Extending Lattice: Using generics and methods to implement new visualization methods within the Trellis framework

Deepayan Sarkar

Abstract

The lattice add-on package implements Trellis graphics in R. One of the major recent changes to the package API has been to make all high level functions generic, with the traditional implementations available as the "formula" method. This allows for cleaner and more flexible implementations for certain uses that were permitted in the original S-PLUS version on a one-off basis. For example, dotplot could be used to display one-way tables, but the new approach naturally extends to multi-way tables as well. More importantly, it opens up the possibility of new Trellis displays specifically designed for previously unsupported classes. We present some examples of such extensions and describe a few issues generally relevant to the development of new Trellis-style visualizations using the lattice infrastructure.

1 Introduction

The lattice add-on package for R (R Development Core Team, 2007) implements common statistical graphics with multipanel conditioning, not unlike the Trellis suite in S-PLUS. The primary user interface is a collection of high level functions (xyplot, histogram, cloud, etc.), each producing a certain type of statistical graphic by default. Many variations of these standard graphics are built into lattice, and can be activated with additional arguments in the high level function calls. Figure 1 gives an example using the densityplot function. However, other kinds of extensions may not be as easily implemented by the casual user. Third party software authors thus have the opportunity to develop non-trivial extensions that implement novel visualization methods, while taking advantage of the considerable infrastructure already present in lattice. In this paper, we give an overview of the facilities in lattice that aid such extensions.

1.1 Panel functions

The conventional approach to implementing novel visualizations has been to write new panel functions (and sometimes new prepanel functions as well). To

```
> faithful$Eruptions <- equal.count(faithful$eruptions, 4)
> densityplot(~ waiting | Eruptions, faithful,
+             kernel = "epanechnikov", plot.points = "rug")
```

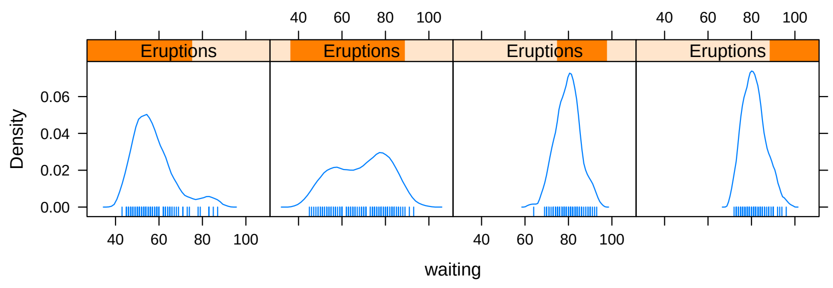

Figure 1: Kernel density estimate of waiting time till eruptions of the Old Faithful geyser, conditional on the duration of the previous eruption. An optional argument is used to choose the Epanechnikov kernel rather than the default Gaussian. Arguments to default panel functions (panel.densityplot in this example) can be supplied in this manner to produce many common variants.

continue the density example from Figure 1, suppose one wishes to forego the kernel density estimation method altogether and use the log-spline density esitimate (Stone et al., 1997) implemented in the logspline package. This can be achieved, as shown in Figure 2, through a panel function that computes and draws the density, along with a prepanel function that computes panel limits. The relevant functions are defined as:

```
> library(logspline)
> prepanel.ls <- function(x, n = 50, ...) {
+     fit <- logspline(x)
+     xx <- do.breaks(range(x), n)
+     yy <- dlogspline(xx, fit)
+     list(ylim = c(0, max(yy)))
+ }
> panel.ls <- function(x, n = 50, ...) {
+     fit <- logspline(x)
+     xx <- do.breaks(range(x), n)
+     yy <- dlogspline(xx, fit)
+     panel.rug(x = x, start = 0, end = 0,
+               x.units = c("npc", "native"))
+     panel.lines(xx, yy, ...)
+ }
```

```
> densityplot(~ waiting | Eruptions, data = faithful,
+             prepanel = prepanel.ls, panel = panel.ls)
```

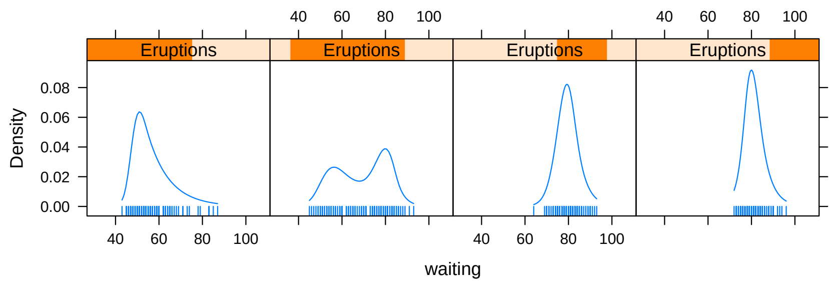

Figure 2: Conditional log-spline density estimates, implemented using user defined prepanel and panel functions.

