Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003)
March 20–22, Vienna, Austria *ISSN 1609-395X*
Kurt Hornik, Friedrich Leisch & Achim Zeileis (eds.)
http://www.ci.tuwien.ac.at/Conferences/DSC-2003/

Some Notes on `lattice`

Deepayan Sarkar

Abstract

Trellis Graphics is implemented in S-PLUS using the traditional S graphics engine. In contrast, `lattice`, the implementation of Trellis Graphics in R, uses `grid` graphics as the underlying mechanism. Being an independent implementation, `lattice` differs from the original in several other aspects as well. Most of the literature available on Trellis Graphics describes the implementation in S-PLUS. This paper attempts to supplement that material by giving an overview of the major differences between the two.

In section 2 we discuss the benefits of using `grid` and provide some examples illustrating its usefulness. In section 3 we discuss some differences between the Trellis and `lattice` user level API's. An elementary knowledge of Trellis Graphics is assumed.

1 Introduction

Trellis Graphics, a prominent feature of S-PLUS, had been missing from R until recently. One of the reasons for this was that the standard R graphics system is not very well suited to producing Trellis-like graphs. The `grid` graphics package (Murrell, 2002) provides several powerful features which make a clean implementation of Trellis much easier.

The use of `grid` has its disadvantages, stemming from the fact that the traditional graphical primitives in S don't work in `grid`. For example, code written for S-PLUS may not always work unchanged with `lattice`. Section 2 discusses why despite this, `grid` is a better choice as the graphics engine underlying `lattice`.

The high level functions in `lattice` are modeled on the ones in S-PLUS. These are described quite thoroughly in several papers available at Bell Labs' web site (e.g., Becker, Cleveland, and Shyu (1996)). Although the use of `grid` makes some differences unavoidable, the description of Trellis in S-PLUS is applicable for the most part to `lattice` as well.

However, apart from the differences made unavoidable by the use of `grid`, there are also some design aspects in which `lattice` consciously deviates from Trellis, mostly in the form of extensions to the Trellis API. These could be useful to users

coding exclusively for R, but will often lead to incompatibilities with Trellis. In section 3, we give examples and brief justification for some of these extensions.

2 The role of `grid` in `lattice`

Trellis Graphics is different from traditional S graphics in many ways. It uses a formula interface to specify the form of a plot, it has a completely new system for customizing parameters, and of course, it has in-built the functionality to handle an arbitrary number of conditioning variables. But none of this is any more difficult to implement than it should be with traditional S graphics.

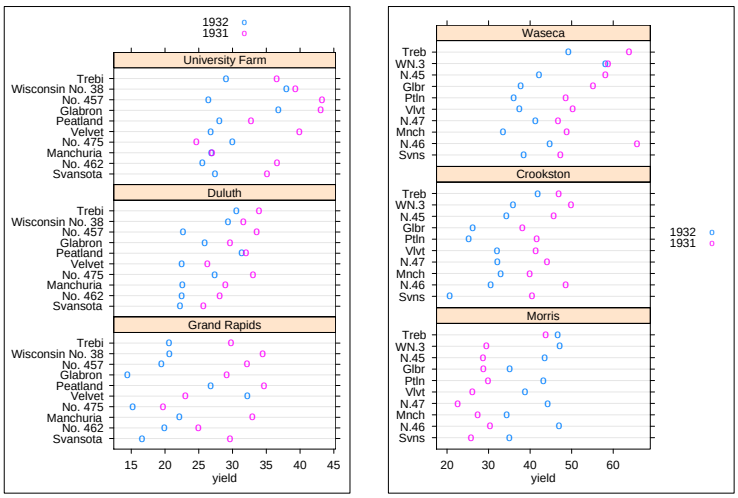

Figure 1: Two lattice plots, produced by essentially identical calls, except for different placement of the key, and abbreviated labels for the y-axis in the right hand plot.

