```
  viewport(layout.pos.row=4,
           xscale=xrange,
           yscale=c(0, max(cardata$total))))
grid.lines(unit(1:n, "native"),
           unit(cardata$total, "native"))
grid.yaxis()
pop.viewport()
draw.workday(5)
```

Some important points about this example are:

1. It is not impossible to do this using R's base graphics, but it is more "natural" using **grid**.

2. Having created this graphic using **grid**, arbitrary annotations are possible — all coordinate systems used in creating the unusual arrangement are available at the user-level for further drawing.

3. Having created this graphic using **grid**, it may be easily embedded in or combined with other graphical components.

Final remarks

The most obvious use of the functions in **grid** is in the development of new high-level statistical graphics functions, such as those in the **lattice** package. However, there is also an intention to lower the barrier for normal users to be able to build everyday graphics from scratch.

There are currently no complete high-level plotting functions in **grid**. The plan is to provide some default functions for high-level plots, but such functions inevitably have to make assumptions about what the user wants to be able to do — and these assumptions inevitably end up constraining what the user is able to achieve. The focus for **grid** will continue to be the provision of as much support as possible for producing complex statistical graphics by combining basic graphical components.

grid provides a number of features not discussed in this article. For information on those features and more examples of the use of **grid**, see the documentation at http://www.stat.auckland.ac.nz/~paul/grid/grid.html. **grid** was also described in a paper at the second international workshop on Distributed Statistical Computing (Murrell, 2001).

Bibliography

Bell lab's trellis page. URL http://cm.bell-labs.com/cm/ms/departments/sia/project/trellis/.

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. 17

William S. Cleveland. *Visualizing data*. Hobart Press, 1993. ISBN 0963488406. 17

Paul Murrell. R Lattice graphics. In Kurt Hornik and Friedrich Leisch, editors, *Proceedings of the 2nd International Workshop on Distributed Statistical Computing, March 15-17, 2001, Technische Universität Wien, Vienna, Austria*, 2001. URL http://www.ci.tuwien.ac.at/Conferences/DSC-2001/Proceedings/. ISSN 1609-395X. 19

Murrell, Paul R. Layouts: A mechanism for arranging plots on a page. *Journal of Computational and Graphical Statistics*, 8:121–134, 1999. 16

Paul Murrell
University of Auckland, NZ
paul@stat.auckland.ac.nz

Lattice

An Implementation of Trellis Graphics in R

by Deepayan Sarkar

Introduction

The ideas that were later to become Trellis Graphics were first presented by Bill Cleveland in his book *Visualizing Data* (Hobart Press, 1993); to be later developed further and implemented as a suite of graphics functions in S/S-PLUS. In a broad sense, Trellis is a collection of ideas on how statistical graphics should be displayed. As such, it can be implemented on a variety of systems. Until recently, however, the only actual implementation was in S-PLUS, and the name Trellis is practically synonymous with this implementation.

lattice is another implementation of Trellis Graphics, built on top of R, and uses the very flexible capabilities for arranging graphical components provided by the **grid** add-on package. **grid** is discussed in a companion article in this issue.

In keeping with the R tradition, the API of the high-level Lattice functions are based on published descriptions and documentation of the S-PLUS Trellis Graphics suite. It would help to remember, however, that while every effort has been made to enable Trellis code in S-PLUS to run with minimal modification, Lattice is different from Trellis in S-PLUS in