1.2 Limitations

Not all extensions can be formulated in this manner. Consider the following survival fit with two groups:

```
> library(survival)
> fit <- survfit(Surv(time, status) ~ x, data = aml)
```

Suppose we wish to plot the survival curves for the two groups in separate panels (which is not possible with the plot.survfit method) along with confidence bands. It so happens that all the relevant information is accessible from the "survfit" object, and the plot we want is fairly simple to produce, as shown in Figure 3. However, this approach has several limitations, not least of which is that it requires the user to know the internal representation of "survfit" objects.

1.3 Generics and methods

One solution to this is a generic-method system: specifically, if one could write an xyplot method for "survfit" objects, such a method could encapsulate any non-standard arguments and special knowledge of class internals. This has the obvious benefit of not needing new function names unless they are appropriate, keeping the namespace relatively clean. To this end, all lattice high level functions have been made S3 generic functions, with argument list (x, data, ...). The data defining the display in these functions have traditionally been specified using a formula and data frame interface, much like the various statistical modeling functions in S. These interfaces are now available as "formula" methods, which are usually the workhorse that other methods end up calling. Thus, a rudimentary xyplot method for "survfit" objects might be implemented as

```
> xyplot.survfit <-
+     function(x, data = NULL, type = "s", conf.int = FALSE,
> xyplot(upper+surv+lower ~ time | rep(names(strata), strata),
+        data = fit, type = "s")
```

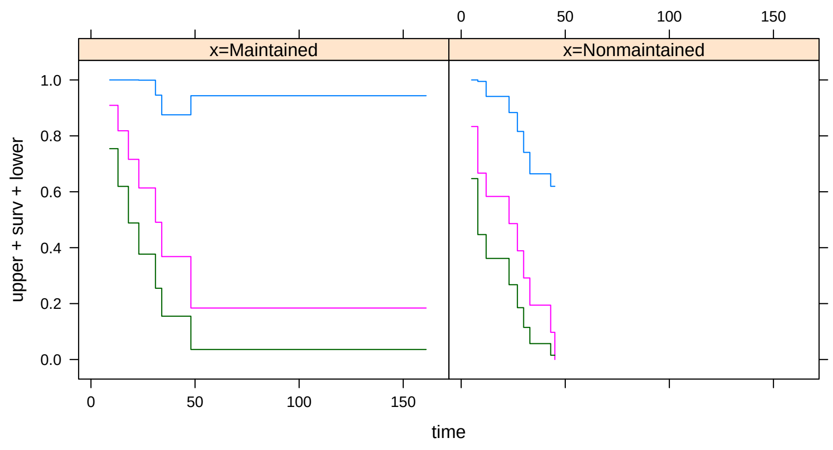

Figure 3: Survival curves for two groups. The call to produce this plot looks simple; however, it requires the user to know (1) that "survfit" objects are R lists, (2) the meaning of components upper, lower, surv and strata, (3) the fact that the formula in an xyplot call can contain multiple terms (separated by a "+" sign) that are superposed and (4) that the type="s" argument produces the suitable "steps". An xyplot.survfit method that encapsulates these details would be more intutive for the user, and could provide additional niceties like an option for marking censoring times.

```
+               ylab = "Probability of Survival", ...)
+ {
+     g <- with(x, rep(names(strata), strata))
+     if (conf.int)
+         xyplot(upper+surv+lower ~ time | g,
+                data = x, type = type, ylab = ylab, ...)
+     else
+         xyplot(surv ~ time, data = x, groups = g,
+                type = type, ylab = ylab, ...)
+ }
```

where the conf.int argument determines whether the confidence intervals will be plotted. By default, they are not, and all the survival curves are plotted within a single panel. Figure 3 can then be reproduced (with a better y axis label) using

```
> xyplot(fit, conf.int = TRUE)
```

In the remainder of this paper, we present several other, more realistic, examples that we hope can serve as models for developers interested in creating their own extensions.