However, Trellis Graphics is different from traditional S graphics in another aspect: it always tries to make as much use as possible of the available plotting space. Space will be allocated for a key (a.k.a. legend) or a title only if there is one. Exactly that much space will be used for tick marks and axis annotation as is required. Consider the two plots in Figure 1, representing two halves of the barley data set (Cleveland, 1993). Both are drawn inside `grid` viewports of identical dimension, with almost identical calls. They differ only in the location of the key, and in that the Y-axis labels have been abbreviated in the second plot. Notice that in both cases, every component of the graph except the actual Dot Plots inside the panels uses the minimum space necessary, allowing the panels to expand to use the remaining space. This is something that would be very difficult to get right without `grid`.

2.1 Why is this difficult to implement ?

When using the traditional S graphics engine, a graphics device (conceptually) consists of a plotting region, along with figure margins used for the title, axis labels,

annotations, etc. The dimensions of these margins can be controlled via the `par()` function. In the case of Trellis Graphics, this would need to be adjusted dynamically by the plotting function. A more sophisticated alternative, more suited to the other demands of Trellis Graphics, is the `layout()` function. Whatever the mechanism, to dynamically adjust the dimensions, we need to determine those dimensions in a form that would be understood by `par` or `layout`. Unfortunately, the figure margins can be only specified either as the number of lines of text or inches in `par`, and in centimetres in `layout`.

Consider the problem of determining the amount of space required for the key. The key itself can be quite sophisticated. It could consist of an arbitrary combination of columns of text, lines, rectangles or points, possibly divided into blocks. The width of each of these columns can be controlled separately, and the widths and heights of the text components of course would depend on the text itself. Further complications arise when these texts are expressions. Technically, of course, it is possible to obtain the physical dimensions of each of these components. The `strwidth()` and `strheight()` functions, for instance, can return widths and heights in inches. However, dealing with these would clearly be quite messy.

2.2 How does `grid` help ?

`grid` allows a tremendously flexible mechanism for specifying lengths. To see how this becomes useful, let's delve a little into how `lattice` plots are produced.

A `lattice` plot is always drawn inside a grid viewport, which can be an arbitrary rectangular region on the plotting device. Once the layout (arrangement of the panels) is decided, the first thing that `print.trellis` (the `lattice` function responsible for the actual plotting) does is to create a layout using `grid.layout`. This layout is at the the heart of the actual display, and consists of rows and columns that correspond to different components of the graph.

To take a simple case, consider Figure 2. A simplified description of the rows in the layout that created this plot is given in Table 1.

Component	height	units	data
Main Label	1.2	"strheight"	"Iris Data"
Key	1	"grobheight"	'Grob' from draw.key()
Tick labels	0	"lines"	
Ticks marks	.5	"lines"	
Panel	1	"null"	
Tick marks	.5	"lines"	
Tick labels			*See text*
X Label	1	"strheight"	"Petal\nLength"
Key	0	"lines"	
Sub Label	0	"lines"	

Table 1: A slightly simplified description of the rows that go into the `grid` layout for Figure 2, top to bottom. The height of the tick labels is more complicated, being the maximum of several "strheight" units.

Note that several rows are for components that are actually missing from the plot, the heights for these set to 0 "lines", and so do not occupy any space. There are rows for the key both at the top and the bottom, but only the top one is used,

```
> data(iris)
> print(xyplot(Sepal.Length ~ Petal.Length, data = iris,
+     groups = Species, auto.key = list(columns = 3),
+     main = list("Iris Data", cex = 1.2), xlab = "Petal\nLength",
+     ylab = "Sepal\nLength"))
```

