the nodes to their LocusLink ID values.

```
> require("geneplotter")

Loading required package: geneplotter
Loading required package: annotate
Loading required package: reposTools
[1] TRUE

> require("fibroEset")

Loading required package: fibroEset
[1] TRUE

> require("hgu95av2")

Loading required package: hgu95av2
[1] TRUE

> data("fibroEset")
> plotExpressionGraph(IMCAGraph,
+                     IMCAAttrs$LocusLink,
+                     exprs(fibroEset)[,1],
+                     hgu95av2LOCUSID,
+                     attrs =
+                     IMCAAttrs$defAttrs,
+                     subGList =
+                     IMCAAttrs$subGList,
+                     nodeAttrs =
+                     IMCAAttrs$nodeAttrs,
+                     edgeAttrs =
+                     IMCAAttrs$edgeAttrs)
```



One can also simply choose to layout the pathway based on the needs and desires of a particular situation. For instance, the following layout could be used in situations where the node names are the important visual cue, as opposed to the previous example where the nodes themselves are being used to demonstrate values:

```
> z <- IMCAGraph
> nodes(z)[39] <- c("phosphati.\nsign. sys.")
> nag <- agopen(z, name = "nag",
+               attrs = list(node =
+               list(color = "white", fontcolor =
+               "white"), edge =
+               list(arrowsize = 2.8, minlen = 3)))
> nagxy <- getNodeXY(nag)

> plot(nag)
> text(nagxy, label = nodes(z), cex = 0.8)
```




## Conclusions

Rgraphviz provides a flexible interface to Graphviz to obtain layout information. Rendering the graph is handled in R, using R graphics. There are still a few rough edges but the package is quite flexible and can be used to layout and render any graph. Graph layout is still very much an art. You will seldom get a good layout without doing some tweaking and adjusting. We have demonstrated a few of the very many tools that Graphviz offers.

*Jeff Gentry*
*DFCI*
jgentry@jimmy.harvard.edu

*Vincent Carey*
*Channing Lab*
stvjc@channing.harvard.edu

*Emden Gansner*
*AT&T Labs*
erg@research.att.com

*Robert Gentleman*
*DFCI*
rgentlem@jimmy.harvard.edu

# Fusing R and BUGS through Wine

**An Introduction to Package** rbugs

*by Jun Yan*

## Historical Review

BUGS (Spiegelhalter et al., 1996) has been a very successful statistical software project. It is widely used by researchers in many disciplines as a convenient tool for doing Bayesian statistical analysis. The BUGS developers have shifted developmental efforts to WinBUGS, and the classic BUGS is not being developed further. The current release WinBUGS 1.4 comes with a scripting facility which permits batch running, and also therefore the ability to run WinBUGS from other programs.

Fusing R and BUGS has been available from several sources. On Windows systems, it dates back to Kennith Rice's package EmBedBUGS (Rice, 2002). EmBedBUGS is available in an Windows self-extracting archive and is not in the standard format of an R package. First version adapted from EmBedBUGS, Andrew Gelman's collection of functions bugs.R (Gelman, 2004) has evolved into a comprehensive tool, which can run WinBUGS 1.4 from R, collect the MCMC samples, and perform basic output analysis. This collection was recently packaged by Sturtz and Ligges (2004) as R2WinBUGS, with some tools (including print and plot methods) for output analysis slightly changed from Andrew Gelman's original function.

On Linux/Unix systems, however, less work has appeared since the pause of the support for classic BUGS. Plummer (2003) reported experience and some known problems of running WinBUGS under Wine, "an Open Source implementation of the Windows API on top of X and Unix" (Wine, 2004). With Wine, it is possible to provide a facility of fusing R and BUGS on a Linux system similar to what's available on a Windows system. This is what package rbugs aims at.

## Design

The powerfulness of BUGS (In the sequel, I use BUGS instead of WinBUGS when there is no confusion in the context, hoping some day classic BUGS will be supported.) lies in that, with a straightforward syntax for model specification, it provides a universal MCMC sampler of posterior distributions for rather complicated problems. Users do not need to worry about how the MCMC samples are actually drawn. The design philosophy of rbugs, therefore, is to take the advantage of the universal MCMC sampler of BUGS through an interface as simple as possible, and return the MCMC samples in a format which can be fed into other R packages specializing in Bayesian output analysis, such as boa (Smith, 2004) and coda (Plummer et al., 1996). In addition, users (particularly those who are uncomfortable with point-and-click) enjoy accesses to various files generated during the preparation of running BUGS in batch-mode.

Compared to package R2WinBUGS, rbugs is different in the following sense: 1) It does not provide Bayesian output analysis and only serves as a fuse to connect R and BUGS; 2) It provides access to automating the preparation of script file, model file, data file, and initial value file, which are needed by running BUGS in batch-mode; and 3) Its main target users are Linux users having access to Wine.

## Configuration

Package rbugs has been tested on both Linux and Windows. After installation, it's worth setting two environment variables in the '.Renviron' file to save some typing: BUGS and WINE. These two variables store the full name of the executables of BUGS and Wine, respectively. They are used as the default values in function rbugs. The following is an example on my machine:

```
BUGS="c:/program files/winbugs14/winbugs14.exe"
WINE="/var/scratch/jyan/wine-20040408/wine"
```

The definition of WINE is only necessary if BUGS is to be used via Wine. In that case, the wine configuration in './wine/config' in the home directory will be processed by an internal function to create a map from the Windows drives to the native directories. Further discussion about the usage via wine is presented next.

## Run BUGS in a Single Call

To run BUGS in batch-mode, a minimum of four files are needed as input: a script file, a model file, a data file, and an initial value file for each chain to be run. Except the model file, other three types of files can be generated. The model file would need to be written by a user outside of R. The output from BUGS are saved in files specified in the script file, and can be read into R and used for convergence and output analysis. From sufficient information collected from its arguments, such as the data list, parameters to be monitored, number of chains, etc., function rbugs generates the data file, initial value files, and script file that are needed, calls BUGS through an OS-specific system call, and returns the MCMC output as a list of matrix. The returned object can be further processed by packages boa and coda, taking the advantage of various native analysis available in R. An example is provided with the pumps data in the Example Volume I of BUGS:

```
> ? pumps
```

## Experience with Wine

On a RedHat 3.0 workstation, I experimented running WinBUGS via Wine 20040408. As reported by Plummer (2003), buttons still don't respond to clicks. One would have to use the return key after pointing to a button. This may not be a problem for people who do not like using a mouse anyway. Fortunately, the batch-mode works fine and the results from some examples I tried are the same as those obtained from a Windows system.

Installation guide for Wine can be found from its website. For people who don't have root access, it's sufficient to just compile it and set `WINE` as the full name of the Wine executable in the compiling directory. The compiling is straightforward.