2 Examples

2.1 S3 methods

In view of the generic-method approach, there are clearly two kinds of extensions. On one hand, there are situations where existing function names are appropriate; i.e., the function implements a familiar visualization but for a new type of data source. On the other hand, entirely new names might be appropriate for functions that implement novel visualizations. Examples of the first kind are given in Figure 4 (a dotplot method for matrices, from the lattice package) and Figure 5 (an xyplot method for a replicated MCMC object, from the coda package). For the most part, these methods transform their first argument to a suitable data frame and call the formula method, perhaps changing the defaults of some arguments and adding a few more that control the initial transformation. Rather than listing the function definitions here, we refer the reader to the actual implementations, available from CRAN (`http://cran.r-project.org`), for further details.

2.2 S4 methods

While the S3 scheme works well for plotting whole objects, it is insufficient in situations where the flexibility of a formula interface is desirable, but with data objects that go beyond the restrictive data frame paradigm. This is particularly important in the context of modern high throughput bioinformatics data, where covariate information is often referred to as “phenodata” (usually small)

```
> dotplot(VADeaths, auto.key = list(space = "right"),
+         xlab = "Rate (per 100)", aspect = 0.8, type = "o")
```

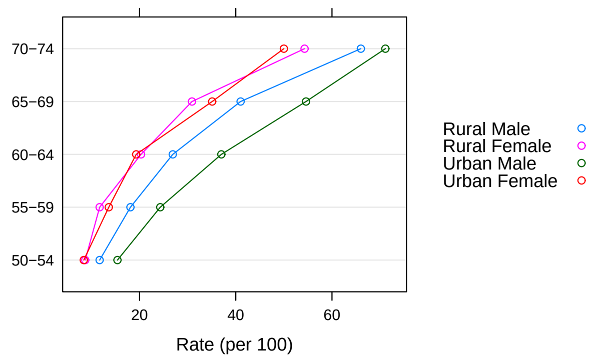

Figure 4: Dot plot of death rates (per 100) in Virginia, 1940, produced using the (S3) dotplot method for "matrix" objects. It is possible to recreate this plot with the traditional formula interface, but this requires familiarity with the as.data.frame.table function that must be used to transform the table into a suitable data frame.

and each "sample" consists of thousands of measurements on the basic experimental unit. Fortunately, multiple dispatch using the S4 method system makes this fairly simple and the primary challenge is the handling of potentially large data sets. In Figure 6, we use the densityplot method from the Bioconductor (Gentleman et al., 2004) package flowViz that dispatches on a "formula" object x and a "flowSet" object data. In this example, GvHD is a "flowSet" object with 35 samples with some associated pheonodata (e.g., Patient ID and Visit number). Each sample produces a (on average) 15000×8 data matrix, the columns of which (e.g., FSC-H) are the variables we are interested in visualizing. The naïve approach would be to convert the full data into an expanded data frame (a "join" operation), but this would produce a data frame with roughly 15000×35 rows! The solution used in the flowViz package is to use only the phenodata to construct a lattice call, keeping the environment containing the actual data in the scope of the prepanel and panel functions, which access it only when necessary. No particular care needs to be taken when defining such S4 methods, as dispatch to appropriate S3 methods is handled automatically.

2.3 New functions

Of course, existing generic function names may not be meaningful for new visualizations, and a completely new function name is often warranted. Even in such cases, our recommendation is to create the new function as a generic, along with specific methods as necessary. This has the benefit of encouraging future

```
> library(coda); data(line)
> xyplot(line, strip = FALSE, strip.left = TRUE, start = 10)
```