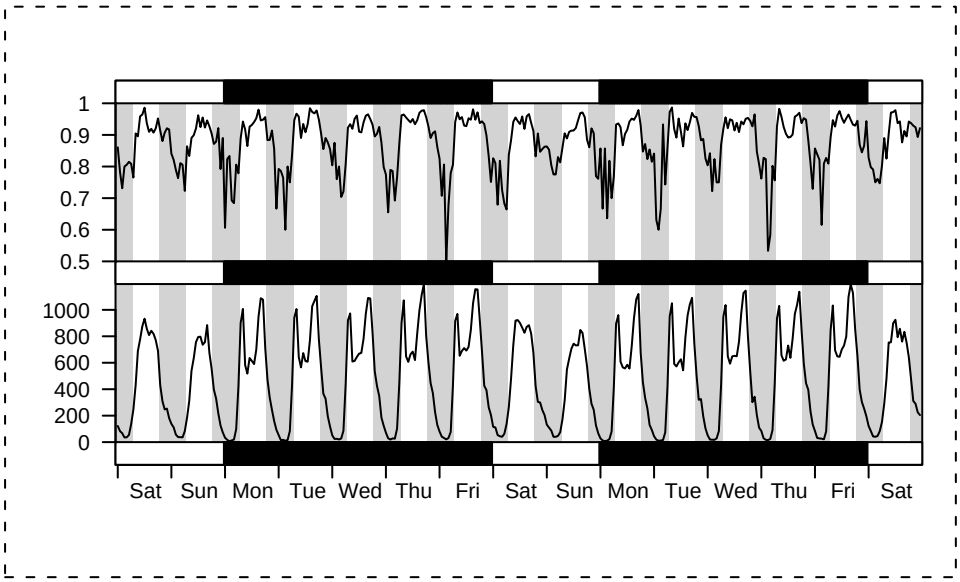

Figure 6: A custom plot produced using **grid**.

many respects; some unavoidable, some intentional.

The Trellis paradigm

Familiarity with some general principles Trellis uses in displaying statistical graphs can go a long way towards understanding it. The most important feature of Trellis Graphics is the ability to produce *multi-panel, multi-page* graphs — this was in fact the primary motivation (as well as the origin of the name) for Trellis. Perhaps no less important, however, is a *basic paradigm shift* from the way graphics is formulated and displayed conventionally in the S languages.

Conventional S graphics draws plots incrementally. For example, it is common to draw a histogram, then draw a box around it, add the axes, and perhaps add a few labels (`main`, `sub`, etc) to get the final plot. Trellis, on the other hand, would first create an object that would contain all the information necessary to create the plot (including what to draw, how to mark the axes, what `main`, `sub` are, if anything), and then plot it in one go.

This approach has certain advantages. For conventional S graphics, there is no way of knowing in advance whether the axes will be drawn, whether there will be a main label (and if so, how long or how large it would be), etc. Consequently, when the first plot is drawn, there has to be enough space left for everything else, in case the user decides to add these later. This is done by pre-allocating space for axis labels, main labels, etc. Obviously, the default amount of space is not always appropriate, and it is often necessary to change these spacings via the `par()` function. Trellis plots, however, take a different approach. A plot is drawn all at once, and thus exactly the requisite amount of space can be used for each component of the display.

Also unlike conventional S graphics, high level Lattice functions do not directly draw anything when they are called, but rather, they are more like other S functions in that they produce an object (of class `"trellis"`) which when `print`ed produces the actual plot. This enables changing several aspects of the plot with minimal recomputation via the `update.trellis` method, as well as the ability to save a plot for later plotting and draw the same plot in several devices. The `print` method for `"trellis"` objects also allows finer control over the placement of plots within the plotting region, in turn making possible more than one Trellis plot per page. This is similar to setting `par(mfrow)`, but slightly more flexible.

On the downside, this holistic approach to plotting can make calls to produce even slightly complicated plots look very intimidating at first (although the default display options are quite satisfactory for the most part, the need to change these also arises quite frequently).

Getting started

A few high level functions provide the basic interface to Lattice. Each of these, by default, produce a different kind of statistical graph. Scatter Plots are produced by `xyplot`, Box Plots and Dot Plots by `bwplot` and `dotplot`, Scatter Plot Matrices by `splom`, Histograms and Kernel Density Plots by `histogram` and `densityplot`, to name the most commonly used ones. These functions need, at a minimum, only one argument: a formula specifying the variables to be

used in the plot, and optionally, the name of a data frame containing the variables in the formula.