When using `rbugs`, one needs to pay attention to the difference in two of its arguments: `workingDir` and `bugsWorkingDir`. On a Windows system, they should be the same. But on a Linux system, `workingDir` refers to the directory that's recognizable by native operations, while `bugsWorkingDir` refers to the same directory as `workingDir` but translated to a Windows directory recognizable by WinBUGS via Wine. For example, on my system, drive C is defined in the Wine configuration:

```
[Drive C]
"Path" = "/var/scratch/jyan/c"
```

If I would like to use

```
bugsWorkingDir="c:/tmp",
```

then I would need to have

```
workingDir="/var/scratch/jyan/c/tmp".
```

With these straightened out, the pumps example can be run on a Linux system. Same as on a Windows system, `rbugs` will launch BUGS. In the current release of `rbugs`, the debug information from Wine is redirected to a temporary file and deleted on exit.

The configuration information of wine is usually stored in '.wine/config' in the home directory. An internal function processes this config file and stores the drive mapping between Windows and the native Linux system in a internal data frame `.DriveTable`. When using `rbugs`, if `workingDir` is the default, `NULL`, then `bugsWorkingDir` is translated using the drive mapping table.

## Preparing Files for BUGS Batch-mode

Often times, one would like to use a call of `rbugs` to do some exploration, checking if the model and the data compile fine in BUGS. It would be useful to have the generated script file, data file, and initial value files available for using BUGS directly in other circumstances. There is no difficulty in generating the script files. But for the data file and the initial value files, the format of the data becomes an important issue. In the WinBUGS 1.4 manual, under the section of Model Specification, formatting of data is discussed. It reads that BUGS can take read files created from the S-Plus `dput` function. Unfortunately, this is not (or no longer) true for both the most recent versions of S-Plus and R. Let's look at R-1.9.0 only:

```
> a <- matrix(c(314159265358979, 0.0001,
                -0.0001,          0.05), 2, 2)
> dput(list(a = a), "tmp")
> file.show("tmp")
structure(list(a = structure(c(314159265358979,
1e-04, -1e-04, 0.05), .Dim =
as.integer(c(2, 2)))), .Names = "a")
```

A BUGS user immediately sees that BUGS will complain when it reads this! Besides the extra characters of "as.integer" and ".Names", there are less documented subtle issues: 1) "e" should be "E"; 2) "1e" should be "1.0E"; and 3) the first number exceeded 14 digits.

In not necessarily the most efficient way, the function `format4Bugs` converts the data to characters with `formatC` and then uses a modification of a format data function by Kennith Rice to return hopefully the right format for BUGS.

Using `format4Bugs`, functions `genDataFile` and `genInitsFile` prepare data file and initial value files. Function `genBugsScript` generate a script file. All these files are accessible by users and hence ease the usage of BUGS in other circumstances.

## Remarks

Since Wine is built upon X windows, WinBUGS would not run from an ssh terminal without X windows support. Many people had wished the classic BUGS were supported. That not happening soon, it would be desirable to have a better supported command line interface of WinBUGS, so that launching the GUI becomes an options.

As an infrequent Windows user, I am more oriented to experimenting and supporting fusing R and BUGS through Wine on Linux systems. Windows users are referred to package `R2WinBUGS`. A forthcoming paper by the package authors will provide detailed demonstration and become a standard reference.

Plummer (2004) just released 0.50 of JAGS. Quote from Martyn Plummer: "JAGS is Just Another Gibbs Sampler - an alternative engine for the BUGS language that aims for the same functionality as classic BUGS. JAGS is written in C++ and licensed under the GNU GPL. It was developed on Linux and also runs on Windows." The functions in package `rbugs` can also be used to prepare files for JAGS. I am looking forward to seeing the growth of JAGS.

I also tried using R for Windows through Wine. It worked last winter with Wine 20031016, but is not working with Wine 20040408 now. Unfortunately, since my Wine 20040408 was compiled after my system has been recently upgraded to Red Hat Workstation 3.0, I cannot tell which change has caused it.

## Bibliography


Gelman, A. (2004), "bugs.R: functions for running WinBugs from R," http://www.stat.columbia.edu/~gelman/bugsR/. 19

Plummer, M. (2003), "Using WinBUGS under Wine," http://calvin.iarc.fr/bugs/wine/. 19, 20

Plummer, M. (2004), "JAGS version 0.50 manual," http://www-fis.iarc.fr/~martyn/software/jags/. 20

Plummer, M., Best, N., Cowles, K., and Vines, K. (1996), "coda: Output analysis and diagnostics for MCMC," http://www-fis.iarc.fr/coda/. 19

Rice, K. (2002), "EmBedBUGS: An R package and S library," http://www.mrc-bsu.cam.ac.uk/personal/ken/embed.html. 19

Smith, B. (2004), "boa: Bayesian Output Analysis Program for MCMC," http://www.public-health.uiowa.edu/boa. 19

Spiegelhalter, D. J., Thomas, A., Best, N. G., and Gilks, W. (1996), *BUGS: Bayesian inference Using Gibbs Sampling, Version 0.5, (version ii)* http://www.mrc-bsu.cam.ac.uk/bugs. 19

Sturtz, S. and Ligges, U. (2004), "R2WinBUGS: Running WinBUGS from R," http://cran.r-project.org/src/contrib/Descriptions/R2WinBUGS.html. 19

Wine (2004), "Wine," http://www.winehq.org. 19



*Jun Yan*
*University of Iowa, U.S.A.*
`jyan@stat.uiowa.edu`


# R Package Maintenance

*Paul Gilbert*

## Introduction

Quality control (QC) for R packages was the feature that finally convinced me to maintain R packages and also run them in S, rather than the reverse. A good QC system is essential in order to contain the time demands of maintaining many packages with interdependencies. It is necessary to have quick, easy, reliable ways to catch problems. This article explains how to use the R package QC features (in the "tools" package by Kurt Hornik and Friedrich Leisch) for ongoing maintenance and development, not just as a final check before submitting a package to CRAN. This should be of interest to individuals or organizations that maintain a fairly large code base, for their own use or the use of others.

The main QC features for an R package check that:

- code in package directory R/ is syntactically correct
- code in package directory tests/ runs and does not crash or stop()
- documentation is complete and accurate in several respects
- examples in the documentation actually run
- code in package directory demo/ runs
- vignettes in package directory inst/doc/ run

These provide several important features for package maintenance. Developers like to improve code, but documentation updates are often neglected. A simple method to identify necessary documentation changes means documentation maintenance is (almost) painless. The QC tools can be used to help flag when documentation changes are necessary. They also ensure that packaged code can be quickly tested to ensure it works with a new version of R (or a new compiler, or a new operating system, or a new computer). The system explained below also helps check dependencies among functions in different packages, easing development by quickly identifying changes that break code in other packages.

The system described here uses the QC features in R in conjunction with the *make* utility. It checks code and documentation of multiple packages, automatically when changes to source files imply that these checks need to be done. The key is a good 'Makefile' with interdependencies properly identified. It should be possible to run this system with a relatively small investment in "local setup" for a different set of packages, perhaps only a couple of hours. This presumes a certain familiarity with *make*. For a complicated set of package, a somewhat larger amount of time may be necessary in order to understand interdependencies among packages. If your packages are not well organized then a much larger time investment will be necessary, but well worthwhile.