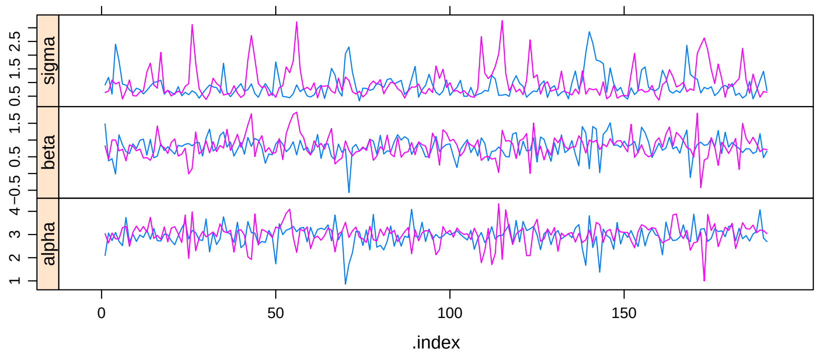

Figure 5: A diagnostic plot for MCMC objects, produced using the xyplot method for "mcmc.list" objects defined in the coda package. Unlike a standard xyplot call, this version by default allows different vertical scales and chooses a layout that makes panels as wide as possible (rather than have them be close to squares). The start argument, new in this method, is used to skip the first few samples.

extensions, and with a coordinated choice of argument names, it also allows multiple methods in multiple packages (perhaps written by different authors) to be used simultaneously without causing naming conflicts. An example of this is the rootogram function used in Figure 7 to create hanging rootograms (Tukey, 1977). While this example uses the "formula" method in the latticeExtra package, one can concurrently use the version of rootogram in the vcd package, where it is also a generic function. Generally speaking, creating a new visualization function usually involves writing new prepanel and panel functions, and writing a wrapper that ends up calling an existing lattice high level function with a compatible formula. The rootogram method for "formula" objects is a good example of this approach.

Our final example (Figure 8) uses the hexbinplot function from the hexbin package, which implements hexbin plots (Carr et al., 1987) with multipanel conditioning. This is a particularly interesting example because of the way it handles the legend describing the color coding within a panel, which is difficult because the lattice model has no formal mechanism to allow the panel function to communicate with the legend. As with high level functions in lattice, hexbinplot is an S3 generic function for which further methods can be written.

```
> library(flowViz); data(GvHD)
> densityplot(factor(Visit) ~ `FSC-H` | factor(Patient), data = GvHD)
```

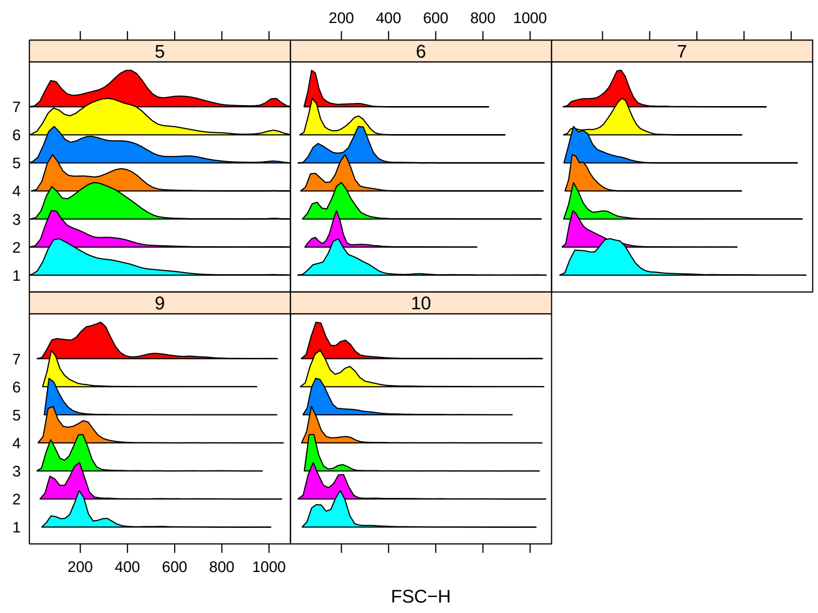

Figure 6: A visualization of the FSC-H channel in the GvHD data (flowViz). Each panel represents a patient, and the estimated densities of FSC-H for multiple visits are stacked on top of each other within each panel. The densityplot method used is an S4 method with signature x="formula", data="flowSet".

```
> library(latticeExtra)
> df <- make.groups(p47 = rpois(1000, lambda = 47),
+                   p50 = rpois(1000, lambda = 50),
+                   p53 = rpois(1000, lambda = 53))
> rootogram(~ data | which, data = df,
+           dfun = function(x) dpois(x, lambda = 50))
```