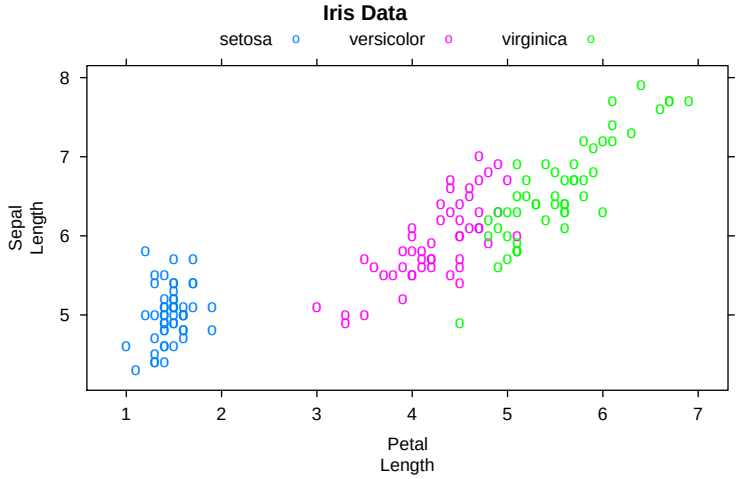

Figure 2: A simple plot using the Iris Data.

for which the height is specified simply as 1 "grobheight" – i.e., just enough space to hold the 'grob' (`grid` graphics object) specified as 'data'. In this case, this 'grob' is produced by an independent function called `draw.key()` which produces the key. Thus, the `lattice` code never has to explicitly deal with the physical dimensions of the key.

The `draw.key()` function itself uses another powerful feature of `grid` – the ability of a "grob" to pack within itself several smaller "grob"s. The "grob" that is finally returned by `draw.key()` is initially created as a "frame" (`grid.frame()`) object with a specified layout. Smaller "grob"s are then created – lines, rectangles, text, etc – and placed inside the frame (`grid.place()`).

`xlab`, the label for the x variable, is an example of how the "strheight" unit is used. In this case, this is a string containing a newline escape sequence. In general it could be a more complicated mathematical expression. `print.trellis()` doesn't need to care what it is and how many inches it occupies, it simply delegates that problem to `grid`.

2.3 Example: a rotated histogram

The decision to use `grid` brings with it the ability to do some useful things that would not have been otherwise possible. Although it's not very common, one might want to obtain a side-up histogram. Rather than writing a new function or modifying an existing one, simply rotating the viewport in which the plot is made produces the required plot (Figure 3)

```
> push.viewport(viewport(w = 2/3, angle = -90))
> print(histogram(rnorm(500), ylab = NULL, xlab = NULL,
+     scales = list(x = list(rot = 90), y = list(rot = 90,
+         alternating = 2))), newpage = FALSE)
> invisible(pop.viewport())
> grid.text(lab = "i.i.d. Normal Variates", rot = 90,
+     vp = viewport(x = 1/6))
```

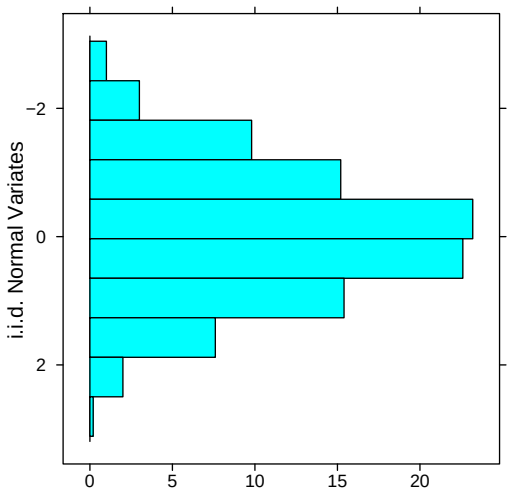

Figure 3: Rotated histogram of a Normal random sample.

2.4 Example: fixed scale plots

This example features a relatively recent addition to `lattice`, motivated by an email exchange with Federico Calboli who was trying to produce a LOD scores plot for QTL mapping data, similar to those in Macdonald and Goldstein (1999). The overall figure consists of 3 individual plots, each with one panel. The catch was that the 3 plots had different ranges on the X-axis, but were to have the same scale — i.e., one inch should correspond to the same increment on each plot. This was easy to achieve approximately, by placing the plots in regions that had width proportional to the X ranges. However, it was impossible to exactly account for the labels and tick marks.

