R Help Desk

Automation of Mathematical Annotation in Plots

Uwe Ligges

Welcome to the R Help Desk

Welcome to the first issue of a regular series of *R Help Desk* columns.

As the title of the column suggests, it is intended to present answers to frequently asked questions related to R, for example well known questions from the R mailing lists. More specifically, the intention is to address problems which cannot be described and explained completely in a few lines of text, as it is common in manuals, help pages, typically styled answers on mailing lists, or the R FAQ (Hornik, 2002).

So, on the one hand, articles published in this column are intended to present solutions to common problems. On the other hand, to be easy to read by less experienced R users, the articles should not be too technical.

Contributions

It is a pleasure to start as the editor of this column. Like Bill Venables in his first issue of the *Programmer's Niche*, I would like to take the opportunity to invite *you*, the reader, to *contribute* articles. If you have any ideas on how to describe solutions to common programming problems and (more or less) frequently asked questions, please send your contributions to `ligges@statistik.uni-dortmund.de`.

Introduction to mathematical annotation in plots

Many users know about R's capabilities of typesetting mathematical annotation in plots, which were introduced by Murrell and Ihaka (2000). Related to this topic, I frequently heard and read sentences like "Mathematical annotation in plots can be typeset using a LaTeX–like syntax".

- This statement is partly true for two reasons: Typesetting is programmed in a way, both in R and in LaTeX. The "keywords" for typesetting objects like greek letters, fractions etc. are quite similar.

- The statement is mainly wrong, or at least confusing: The *syntax* is fortunately more or less the syntax of the S language with some small specialities, therefore most R users will know about its main rules.

Let us start collecting the required information to typeset formulas in R. We do not need any special functions to typeset, since the regular mechanisms used to typeset character strings, like the arguments `main` or `xlab` in `plot()`, or functions like `text()` etc., are sufficient.

We will not be able to specify mathematical annotation as character strings, but we need to specify them as S expressions or calls — without evaluating them. Some functions to specify unevaluated S expression, or, more specifically, functions to manipulate and work with S expressions and calls are described in detail by Venables and Ripley (2000). For those interested in more technicalities, the Programmer's Niche by Venables (2002) in the previous newsletter gives some nice insights into the language.

Anyway, the most frequently used reference for our purpose certainly is the help page `?plotmath`.

Having collected most of the required information, we know that `expression()` is an appropriate function to specify an unevaluated S expression. Thus, we can easily produce the following example (just try it out!).

```
> plot(0, main =
    expression(y == alpha*x[1] + beta*x[2]^2))
```

The resulting plot will have a rather nice formula in its main title.

Automation

The question how to replace some variables in *formulas* by their *values* seems to be more sophisticated, but is still documented in the examples of `?plotmath`.

In particular, the function `substitute()` is designed to substitute any variables in a call by their value (from a given environment or list of objects). As an extension of our first example

```
> a <- 3.5
> x <- 1:2
> substitute(y == a + alpha*x[1] + beta*x[2]^2,
    list(a = a))
```

will replace the variable `a` (but not `x`) in the expression with its value. Such a mechanism is of special interest for some automated generation of plots, where the user is not willing to specify the calls to each plot separately.

Let us construct a small example (you might want to try it out before looking at the code): Consider you are working with a bivariate normal distribution, for which you automatically calculate the mean μ and the covariance matrix Σ_x:

$$\mu = \begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix}, \quad \Sigma_x = \begin{pmatrix} \sigma_1 & \sigma_3 \\ \sigma_2 & \sigma_4 \end{pmatrix}.$$

In the same procedure, you want to generate a nice plot for some presentation, including the formula for the density of your multivariate normal distribution:

$$f(x) = \frac{1}{\sqrt{(2\pi)^n \det(\Sigma_x)}} \times \exp\left(-\frac{1}{2}(x-\mu)^T \Sigma_x^{-1}(x-\mu)\right).$$

Further on, suppose you would like to print out the calculated values of μ and Σ_x in the same plot. One possible solution would be the following code.

```
> ## Let us set up an empty plot:
> plot(1:8, type = "n")
> ## a list of imaginary calculated values:
> param.list <- list(mu1 = 0, mu2 = 0,
    s1 = 3, s2 = 2, s3 = 2, s4 = 4)
> ## typeset density of 2-var. normal dist.
> text(1, 6, adj = 0, labels = expression(
    f(x) == frac(1, sqrt((2 * pi)^n ~~
      det(Sigma[x]))) ~~ exp * bgroup("(",
      -frac(1, 2) ~~ (x - mu)^T * Sigma[x]^-1 *
      (x - mu), ")")))
> ## typeset concrete values of mu and Sigma
> ##  (from param.list):
> text(8, 3, adj = 1, labels = substitute(
    "with " * mu == bgroup("(", atop(mu1, mu2),
      ")") * " , " * Sigma[x] ==
      bgroup("(", atop(s1 ~~ s3, s2 ~~ s4), ")"),
      param.list))
```

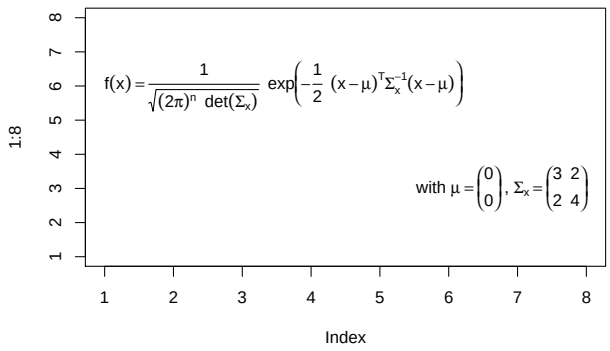

Figure 1: Example – Typesetting the density of a bivariate normal distribution with substituted values

More automation

From another point of view, substitution might be desirable for variables (objects) *containing* expressions, or expression-like strings. Consider you would like to pass such an object as an argument to a function, and within that function the title for the plot shall be constructed from different elements.