The `formula` argument and conditioning

Unlike most conventional graphics functions, the variables in the plot are almost always specified in terms of a formula describing the structure of the plot. This is typically the unnamed first argument. For bivariate functions like `xyplot`, `bwplot`, etc, it is of the form `y~x | g1 * ··· * gn`. Here `x`, `y`, `g1`, ..., `gn` should be vectors of the same length, optionally in a data frame, in which case its name has to be given as the `data` argument.

The variables `g1`, ..., `gn` are optional, and are used to investigate the relationship of several variables through conditioning. When they are absent, the plots produced are similar to what can be done with conventional S graphics. When present, we get Conditioning Plots — separate 'panels' are produced for each unique combination of the levels of the conditioning variables, and only the subset of `x` and `y` that corresponds to this combination is used for displaying the contents of that panel. These conditioning variables are usually factors, but they can also be *shingles*, an innovative way of conditioning on continuous variables. Conditioning is illustrated in (the left part of) Figure 3, where dotplots of yields of different varieties of barley are displayed in different panels determined by the location of the experiment.

Conditioning can be done in all high level functions, but the form of the first part of the formula changes according to the type of plot produced. The typical form is `y ~ x`, where the `y` variable is plotted on the vertical axis, and `x` on the horizontal. For univariate functions like `histogram` and `densityplot`, where only the `x` variable is needed, the form of the first part is `~x`, and for trivariate functions like `cloud` and `levelplot` whose panels use three variables, the form is `z ~ x * y`.

Aspect ratio and banking

The information that the human eye can perceive from a plot can change significantly depending simply on the aspect ratio used for displaying the plot. Some Trellis functions can use 'banking' to automatically select a close to optimum aspect ratio that makes the plot more informative, when given the `aspect="xy"` argument. Figure 3 includes an example illustrating the usefulness of this feature.

Changing panel functions using grid

While the high level functions provide the ability to produce the most commonly used statistical graphs, it is often necessary to go beyond that and produce displays tailored for the problem at hand. Much of the power of Trellis comes from the ability to change the default display by means of a function supplied as the `panel` argument. This can be any function written using **grid**, and is called with the panel region set as the current viewport.

A word of caution: Conventional R graphics code will not work in the panel function. However, many predefined functions that can serve as building blocks of custom panel functions are already included in the Lattice package. (These usually suffice for *porting* Trellis code written for S-PLUS.)

An example below illustrates how to modify a call to `bwplot` to produce an *interaction plot*:

```
library(nlme)
data(Alfalfa)
levels(Alfalfa$Date) <-
    c('None', 'Sep 1', 'Sep 20', 'Oct 7')
bwplot(Yield ~ Date | Variety, Alfalfa,
       groups = Block, layout = c(3, 1),
       panel = "panel.superpose",
       panel.groups = "panel.linejoin",
       xlab = "Date of third cutting",
       main = "Block by Date interaction
               in 3 Varieties of Alfalfa")
```

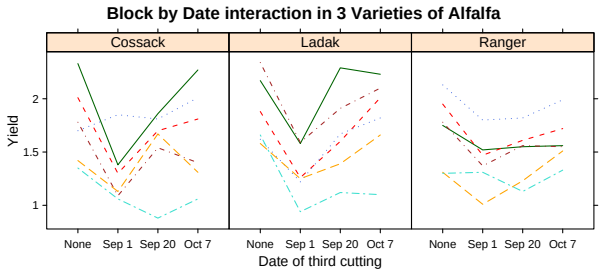

Figure 1: Interaction plot produced by `bwplot` with a custom panel function

A very useful feature of Lattice is that all arguments not recognized by high level functions are passed on to the panel function. This allows very general panel functions to be written, which can then be controlled by arguments given directly to the high level function.

Other commonly used features

All high level Lattice functions accept several arguments that modify various components of the display, some of which have been used in the examples given here. The most commonly used ones are briefly mentioned below, further details can be found in the documentation.

Panel Layout: When using conditioning variables, especially when a large number of panels are produced, it is important to arrange the panels in a nice and informative manner. While the default layout is usually satisfactory, finer control can be achieved via the `layout`, `skip`, `between` and `as.table` arguments.