## Make

This is not a tutorial about *make*, but a rudimentary explanation is given in order to make the remainder of the artical accessible to a wider audience. Briefly, the *make* utility uses targets (rules) which may have prerequisites (other targets or files). These are indicated in a file typically called 'Makefile'. This works most easily when a target is the name of a file generated from another file, for example, a compiled target file called foo generated from a C code prerequisite file called foo.c. Make determines that a target is out of date and must be re-generated if the file timestamp for the target is older than the timestamp of any prerequisite. This is recursive, so a target must be re-generated (or "re-made") if it depends on a target, that depends on a target, ..., that depends on a file that is newer. Properly mapping out the dependencies in a 'Makefile' eventually saves an enormous amount of time, because a change in a source file only necessitates re-generating dependent targets. To understand correctly how this is used in the context of R package maintenance, it is important to recognize that "re-made" does not mean simply that code (or documentation) is checked to be syntactically correct, it also means a number of tests are completed to insure it works correctly.

## Make and R Package QC

In order to implement the system for R package maintenance, one critical simplifying assumption is that code testing does not depend on documentation testing. This may seem obvious, but it has the implication that examples in the documentation are not the most important way to catch mistakes in the code. That is, there should be files in the tests/ directory of a package that will generate errors if mistakes are introduced into the code. These would typically run functions, check results against known values, and *stop()* if an error is indicated.

With this simplifying assumption it is possible to distinguish two main targets for each package: "code" and "doc." These are each aliases for several "sub-targets." The code target tests the code in a package. It may be a prerequisite for code in other packages, but the doc target in a package is never a prerequisite in other packages. This means that a change to .Rd files in the man directory, or to .R files in the demo directory, or to vignette files, will signal remaking only for the package itself, and not for other packages. Changes to code files in the R directory or files in the tests directory will signal re-making for the package, and this may imply re-making of other packages that depend on it.

As an example, I have package dse2, which depends on dse1, which depends on packages tframe and setRNG. Changes in files in tframe/R should provoke a remake of dse1 and dse2, but changes in tframe/man or tframe/inst/doc should not provoke a remake of dse1 and dse2.

The 'Makefile' line for some targets uses "R CMD check", but in most cases the targets directly use functions in library("tools"). Shell variables, doc targets, and many code targets, are common to all packages and can be specified in common files, 'Makevars' and 'Makerules', which are included into the 'Makefile' for each package. (For technical reasons it is best to have these in two files rather than one.) The key code sub-target (Rcode) has different prerequisites for each package and must thus be specified in the specific 'Makefile' for each package.

As an example, Figure 1 shows the critical part [1] of the 'Makefile' for my dse1 package, which has the packages tframe and setRNG as a prerequisites:

After first including the common variables from ../Makevars, this specifies the default target prerequisites. (Left of the colon is a target name, right of the colon is the list of prerequisites, backslash indicates line continuation.)

Packages are each in a subdirectory below a common directory, so *../tframe* refers to the relative path from the package dse1 directory to the directory for the package tframe. Some targets, like Rcode, are not naturally files, so to take advantage of the timestamp mechanism used by *make* it is necessary to create an artificial file (placed in a subdirectory referred to by the variable *FLAGS*). The critical part of the macro[2] *RchkCodeMacro* is specified in 'Makevars' in Figure 2

This checks the code using any necessary packages from the location indicated by *CHKLIBS*, which is where packages that have already been checked are installed.

As another example, some of the documentation targets are specified in 'Makerules' (in Figure 3) by

[1]The complete generic makefiles should be available in the contributed section of CRAN.
[2]The *define* feature and some other aspects of these files may be specific to GNU *make*.

```
include ../Makevars

default:    undoc checkDocFiles codoc examples latex demos \
            checkDocStyle  checkFF checkMethods checkReplaceFuns \
            Rcode checkVignettes pdfVignettes tar

Rcode: R/*.R tests/*.R LICENSE DESCRIPTION INDEX \
            ../tframe/$(FLAGS)/Rcode  ../setRNG/$(FLAGS)/Rcode
     ${RchkCodeMacro}

include ../Makerules
```

Figure 1: Makefile for dse1

```
define RchkCodeMacro
     ...
     R_LIBS=$(CHKLIBS)  $(RENV) R CMD check \
          --outdir=$(CURDIR)/$(TMP) --library=$(CURDIR)/$(TMP) \
          --no-vignettes --no-codoc --no-examples \
          --no-latex  $(CURDIR)
     ...
     @touch $(FLAGS)/$@
endef
```

Figure 2: *RchkCodeMacro* is specified in 'Makevars'

```
undoc checkDocFiles checkDocStyle:      man  R/*.R
     @$(MKDIR) $(TMP)
     @echo "library(tools); $@(dir='$(CURDIR)')" | R --vanilla -q >$(TMP)/$@
     @test -z "`grep 'Error' $(TMP)/$@`" || (cat $(TMP)/$@ ; exit 1 )
#       check errors from undoc and checkDocFiles
     @test -z "`grep 'Undocumented' $(TMP)/$@`" || (cat $(TMP)/$@ ; exit 1 )
     @$(MKDIR) $(FLAGS)
     @mv  $(TMP)/$@ $(FLAGS)/$@
```

Figure 3: Documentation targets in 'Makerules'

This specifies the targets undoc, checkDocFiles, and checkDocStyle, which all depend on any files in the man directory, as well as any code files R/*.R. The output from the R sessions that runs *undoc()* and *checkDocFiles()* print errors and warnings, but these do not automatically produce a shell error signal as a flag that *make* recognizes. It is possible to do this using *R* code that determines if the result should indicate an error, and sets *q(status=1)* but that is not done in this example. Instead, a test on a grep of the output is used to determine the shell error status. (This may change in the future.) If the signal does not indicate a failure (exit 1) then the output is moved to the *FLAGS* directory to indicate that the target has completed successfully.

## Summary

There are trade-offs in the way R code is organized into packages. If all code is in one package then there are no package inter-dependencies, but everything must be tested after any change. Faster computers make it possible to consider this, and the *make*/QC system described here would be extra overhead and of limited value in that situation. However, more documentation and examples, along with more extensive test suites, take longer to run, and so encourage a finer breakdown into packages. In addition to this, there are two complementary reasons for organizing functions into packages. One is to limit dependencies, as much as reasonably possible, between groups of functions that are not closely related and may not often be used together. The second is to group together "kernel" functions which are tools used by several other packages. The dependencies among packages must be carefully mapped out, which forces one to think carefully about what is kernel code and what is not. These reasons for organizing code into packages may be even more important in a situation where multiple programmers or users are maintaining packages.

It is important to see that the savings in this *make*/QC system come from a few different aspects. The first is that packages of kernel code used by other packages tend to be more stable and less frequently changed than the packages that use them. If kernel packages are not changed, they do not need to be re-made. The second aspect is that dependencies among packages are in the code, not in the documentation. Thus documentaion changes imply only that the documentation for that particular package needs to be checked. The aspect that results in the most important savings, however, is that the need for many documentation changes are flagged immediately, while you still remember what that marvelous change in the code really did.

## Acknowledgments

I am grateful to Kurt Hornik for many helpful explanations and comments.

*Paul Gilbert,*
*Department of Monetary and Financial Analysis,*
*Bank of Canada,*
*234 Wellington St.,*
*Ottawa, Canada, K1A 0G9*
pgilbert@bank-banque-canada.ca

# Changes in R

*by the R Core Team*

## User-visible changes in 2.0.0

- The stub packages from 1.9.x have been removed: the `library()` function selects the new home for their code.