To solve this, the `print` method for "trellis" objects now allows the user to control the exact dimensions of the panels. In the layout for a `lattice` plot, the rows and columns that correspond to the panels are usually assigned "null" units, which allows them to expand to fill the plotting regions. This has been generalised to allow the user to specify these dimensions explicitly in terms of `grid` units. Figure 4 was obtained by specifying the widths of each panel in millimeters, proportional to the X-axis limits.

```
> print(xplot, position = c(0,0,0.27,1), more = TRUE,
+       panel.width = list(58/2, "mm"))
> print(c2.plot, position = c(0.24,0,0.62,1), more = TRUE,
+       panel.width = list(99.51/2, "mm"))
> print(c3.plot, position = c(0.59,0,1,1),
+       panel.width = list(103/2, "mm"))
```

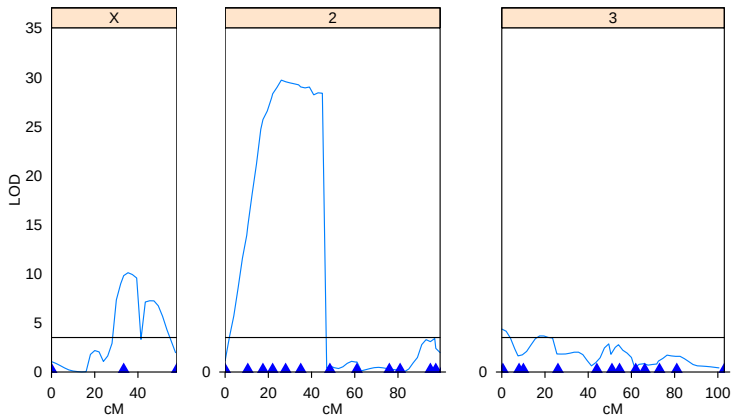

Figure 4: LOD scores plot: `xplot`, `c2.plot` and `c3.plot` are "trellis" objects representing 3 plots corresponding to 3 chromosomes of different lengths. The panel widths are specified explicitly, proportional to the X-axis limits, so that all 3 plots have the same scale. (This plot uses dummy data since the original work was unpublished at the time of writing.)

3 Some features of the `lattice` API

3.1 Grouped displays

Trellis Graphics in S-PLUS has all the basic support necessary for grouped displays. However, it has only one panel function — `panel.superpose()` — that allows grouped displays, and even that has to be explicitly specified when required. `lattice` takes the approach that whenever the `groups` argument is specified in a call, the display should be grouped by default, provided grouping makes sense in the context of that plot. Such functions include `xyplot, dotplot, stripplot, barchart, densityplot, cloud, wireframe` and `splom`. Figure 5 shows an example of this in a `barchart` call.

```
> print(barchart(yield ~ variety | year, data = barley,
+     groups = site, stack = TRUE, reference = TRUE,
+     ylab = "Barley Yield", auto.key = list(rect = TRUE,
+         points = FALSE, columns = 3),
+         scales = list(x = list(rot = 45))))
```