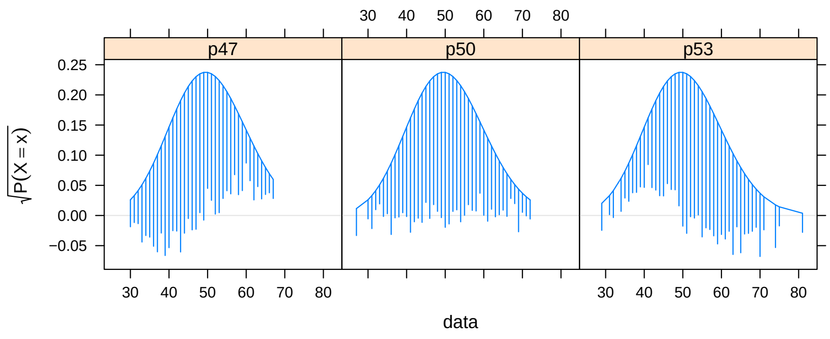

Figure 7: Hanging rootograms (Tukey, 1977) comparing the frequency distribution of simulated Poisson random variates, with mean 47, 50, and 53, to the theoretical Poisson(50) distribution. A good fit is indicated by the "hanging" lines uniformly ending near the vertical origin, as in the middle panel. This is an example of a new display function, complete with the usual features expected in Trellis graphics, implemented using the infrastructure provided by the lattice package.

```
> library(hexbin); data(NHANES)
> hexbinplot(Hemoglobin ~ TIBC | Sex, data = NHANES,
+            aspect = 0.85, type = "g")
```

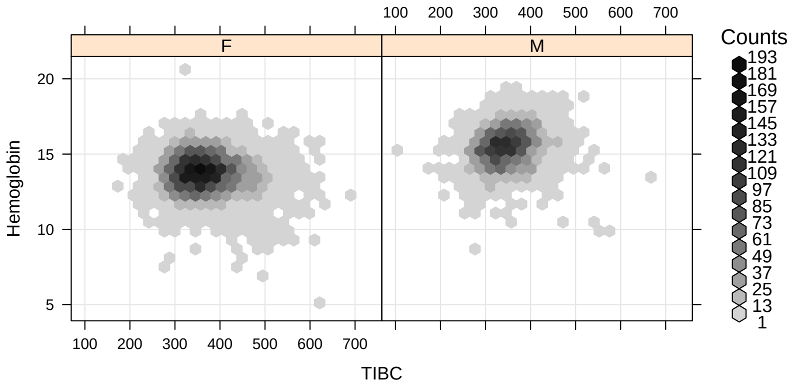

Figure 8: A conditional plot implementing the hexagonal binning algorithm of Carr et al. (1987). This example is somewhat challenging for the Trellis model, as it requires the panels to communicate information regarding bin counts to the legend.

3 Summary

We have presented in this paper several examples illustrating how the new generic-method system introduced in lattice allows the development of new intuitive, user friendly interfaces. With careful coding practices, such methods can handle large and complex data structures. In most cases, these new functions can piggyback on existing high level functions, and rarely require the use of unexported lattice utilities (in fact, such need should be considered a flaw in the design of lattice and reported as such). There are of course other, somewhat different, approaches that extend lattice (notably the nlme and Hmisc packages), which may serve as better models in specific situations.

One important aspect still missing is an extensible graphical settings system that allows developers to introduce new graphical parameters that can be manipulated transparently using the standard interfaces available in lattice for that purpose. Hopefully, this shortcoming will be addressed at some future date.

References

D. B. Carr, R. J. Littlefield, W. L. Nicholson, and J. S. Littlefield. Scatterplot matrix techniques for large *N*. *Journal of the American Statistical Association*, 82:424–436, 1987.

Robert C. Gentleman, Vincent J. Carey, Douglas M. Bates, et al. Bioconductor: Open software development for computational biology and bioinformatics. *Genome Biology*, 5:R80, 2004. URL `http://genomebiology.com/2004/5/10/R80`.

R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2007. URL `http://www.R-project.org`. ISBN 3-900051-07-0.

Charles J. Stone, Mark H. Hansen, Charles Kooperberg, and Young K. Truong. Polynomial splines and their tensor products in extended linear modeling: 1994 Wald memorial lecture. *The Annals of Statistics*, 25(4):1371–1470, 1997.

John W. Tukey. *Exploratory Data Analysis*. Addison-Wesley Publishing Co Inc, 1977.