- 'Lazy loading' of R code has been implemented, and is used for the standard and recommended packages by default. Rather than keep R objects in memory, they are kept in a database on disc and only loaded on first use. This accelerates startup (down to 40% of the time for 1.9.x) and reduces memory usage – the latter is probably unimportant of itself, but reduces commensurately the time spent in garbage collection.

  Packages are by default installed using lazy loading if they have more than 25Kb of R code and did not use a saved image. This can be overridden by `INSTALL --[no-]lazy` or via a field in the `DESCRIPTION` file. Note that as with `--save`, any other packages which are required must be already installed.

  As the lazy-loading databases will be consulted often, R will be slower if run from a slow network-mounted disc.

- All the datasets formerly in packages 'base' and 'stats' have been moved to a new package 'datasets'. `data()` does the appropriate substitution, with a warning. However, calls to `data()` are not normally needed as the data objects are visible in the 'datasets' package.

  Packages can be installed to make their data objects visible via `R CMD INSTALL --lazy-data` or via a field in the `DESCRIPTION` file.

- Package 'graphics' has been split into 'grDevices' (the graphics devices shared between base and grid graphics) and 'graphics' (base graphics). Each of the 'graphics' and 'grid' packages load 'grDevices' when they are attached. Note that `ps.options()` has been moved to grDevices and user hooks may need to be updated.

- The semantics of `data()` have changed (and were incorrectly documented in recent releases) and the function has been moved to package 'utils'. Please read the help page carefully if you use the 'package' or 'lib.loc' arguments.

  `data()` now lists datasets, and not just names which `data()` accepts.

- Dataset 'phones' has been renamed to 'WorldPhones'.

- Datasets 'sunspot.month' and 'sunspot.year' are available separately but not via `data(sunspot)` (which was used by package lattice to retrieve a dataset 'sunspot').

- Packages must have been re-installed for this version, and `library()` will enforce this.

- Package names must now be given exactly in `library()` and `require()`, regardless of whether the underlying file system is case-sensitive or not. So `library(mass)` will not work, even on Windows.

- R no longer accepts associative use of relational operators. That is, `3 < 2 < 1` (which used to evalute as `TRUE`!) now causes a syntax error. If this breaks existing code, just add parentheses — or braces in the case of plotmath.

- The R parser now allows multiline strings, without escaping the newlines with backslashes (the old method still works). Patch by Mark Bravington.

## New features

- There is a new atomic vector type, class `"raw"`. See `?raw` for full details including the operators and utility functions provided.

- The default `barplot()` method by default uses a gamma-corrected grey palette (rather than the heat color palette) for coloring its output when given a matrix.

- The 'formula' method for `boxplot()` has a 'na.action' argument, defaulting to `NULL`. This is mainly useful if the response is a matrix when the previous default of 'na.omit' would omit entire rows. (Related to PR#6846.)

  `boxplot()` and `bxp()` now obey global 'par' settings and also allow the specification of graphical options in more detail, compatibly with S-PLUS (fulfilling wishlist entry PR#6832) thanks to contributions from Arni Magnusson. For consistency, 'boxwex' is not an explicit argument anymore.

- `chull()` has been moved to package 'graphics' (as it uses `xy.coords`).

- There is now a `coef()` method for summaries of "nls" objects.

- `compareVersion()`, `packageDescription()` and `read.00Index()` have been moved to package 'utils'.

- `convolve()`, `fft()`, `mvfft()` and `nextn()` have been moved to package 'stats'.

- `coplot()` now makes use of 'cex.lab' and 'font.lab' `par()` settings.

- `cumsum/prod/max/min()` now preserve names.

- `data()`, `.path.packages()` and `.find.packages()` now interpret `package = NULL` to mean all loaded packages.

- `data.frame()` and its replacement methods remove the names from vector columns. Using `I()` will ensure that names are preserved.

- `data.frame(check.names = TRUE)` (the default) enforces unique names, as S does.

- `.Defunct()` now has 'new' and 'package' arguments like those of `.Deprecated()`.

- The `plot()` method for "dendrogram" objects now respects many more nodePar and edgePar settings and for edge labeling computes the extents of the diamond more correctly.

- `deparse()`, `dput()` and `dump()` have a new 'control' argument to control the level of detail when deparsing. `dump()` defaults to the most detail, the others default to less. See `?.deparseOpts` for the details.

  They now evaluate promises by default: see `?dump` for details.

- `dir.create()` now expands ~ in filenames.

- `download.file()` has a new progress meter (under Unix) if the length of the file is known — it uses 50 equals signs.

- dyn.load() and library.dynam() return an object describing the DLL that was loaded. For packages with namespaces, the DLL objects are stored in a list within the namespace.

- New function `eapply()`: apply for environments. The supplied function is applied to each element of the environment; the order of application and the order of the results are not specified.

- `edit()` and `fix()` use the object name in the window caption on some platforms (e.g. Windows).

- Function `file.edit()` function added: like `file.show()`, but allows editing.

- Function `file.info()` can return file sizes > 2Gb if the underlying OS supports such.

- `fisher.test(*, conf.int=FALSE)` allows the confidence interval computation to be skipped.

- `formula()` methods for classes "lm" and "glm" used the expanded formula (with '.' expanded) from the terms component.

- The 'formula' method for `ftable()` now looks for variables in the environment of the formula before the usual search path.

- A new function `getDLLRegisteredRoutines()` returns information about the routines available from a DLL that were explicitly registered with R's dynamic loading facilities.

- A new function `getLoadedDLLs()` returns information about the DLLs that are currently loaded within this session.

- The package element returned by `getNativeSymbolInfo()` contains reference to both the internal object used to resolve symbols with the DLL, and the internal DllInfo structure used to represent the DLL within R.

- `help()` now returns information about available documentation for a given topic, and notifies about multiple matches. It has a separate `print()` method.

  If the latex help files were not installed, `help()` will offer to create a latex file on-the-fly from the installed .Rd file.

- `heatmap()` has a new argument 'reorderfun'.

- Most versions of `install.packages()` have a new optional argument `dependencies = TRUE` which will not only fetch the packages but also their uninstalled dependencies and their dependencies.

  The Unix version of `install.packages()` attempts to install packages in an order that reflects their dependencies. (This is not needed for binary installs as used under Windows.)

- `interaction()` has new argument 'sep'.

- `interaction.plot()` allows `type = "b"` and doesn't give spurious warnings when passed a `matplot()`-only argument such as 'main'.

- `is.integer()` and `is.numeric()` always return `FALSE` for a factor. (Previously they were true and false respectively for well-formed factors, but it is possible to create factors with non-integer codes by underhand means.)

- New functions `is.leaf()`, `dendrapply()` and a `labels()` method for dendrogram objects.

- `legend()` has an argument 'pt.lwd' and setting 'density' now works because 'angle' now defaults to 45 (mostly contributed by Uwe Ligges).

- `library()` now checks the version dependence (if any) of required packages mentioned in the `Depends:` field of the `DESCRIPTION` file.