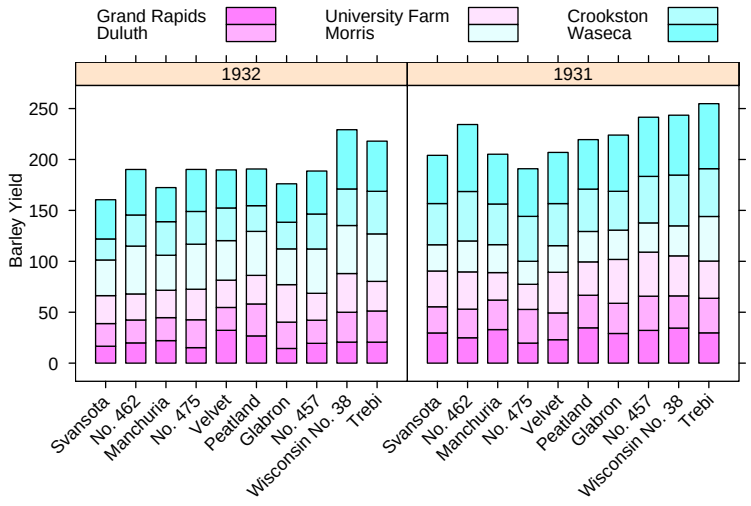

Figure 5: Stacked barchart for the barley data

One problem with the Trellis API for grouped displays is that it requires the data frame to be in the "long" format (to borrow terminology from the `reshape` function). Suppose we want Kernel Density Estimates for the Iris data, with separate panels for each Species. It is easy to obtain this for, say, the Sepal Length, with a call like

```
densityplot( ~ Sepal.Length | Species, iris)
```

But suppose we want density estimates for all four variables in each panel, differentiated by color or line type. `densityplot` can produce grouped displays, so this is not too difficult, but it requires the four data vectors to be concatenated into one long vector, and the creation of a new 'grouping variable'. This can all be done with `reshape()` as follows:

```
> iris2 <- reshape(iris,
+     varying = list(c("Sepal.Length", "Sepal.Width",
```

```
+                       "Petal.Length", "Petal.Width")),
+     timevar = "grpvar", direction = "long",
+     v.names = "length")
> print(densityplot(~ length | Species, iris2, groups = grpvar, ...
```

But that is an at best inconvenient, and at worst obscure option. `lattice` extends the Trellis `formula` interface to make this easier to code — by allowing more than one variable to be specified on either side of the ~ in the formula. Figure 6 illustrates the usage for the Iris example above.

```
> print(densityplot(~Sepal.Length + Sepal.Width +
+     Petal.Length + Petal.Width | Species, iris,
+     allow.multiple = TRUE, plot.points = FALSE,
+     xlab = "Measurements on 3 species of Iris",
+     scales = list(alternating = FALSE),
+     auto.key = list(line = TRUE, point = FALSE, columns = 4)))
```

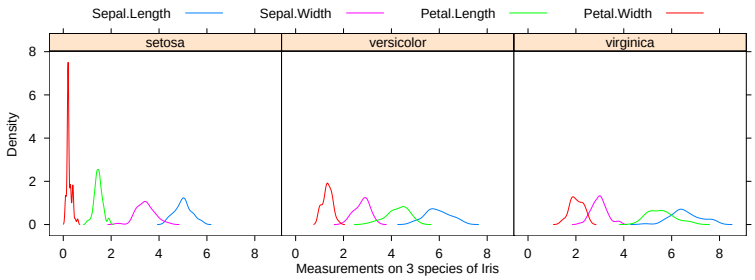

Figure 6: A grouped densityplot. The grouping is specified not in the usual way, but rather as multiple variables in the right hand side of the formula. Note that this needs the `allow.multiple` flag to be set, to ensure S compatibility.

3.2 `factors` in the Trellis `formula`

Trellis is picky about what sort of variables it allows in the formula. `bwplot`, `dotplot`, `barchart` and `stripplot`, for example, allow only the `y` variable to be a factor. `lattice` allows `x` to be the factor, which in fact enables the creation of the more conventional form of Box Plots and Bar Charts. A typical problem with such plots is that the tick labels (which for factors are its levels) are long and often overlap — `lattice` deals with this by allowing the labels to be rotated and/or abbreviated through the `scales` argument (see Figure 5 for an example).

Most other Trellis functions, including `xyplot`, `levelplot` and `cloud` require all variables to be numeric. While this is consistent with the nature of these plots, it is often appropriate to use categorical variables there, albeit after coercing them to be numeric. However, we would still want the levels of the factor to provide tick mark positions and labels. One such example is given in Figure 7.