The following function will plot the object x according to its class, and label the plot with a compounded title. For the labelling two different objects `arg2` (an expression) and `arg3` (a character string) are expected by the function. This particular design of the function is chosen for illustrating two possible ways to achieve the typesetting:

```
> my.foo <- function(x, arg2, arg3, ...){
    arg3 <- parse(text = arg3)[[1]]
    plot(x, main =
      substitute("Formula in \'arg2\': " * arg2
        * ";  Formula in \'arg3\': " * arg3,
        list(arg2 = arg2, arg3 = arg3)),
      ...)
  }
> my.foo(1:10, arg2 = quote(alpha[1] == 5),
    arg3 = "y == alpha + beta*x + epsilon")
```

Neither for `arg2`, nor for `arg3`, it is possible to substitute the variable by an object of mode `expression`, which can include several objects of mode `call`, because `substitute()` will fail in that case. Instead, the trick is to pass an object of mode `call`. In the first case, `arg2` is specified in the function call using `quote()`. In the second case, `arg3` is specified as a character string that is parsed (i.e. an expression is returned) inside the function. In order to get an object of mode `call`, only the first element of the returned list is extracted.

Legends

Another quite frequently asked question related to mathematical annotation is how to deal with a couple of formulas at once, as required in legends. If substitution of variables by values is not necessary, `expression()` will still do the trick:

```
> plot(1:8, type = "n")
> legend(2, 3, expression(alpha^2, x[5], Omega))
```

But what about substituting? Consider you have calculated values $\alpha = 1, \beta = 2$, and want to present those values within any legend. You will have to substitute the variables of each legend's element separately before putting them together in an expression. The latter can be done by `do.call()` in a somewhat tricky manner, constructing a call to `expression()` with the legend's elements as its arguments:

```
> a <- 3; b <- 5
> legend1 <- substitute(alpha == a, list(a = a))
> legend2 <- substitute(beta == b, list(b = b))
> legend(5, 5,
    do.call("expression", list(legend1, legend2))
```

Working intensively with mathematical annotation in plots involves the use of expressions and calls, thus it is close to the language. Therefore this first issue of the *R Help Desk* got a bit more technical than it was intended to be.

I would like to close with a nice citation of Venables (2002): "Mind Your Language".

Bibliography

K. Hornik (2002). *The R FAQ*. ISBN 3-901167-51-X, http://www.ci.tuwien.ac.at/~hornik/R/. 32

P. Murrell and R. Ihaka (2000). An Approach to Providing Mathematical Annotation in Plots, *Journal of Computational and Graphical Statistics*, 9(3): 582–599. 32

W. N. Venables and B. D. Ripley (2000). *S Programming*. Springer-Verlag, New York. 32

W. N. Venables (2002). Programmer's Niche, *R News*, 2(2): 24–26, ISSN 1609-3631, http://CRAN.R-project.org/doc/Rnews/. 32, 33

Uwe Ligges
Fachbereich Statistik, Universität Dortmund, Germany
ligges@statistik.uni-dortmund.de

Changes in R

by the R Core Team

User-visible changes

- The default colour palette now has `"grey"` instead of `"white"` in location 8. See `palette()`.
- `grid(nx)` behaves differently (but the same as in R versions <= 0.64).

New features

- `barplot()` has a new argument 'axis.lty', which if set to 1 allows the pre-1.6.0 behaviour of plotting the axis and tick marks for the categorical axis. (This was apparently not intentional, but `axis()` used to ignore lty=0.) The argument 'border' is no longer "not yet used".
- New operator `::` in the grammar, for name spaces.
- New faster `rowsum()`, also works on data frames.
- `grep()`, `sub()`, `gsub()` and `regexpr()` have a new argument 'perl' which if `TRUE` uses Perl-style regexps from PCRE (if installed). New capabilities option `"PCRE"` to say if PCRE is available.
- Preparations for name space support:
  - Functions in the **base** package are now defined in a name space. As a temporary measure, you can disable this by defining the environment variable `R_NO_BASE_NAMESPACE`.
  - `UseMethod` dispatching now searches for methods in the environment of the caller of the generic function rather than the environment where the generic is defined.
- The objects created in the **methods** package to represent classes, generic functions, method definitions, and inheritance relations now themselves belong to true classes. In particular, the `"classRepresentation"` objects follow the description in "Programming with Data" (section 7.6).
- Other additions and changes to the **methods** package:
  - The function `setOldClass()` has been added, following the description on page 450 of "Programming with Data". Use it if old-style classes are to be supplied in signatures for `setMethod`, particularly if the old-style classes have inheritance. Many of the old-style classes in the base package should be pre-specified; try `getClass("mlm")`, e.g.
  - The `setGeneric()` function applies some heuristics to warn about possibly erroneous generic function definitions. (Before, obscure bugs could result.)
  - The function `promptMethods()` has been revised to work better and to provide aliases for individual methods.
  - The behavior of the `as()` function has been generalized, in particular with a '`strict=`' argument, the general goal being to let simple extensions of classes pass through in method dispatch and related computations without altering the objects. More to make method behavior more "natural" than for direct use.
  - Some inconsistencies following `detach("package:methods")` have been removed, so it *should* be possible to detach/re-attach the methods package.
- New methods (`[[`, `print`, `str`) and extended `plot()` method (including logical 'horiz') for `"dendrogram"` class.