- `load()` now detects and gives a warning (rather than an error) for empty input, and tries to detect (but not correct) files which have had LF replaced by CR.

- `ls.str()` and `lsf.str()` now return an object of class `ls_str` which has a print method.

- `make.names()` has a new argument `allow_`, which if false allows its behaviour in R 1.8.1 to be reproduced.

- The 'formula' method for `mosaicplot()` has a 'na.action' argument defaulting to 'na.omit'.

- `model.frame()` now warns if it is given `data = newdata` and it creates a model frame with a different number of rows from that implied by the size of 'newdata'.

  Time series attributes are never copied to variables in the model frame unless `na.action = NULL`. (This was always the intention, but they sometimes were as the result of an earlier bug fix.)

- There is a new 'padj' argument to `mtext()` and `axis()`. Code patch provided by Uwe Ligges (fixes PR#1659 and PR#7188).

- Function `package.dependencies()` has been moved to package 'tools'.

- The 'formula' method for `pairs()` has a 'na.action' argument, defaulting to 'na.pass', rather than the value of `getOption("na.action")`.

- There are five new `par()` settings:

  'family' can be used to specify a font family for graphics text. This is a device-independent family specification which gets mapped by the graphics device to a device-specific font specification (see, for example, `postscriptFonts()`). Currently, only PostScript, PDF, X11, Quartz, and Windows respond to this setting.

  'lend', 'ljoin', and 'lmitre' control the cap style and join style for drawing lines (only noticeable on thick lines or borders). Currently, only PostScript, PDF, X11, and Quartz respond to these settings.

  'lheight' is a multiplier used in determining the vertical spacing of multi-line text.

  All of these settings are currently only available via `par()` (i.e., not in-line as arguments to `plot()`, `lines()`, ...)

- PCRE (as used by grep etc) has been updated to version 5.0.

- A 'version' argument has been added to `pdf()` device. If this is set to "1.4", the device will support transparent colours.

- `plot.xy()`, the workhorse function of points, lines and plot.default now has 'lwd' as explicit argument instead of implicitly in `"..."`, and now recycles 'lwd' where it makes sense, i.e. for line based plot symbols.

- The `png()` and `jpeg()` devices (and the `bmp()` device under Windows) now allow a nominal resolution to be recorded in the file.

- New functions to control mapping from device-independent graphics font family to device-specific family: `postscriptFont()` and `postscriptFonts()` (for `postscript()` and `pdf()`); `X11Font()` and `X11Fonts()`; `windowsFont()` and `windowsFonts()`; `quartzFont()` and `quartzFonts()`.

- power (`x^y`) has optimised code for y = 2.

- `prcomp()` is now generic, with a formula method (based on an idea of Jari Oksanen).

  `prcomp()` now has a simple `predict()` method.

- `printCoefmat()` has a new logical argument 'signif.legend'.

- `quantile()` has the option of several methods described in Hyndman and Fan (1996). (Contributed by Rob Hyndman.)

- `rank()` has two new 'ties.method's, "min" and "max".

- New function `read.fortran()` reads Fortran-style fixed-format specifications.

- `read.fwf()` reads multiline records, is faster for large files.

- `read.table()` now accepts `"NULL"`, `"factor"`, `"Date"` and `"POSIXct"` as possible values of colClasses, and colClasses can be a named character vector.

- `readChar()` can now read strings with embedded nuls.

- The "dendrogram" method for `reorder()` now has a 'agglo.FUN' argument for specification of a weights agglomeration function.

- New `reorder()` method for factors, slightly extending that in lattice. Contributed by Deepayan Sarkar.

- Replaying a plot (with `replayPlot()` or via autoprinting) now automagically opens a device if none is open.

- `replayPlot()` issues a warning if an attempt is made to replay a plot that was recorded using a different R version (the format for recorded plots is not guaranteed to be stable across different R versions). The Windows-menu equivalent (History...Get from variable) issues a similar warning.

- `reshape()` can handle multiple 'id' variables.

- It is now possible to specify colours with a full alpha transparency channel via the new 'alpha' argument to the `rgb()` and `hsv()` functions, or as a string of the form "#RRGGBBAA".

  NOTE: most devices draw nothing if a colour is not opaque, but PDF and Quartz devices will render semitransparent colours.

  A new argument 'alpha' to the function `col2rgb()` provides the ability to return the alpha component of colours (as well as the red, green, and blue components).

- `save()` now checks that a binary connection is used.

- `seek()` on connections now accepts and returns a double for the file position. This allows >2Gb files to be handled on a 64-bit platform.

- `source()` with `echo = TRUE` uses the function source attribute when displaying commands as they are parsed.

- `setClass()` and its utilities now warn if either superclasses or classes for slots are undefined. (Use `setOldClass` to register S3 classes for use as slots)

- `str(obj)` now displays more reasonably the STRucture of S4 objects. It is also improved for language objects and lists with promise components.

  The method for class "dendrogram" has a new argument 'stem' and indicates when it's not printing all levels (as typically when e.g., `max.level = 2`).

  Specifying `max.level = 0` now allows to suppress all but the top level for hierarchical objects such as lists. This is different to previous behavior which was the default behavior of giving all levels is unchanged. The default behavior is unchanged but now specified by `max.level = NA`.

- `system.time()` has a new argument 'gcFirst' which, when `TRUE`, forces a garbage collection before timing begins.

- `tail()` of a matrix now displays the original row numbers.

- The default method for `text()` now coerces a factor to character and not to its internal codes. This is incompatible with S but seems what users would expect.

  It now also recycles (x,y) to the length of 'labels' if that is longer. This is now compatible with `grid.text()` and S. (See also PR#7084.)

- `TukeyHSD()` now labels comparisons when applied to an interaction in an `aov()` fit. It detects non-factor terms in 'which' and drops them if sensible to do so.

- There is now a replacement method for `window()`, to allow a range of values of time series to be replaced by specifying the start and end times (and optionally a frequency).

- If `writeLines()` is given a connection that is not open, it now attempts to open it in `mode = "wt"` rather than the default mode specified when creating the connection.

- The screen devices `x11()`, `windows()` and `quartz()` have a new argument 'bg' to set the default background colour.

- Subassignments involving `NA`s and with a replacement value of length $> 0$ are now disallowed. (They were handled inconsistently in R < 2.0.0, see PR#7210.) For data frames they are disallowed altogether, even for logical matrix indices (the only case which used to work).

- The way the comparison operators handle a list argument has been rationalized so a few more cases will now work – see `?Comparison`.

- Indexing a vector by a character vector was slow if both the vector and index were long (say 10,000). Now hashing is used and the time should be linear in the longer of the lengths (but more memory is used).

- Printing a character string with embedded nuls now prints the whole string, and non-printable characters are represented by octal escape sequences.

- Objects created from a formally defined class now include the name of the corresponding package as an attribute in the object's class. This allows packages with namespaces to have private (non-exported) classes.