It is of course possible to do this in Trellis by explicitly coercing the factors to be numeric, but then a lot of extra effort needs to go into manually specifying the tick mark positions and labels to something meaningful.

`lattice` takes the approach that any variable can be anything when specified in the `formula` of a high-level function call, and the nature of this variable will

```
> print(levelplot(yield ~ year * variety | site,
+     barley, aspect = "xy", shrink = c(0.5, 0.9),
+     main = "Barley Yield (bushels / acre)",
+     col.regions = terrain.colors(50),
+     scales = list(alternating = FALSE)))
```

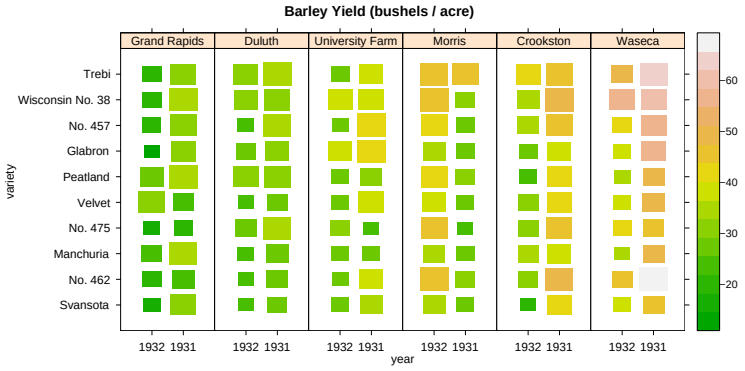

Figure 7: A levelplot (with the size of the rectangles scaled to reflect the value of the `z` variable) of the barley data set, with categorical variables on the X and Y axes. The Morris anomaly (an increase in the yield from 1931 to 1932, the opposite of the trend in all the other sites) is easily identifiable. Figure 1 is a more familiar representation of this data.

determine the tick positions and labels. The decision on how to plot this data is delegated to the `panel` function. Unfortunately, this introduces an incompatibility with Trellis (which assumes that `prepanel` and `panel` functions are always passed numeric arguments). Hopefully, the benefits outweigh the problems in this case.

4 Conclusion

In this paper, we have gone over some aspects in which `lattice` differs from Trellis. We have discussed briefly how `grid` graphics is used to implement `lattice` plots and seen examples where `grid` allows us to easily implement new features. We have also discussed how `lattice` treats grouped displays and factors, which is somewhat different from what Trellis does.

These are not by any means the only differences. As mentioned earlier, the need to use `grid` graphics primitives instead of base functions inside panel functions leads to some incompatibilities. `lattice` also provides some convenience functions that provide simpler interfaces for the customization of graphical parameters, creating legends, etc. The other differences mostly concern predefined panel functions, which often have more features than their Trellis counterparts. Details about these are easily found in the online help pages.

Concerning the code provided here, it should be noted that `lattice` is still evolving, and some details may change over time. The documentation accompanying the package should be consulted if any of the code stops working in future versions.

References

Richard A. Becker, William S. Cleveland, and Ming-Jen Shyu. The visual design and control of Trellis display. *Journal of Computational and Graphical Statistics*, 5:123–155, 1996. URL http://cm.bell-labs.com/cm/ms/departments/sia/project/trellis/display.writing.html.

William S. Cleveland. *Visualizing Data*. Hobart Press, 1993.

Stuart J. Macdonald and David B. Goldstein. A quantitative genetic analysis of male sexual traits distinguishing the sibling species drosophila simulans and d. sechellia. *Genetics*, 153:1683–1699, 1999.

Paul Murrell. The `grid` graphics package. *R News*, 2(2):14–19, 2002.

Affiliation

Deepayan Sarkar
Department of Statistics
University of Wisconsin – Madison
E-mail: deepayan@stat.wisc.edu