- Changes to package 'grid':

  - Calculation of number of circles to draw in `circleGrob` now looks at length of y and r as well as length of x.
  - Calculation of number of rectangles to draw in `rectGrob` now looks at length of y, w, and h as well as length of x.
  - All primitives (rectangles, lines, text, ...) now handle non-finite values (`NA, Inf, -Inf, NaN`) for locations and sizes. Non-finite values for locations, sizes, and scales of viewports result in error messages. There is a new vignette(`nonfinite`) which describes this new behaviour.
  - Fixed (unreported) bug in drawing circles. Now checks that radius is non-negative.
  - `downViewport()` now reports the depth it went down to find a viewport. Handy for "going back" to where you started.
  - The "alpha" `gpar()` is now multiplied by the alpha channel of colours when creating a gcontext. This means that `gpar(alpha=)` settings now affect internal colours so grid alpha transparency settings now are sent to graphics devices. The alpha setting is also cumulative.
  - Editing a gp slot in a grob is now incremental.
  - The "cex" gpar is now cumulative. For example ...
  - New childNames() function to list the names of children of a gTree.
  - The "grep" and "global" arguments have been implemented for `grid.[add|edit|get|remove]Grob()` functions.

    The "grep" argument has also been implemented for the `grid.set()` and `setGrob()`.

- New function `grid.grab()` which creates a gTree from the current display list (i.e., the current page of output can be converted into a single gTree object with all grobs on the current page as children of the gTree and all the viewports used in drawing the current page in the childrenvp slot of the gTree).
- New "lineend", "linejoin", and "linemitre" `gpar()`s: line end can be "round", "butt", or "square"; line join can be "round", "mitre", or "bevel"; line mitre can be any number larger than 1 (controls when a mitre join gets turned into a bevel join; proportional to angle between lines at join; very big number means that conversion only happens for lines that are almost parallel at join).
- New `grid.prompt()` function for controlling whether the user is prompted before starting a new page of output.

  Grid no longer responds to the `par(ask)` setting in the "graphics" package.

- The tcltk package has had the `tkcmd()` function renamed as `tcl()` since it could be used to invoke commands that had nothing to do with Tk. The old name is retained, but will be deprecated in a future release. Similarly, we now have `tclopen()`, `tclclose()`, `tclread()`, `tclputs()`, `tclfile.tail()`, and `tclfile.dir()` replacing counterparts starting with "tk", with old names retained for now.

## New and changed utilities

- `R CMD check` now checks for file names in a directory that differ only by case.
- `R CMD check` now checks Rd files using R code from package tools, and gives refined diagnostics about "likely" Rd problems (stray top-level text which is silently discarded by `Rdconv`).
- `R CMD INSTALL` now fails for packages with incomplete/invalid `DESCRIPTION` metadata, using new code from package tools which is also used by `R CMD check`.
- `list_files_with_exts` (package 'tools') now handles zipped directories.
- Package 'tools' now provides `Rd_parse()`, a simple top-level parser/analyzer for R documentation format.
- `tools::codoc()` (and hence `R CMD check`) now checks any documentation for registered S3 methods and unexported objects in packages with namespaces.
- Package 'utils' contains several new functions:

  - Generics `toBibtex()` and `toLatex()` for converting R objects to BibTeX and LaTeX (but almost no methods yet).

  - A much improved `citation()` function which also has a package argument. By default the citation is auto-generated from the package `DESCRIPTION`, the file `inst/CITATION` can be used to override this, see `help(citation)` and `help(citEntry)`.

  - `sessionInfo()` can be used to include version information about R and R packages in text or LaTeX documents.

## Documentation changes

- The DVI and PDF manuals are now all made on the paper specified by `R_PAPERSIZE` (default 'a4'), even the `.texi` manuals which were made on US letter paper in previous versions.
- The reference manual now omits 'internal' help pages.
- There is a new help page shown by `help("Memory-limits")` which documents the current design limitations on large objects.
- The format of the LaTeX version of the documentation has changed. The old format is still accepted, but only the new resolves cross-references to object names containing _, for example.
- HTML help pages now contain a reference to the package and version in the footer, and HTML package index pages give their name and version at the top.
- All manuals in the 2.x series have new ISBN numbers.
- The *R Data Import/Export* manual has been revised and has a new chapter on *Reading Excel spreadsheets*.

## Changes in C-level facilities

- The `PACKAGE` argument for `.C`/`.Call`/`.Fortran`/`.External` can (and should) be omitted if the call is within code within a package with a namespace. This ensures that the native routine being called is found in the DLL of the correct version of the package if multiple versions of a package are loaded in the R session. Using a namespace and omitting the `PACKAGE` argument is currently the only way to ensure that the correct version is used.

- The header `Rmath.h` contains a definition for `R_VERSION_STRING` which can be used to track different versions of R and libRmath.
- The Makefile in `src/nmath/standalone` now has 'install' and 'uninstall' targets – see the README file in that directory.
- More of the header files, including `Rinternals.h`, `Rdefines.h` and `Rversion.h`, are now suitable for calling directly from C++.

## Newly deprecated and defunct

- Direct use of `R INSTALL|REMOVE|BATCH|COMPILE|SHLIB` has been removed: use `R CMD` instead.
- `La.eigen()`, `tetragamma()`, `pentagamma()`, `package.contents()` and `package.description()` are defunct.
- The undocumented function `newestVersion()` is no longer exported from package utils. (Mainly because it was not completely general.)
- C-level entry point `ptr_R_GetX11Image` has been removed, as it was replaced by `R_GetX11Image` at 1.7.0.
- The undocumented C-level entry point `R_IsNaNorNA` has been removed. It was used in a couple of packages, and should be replaced by a call to the documented macro `ISNAN`.
- The gnome/GNOME graphics device is now defunct.

## Installation changes

- Arithmetic supporting +/-Inf, NaNs and the IEC 60559 (aka IEEE 754) standard is now required — the partial and often untested support for more limited arithmetic has been removed.

  The C99 macro `isfinite` is used in preference to `finite` if available (and its correct functioning is checked at configure time).

  Where `isfinite` or `finite` is available and works, it is used as the substitution value for `R_FINITE`. On some platforms this leads to a performance gain. (This applies to compiled code in packages only for `isfinite`.)
- The dynamic libraries libR and libRlapack are now installed in `R_HOME/lib` rather than `R_HOME/bin`.
- When `--enable-R-shlib` is specified, the R executable is now a small executable linked against libR: see the R-admin manual for further discussion. The 'extra' libraries bzip2, pcre, xdr and zlib are now compiled in a way that allows the code to be included in a shared library only if this option is specified, which might improve performance when it is not.
- The main R executable is now `R_HOME/exec/R` not `R_HOME/R.bin`, to ease issues on MacOS X. (The location is needed when debugging core dumps, on other platforms.)
- Configure now tests for `inline` and alternatives, and the `src/extra/bzip2` code now (potentially) uses inlining where available and not just under gcc.
- The XPG4 `sed` is used on Solaris for forming dependencies, which should now be done correctly.
- Makeinfo 4.5 or later is now required for building the HTML and Info versions of the manuals. However, binary distributions need to be made with 4.7 or later to ensure some of the links are correct.
- `f2c` is not allowed on 64-bit platforms, as it uses longs for Fortran integers.
- There are new options on how to make the PDF version of the reference manual — see the *R Administration and Installation Manual* section 2.2.
- The concatenated Rd files in the installed 'man' directory are now compressed and the `R CMD check` routines can read the compressed files.
- There is a new configure option `--enable-lfs` that will build R with support for $>$ 2Gb files on suitable 32-bit Linux systems.

## Package installation changes

- The `DESCRIPTION` file of packages may contain a `Imports:` field for packages whose namespaces are used but do not need to be attached. Such packages should no longer be listed in `Depends:`.
- There are new optional fields `SaveImage`, `LazyLoad` and `LazyData` in the DESCRIPTION file. Using `SaveImage` is preferred to using an empty file `install.R`.
- A package can contain a file `R/sysdata.rda` to contain system datasets to be lazy-loaded into the namespace/package environment.

- The packages listed in `Depends:` are now loaded before a package is loaded (or its image is saved or it is prepared for lazy loading). This means that almost all uses of `R_PROFILE.R` and `install.R` are now unnecessary.
- If installation of any package in a bundle fails, `R CMD INSTALL` will back out the installation of all of the bundle, not just the failed package (on both Unix and Windows).

## Bug fixes

- Complex superassignments were wrong when a variable with the same name existed locally, and were not documented in R-lang.
- `rbind.data.frame()` dropped names/rownames from columns in all but the first data frame.
- The `dimnames<-` method for data.frames was not checking the validity of the row names.
- Various memory leaks reported by valgrind have been plugged.
- `gzcon()` connections would sometimes read the crc bytes from the wrong place, possibly uninitialized memory.
- Rd.sty contained a length `\middle` that was not needed after a revision in July 2000. It caused problems with LaTeX systems based on e-TeX which are starting to appear.
- `save()` to a connection did not check that the connection was open for writing, nor that non-ascii saves cannot be made to a text-mode connection.
- `phyper()` uses a new algorithm based on Morten Welinder's bug report (PR#6772). This leads to faster code for large arguments and more precise code, e.g. for `phyper(59, 150,150, 60, lower=FALSE)`. This also fixes bug (PR#7064) about `fisher.test()`.
- `{print.default(*, gap = <n>)` now in principle accepts all non-negative values `<n>`.
- `smooth.spline(...)$pen.crit` had a typo in its computation; note this was printed in `print.smooth.spline()` but not used in other "smooth.spline" methods.
- `write.table()` handles zero-row and zero-column inputs correctly.
- `debug()` works on trivial functions instead of crashing (PR#6804)
- `eval()` could alter a data.frame/list second argument, so `with(trees, Girth[1] <- NA)` altered `trees` (and any copy of `trees` too).
- `cor()` could corrupt memory when the standard deviation was zero. (PR#7037)
- `inverse.gaussian()` always printed `1/mu^2` as the link function.
- `constrOptim` now passes `...` arguments through `optim` to the objective function.
- `object.size()` now has a better estimate for character vectors: it was in general too low (but only significantly so for very short character strings) but over-estimated `NA` and duplicated elements.
- `quantile()` now interpolates correctly between finite and infinite values (giving +/-Inf rather than NaN).
- `library()` now gives more informative error messages mentioning the package being loaded.
- Building the reference manual no longer uses roman upright quotes in typewriter output.
- `model.frame()` no longer builds invalid data frames if the data contains time series and rows are omitted by `na.action`.
- `write.table()` did not escape quotes in column names. (PR#7171)
- Range checks missing in recursive assignments using `[[ ]]`. (PR#7196)
- `packageStatus()` reported partially-installed bundles as installed.
- `apply()` failed on an array of dimension $\geq 3$ when for each iteration the function returns a named vector of length $\geq 2$ (PR#7205)
- The GNOME interface was in some circumstances failing if run from a menu — it needed to always specify that R be interactive.
- `depMtrxToStrings` (part of `pkgDepends`) applied `nrow()` to a non-matrix and aborted on the result.
- Fix some issues with nonsyntactical names in modelling code (PR#7202), relating to backquoting. There are likely more.
- Support for S4 classes that extend basic classes has been fixed in several ways. `as()` methods and `x@.Data` should work better.
- `hist()` and `pretty()` accept (and ignore) infinite values. (PR#7220)
- It is no longer possible to call `gzcon()` more than once on a connection.

 ISSN 1609-3631

- `t.test()` now detects nearly-constant input data. (PR#7225)

- `mle()` had problems if ndeps or parscale was supplied in the control arguments for `optim()`. Also, the profiler is now more careful to reevaluate modified `mle()` calls in its parent environment.

- Fix to rendering of accented superscripts and subscripts e.g., `expression((b[dot(a)]))`. (Patch from Uwe Ligges.)

- `attach(*, pos=1)` now gives a warning (and will give an error).

- `power.*test()` now gives an error when 'sig.level' is outside [0,1]. (PR#7245)

- Fitting a binomial glm with a matrix response lost the names of the response, which should have been transferred to the residuals and fitted values.

- `print.ts()` could get the year wrong because rounding issue (PR#7255)

# Changes on CRAN

*by Kurt Hornik*

## New contributed packages

**Malmig** The Malmig package provides an implementation of Malecot migration model in R together with a number of related functions. By Federico C. F. Calboli and Vincente Canto Cassola together with Martin Maechler authored the function `mtx.exp`.

**PBSmapping** This software has evolved from fisheries research conducted at the Pacific Biological Station (PBS) in Nanaimo, British Columbia, Canada. It extends the R language to include two-dimensional plotting features similar to those commonly available in a Geographic Information System (GIS). Embedded C code speeds algorithms from computational geometry, such as finding polygons that contain specified point events or converting between longitude-latitude and Universal Transverse Mercator (UTM) coordinates. It includes data for a global shoreline and other data sets in the public domain. By Nicholas Boers, Jon Schnute, Rowan Haigh, and others.

**RCurl** The package allows one to compose HTTP requests to fetch URIs, post forms, etc., and process the results returned by the Web server. This provides a great deal of control over the HTTP connection and the form of the request while providing a higher-level interface than is available just using R socket connections. Additionally, the underlying implementation is robust and extensive, supporting SSL/HTTPS, cookies, redirects, authentication, etc. By Duncan Temple Lang.

**RNetCDF** This package provides an interface to Unidata's NetCDF library functions (version 3) and furthermore access to Unidata's udunits calendar conversions. The routines and the documentation follow the NetCDF and udunits C interface, so the corresponding manuals can be consulted for more detailed information. By Pavel Michna.

**Rstem** An R interface to the C code that implements Porter's word stemming algorithm for collapsing words to a common root to aid comparison of texts. There is code to for different languages (i.e., Danish, Dutch, English, Finnish, French, German, Norwegian, Portuguese, Russian, Spanish, Swedish). However, these may not be applicable if the words require UTF encoding. This is extensible by allowing different routines to be specified to create the C routines used in the stemming, permitting debugging, profiling, pool management, caching, etc. By Duncan Temple Lang.

**UNF** Computes a universal numeric fingerprint of the data. By Micah Altman.

**accuracy** This is a suite of tools designed to test and improve the accuracy of statistical computation, including: Summarization of the sensitivity of linear and non-linear models (lm, glm, mle, nls) to measurement and numerical error; A generalized Cholesky method for correcting non-invertible Hessians; Tests for the global optimality of non-linear regression and maximum likelihood results; Tools for obtaining true random numbers using entropy collected from the system and/or entropy servers on the internet; A method for converting floating point numbers to normalized fractions; Benchmark data for checking the accuracy of basic distribution functions. By Micah Altman, Jeff Gill, and Michael P. McDonald.

**adehabitat** A collection of tools for the analysis of habitat selection by animals. By Clément Calenge, contributions from Mathieu Basille.

**bayesSurv** Bayesian survival regression with flexible error and (later on also random effects) distributions. By Arnost Komarek.

**catspec** 'sqtab' contains a set of functions for estimating loglinear models for square tables such as quasi-independence, symmetry, uniform association. 'mclgen' restructures a dataframe to enable the estimation of a multinomial logistic model using the conditional logit program 'clogit'. This allows greater flexibility in imposing constraints on the response variable. One application is to specify aforementioned models for square tables as multinomial logistic models with covariates at the respondent level. 'ctab' simplifies the production of (multiway) percentage tables. By John Hendrickx.

**chplot** Informative and nice plots for grouped bivariate data. By Maja Pohar and Gaj Vidmar.

**drfit** drfit provides basic functions for accessing the dose-response data of the UFT Bremen, Department of Bioorganic Chemistry, fitting dose-response curves to this and similar data, calculating some (eco)toxicological parameters and plotting the results. Functions that are fitted are the cumulative density function of the lognormal distribution, of the logistic distribution and a linear-logistic model, derived from the latter, which is used to describe data showing stimulation at low doses (hormesis). The author would be delighted if anyone would join in this effort of creating useful and useable tools for dealing with dose-response data from biological testing. By Johannes Ranke.

**eba** Fitting and testing probabilistic choice models, especially the BTL model (Bradley & Terry, 1952; Luce, 1959), elimination-by-aspects (EBA) models (Tversky, 1972), and preference tree (Pretree) models (Tversky & Sattath, 1979). By Florian Wickelmaier.

**ebayesthresh** Carries out Empirical Bayes thresholding using the methods developed by Johnstone and Silverman. The basic problem is to estimate a mean vector given a vector of observations of the mean vector plus white noise, taking advantage of possible sparsity in the mean vector. Within a Bayesian formulation, the elements of the mean vector are modelled as having, independently, a distribution that is a mixture of an atom of probability at zero and a suitable heavy-tailed distribution. The mixing parameter can be estimated by a marginal maximum likelihood approach. This leads to an adaptive thresholding approach on the original data. Extensions of the basic method, in particular to wavelet thresholding, are also implemented within the package. By Bernard Silverman (with major intellectual input from Iain Johnstone).

**faraway** Functions and datasets for books by Julian Faraway. Books are "Practical Regression and ANOVA in R" on CRAN, "Linear Models with R" appearing in August 2004 published by CRC press and "Extending the Linear Model with R" a book in preparation. By Julian Faraway.

**gam** Functions for fitting and working with generalized additive models, as described in chapter 7 of "Statistical Models in S" (Chambers and Hastie (eds), 1991), and "Generalized Additive Models" (Hastie and Tibshirani, 1990). By Trevor Hastie.

**hierfstat** This R package allows the estimation of hierarchical F-statistics from haploid or diploid genetic data with any numbers of levels in the hierarchy, following the algorithm of Yang (Evolution, 1998, 52(4):950–956). Functions are also given to test via randomisations the significance of each F and variance components, using the likelihood-ratio statistics G, see Goudet et.al. (Genetics, 1996, 144(4): 1933–1940). By Jerome Goudet.

**kinship** `coxme`: general mixed-effects Cox models; `kinship`: routines to create and manipulate $n$ by $n$ matrices that describe the genetic relationships between $n$ persons; `pedigree`: create and plot pedigrees; `bdsmatrix`: a class of objects for sparse block-diagonal matrices (which is how kinship matrices are stored); `gchol`: generalized cholesky decompositions. By Beth Atkinson (atkinson@mayo.edu) for pedigree functions, and Terry Therneau (therneau@mayo.edu) for all other functions.

**limma** Data analysis, linear models and differential expression for microarray data. By Gordon Smyth with contributions from Matt Ritchie, James Wettenhall and Natalie Thorne.

**locfdr** Computation of local false discovery rates. By Bradley Efron and Balasubramanian Narasimhan.

**mfp** Fractional polynomials are used to represent curvature in regression models. A key reference is Royston and Altman, 1994. By Gareth Ambler, with contributions from Axel Benner.

**mitools** Tools to perform analyses and combine results from multiple-imputation datasets. By Thomas Lumley.

**ncvar** This package provides a high-level R interface to Unidata's NetCDF data files. Using this package, netCDF datasets, and all their associated metadata, can be read and written in one go. It is also easy to create datasets including lots of metadata. This package supports both the CF and default NetCDF metadata conventions. It supports more general NetCDF files and conventions than the **ncdf** package by David Pierce, using the low-level NetCDF package **RNetCDF** by Pavel Michna. By Juerg Schmidli.

**plotrix** Various useful functions for enhancing plots. By Jim Lemon.

**pwt** The Penn World Table provides purchasing power parity and national income accounts converted to international prices for 168 countries for some or all of the years 1950–2000. By Guan Yang.

**reldist** R functions for the comparison of distributions. This includes nonparametric estimation of the relative distribution PDF and CDF and numerical summaries as described in "Relative Distribution Methods in the Social Sciences" by Mark S. Handcock and Martina Morris, Springer-Verlag, 1999, Springer-Verlag, ISBN 0387987789. By Mark S. Handcock.

**rmetasim** An interface between R and the metasim simulation engine. Facilitates the use of the metasim engine to build and run individual based population genetics simulations. The simulation environment is documented in: Allan Strand. Metasim 1.0: an individual-based environment for simulating population genetics of complex population dynamics. Mol. Ecol. Notes, 2:373–376, 2002. (Please contact Allan Strand with comments, bug reports, etc). By Allan Strand and James Niehaus.

**snowFT** Extension of the snow package supporting fault tolerant and reproducible applications. It is written for the PVM communication layer. By Hana Sevcikova and A. J. Rossini.

**taskPR** The Task-Parallel R ('task-pR') system, repackaged as an R package. By Nagiza F. Samatova, David Bauer, and Srikanth Yoginath.

**tuneR** Collection of tools to analyze music, handling wave files, transcription, etc. By Uwe Ligges with contributions from Andrea Preusser and Claus Weihs.

**vioplot** A violin plot is a combination of a box plot and a kernel density plot. By Daniel Adler.

## Other changes

- Package **RmSQL** was moved from the main CRAN section to the Archive.

*Kurt Hornik*
*Wirtschaftsuniversität Wien, Austria*
Kurt.Hornik@R-project.org



*R News* is a publication of the R Foundation for Statistical Computing, communications regarding this publication should be addressed to the editors. All articles are copyrighted by the respective authors. Please send submissions to regular columns to the respective column editor, all other submissions to the editor-in-chief or another member of the editorial board (more detailed submission instructions can be found on the R homepage).

R Project Homepage:
http://www.R-project.org/

This newsletter is available online at
http://CRAN.R-project.org/doc/Rnews/