Scales: An important component of the display is how the axis tick marks and labels are shown. This is controlled by the `scales` argument (`pscales` for `splom`).

Grouping variable: Apart from conditioning, another very common way to look at the relationship between more than two variables, particularly when one of them is a factor with a small number of levels, is by using different graphical parameters to distinguish between levels of this factor. This can be done using the `groups` argument, usually with `panel.superpose` as the panel function.

Handling multiple plots

The `print.trellis` function, when called explicitly with extra arguments, enables placing more than one Lattice plot on the same page. For example, the following code places two 3d scatter plots together to form a stereogram (figure 2):

```
data(iris)
print(cloud(Sepal.Length ~
            Petal.Length * Petal.Width,
            data = iris, perspective = FALSE,
            groups = Species,
            subpanel = panel.superpose,
            main = "Stereo",
            screen = list(z=20,x=-70,y=3)),
      split = c(1,1,2,1), more = TRUE)
print(cloud(Sepal.Length ~
            Petal.Length * Petal.Width,
            data = iris, perspective = FALSE,
            groups = Species,
            subpanel = panel.superpose,
            main = "Stereo",
            screen = list(z=20,x=-70,y=0)),
      split = c(2,1,2,1))
```

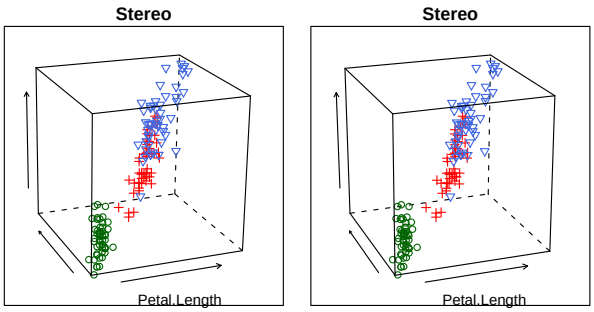

Figure 2: 3d Scatter plots (using `cloud`) of the Iris data, grouped by Species. Stare at the plot and focus your eyes behind the page so that the two images merge. The result should be the illusion of a 3D image.

Settings: Modifying the look and feel

The choice of various parameters, such as color, line type, text size, etc can change the look and feel of the plots obtained. These can be controlled to an extent by passing desired values as arguments named `col`, `lty`, `cex`, etc to high level functions. It is also possible to change the settings globally, by modifying a list that determines these parameters. The default settings are device specific — screen devices get a grey background with mostly light colors, while postscript output has a white background with darker colors. It is possible to change these settings, and to define entirely new 'themes' suited to particular tastes and purposes — for example, someone using the prosper package to create presentations in LaTeX might want to set all the foreground colors to white and use larger text. The functions `trellis.device`, `trellis.par.set`, `lset` and `show.settings` can be useful in this context. Lattice currently comes with one predefined theme that can be loaded by calling `lset(col.whitebg())`, and it is fairly easy to create new ones.

Using Lattice as a tool

Trellis was designed for a specific purpose, namely, to provide effective graphical presentation of the relationship between several variables through conditioning; and that is what it does best. It is a high level graphics package, and makes several assumptions about the form of the display it produces. In that respect, it is not always suitable as a tool for developing new graphical methods. It is quite flexibile in its own way, however, and can be particularly useful for creating custom display functions for specific types of data — the pkgnlme package, for example, uses Lattice functions extensively for its `plot` methods.

Conclusion

This article provides a brief overview of the functionality in the Lattice package, but is by no means a comprehensive introduction. For those interested in learning more, an excellent place to start is Bell Lab's Trellis webpage at http://cm.bell-labs.com/cm/ms/departments/sia/project/trellis/; although specific to the S-PLUS implementation, most of it is also applicable to Lattice. Gory details are of course available from the documentation accompanying the Lattice package, `?Lattice` is the recommended launching point. Finally, Lattice has its own webpage at http://packages.r-project.org/lattice/.

Deepayan Sarkar
University of Wisconsin, U.S.A.
deepayan@stat.wisc.edu

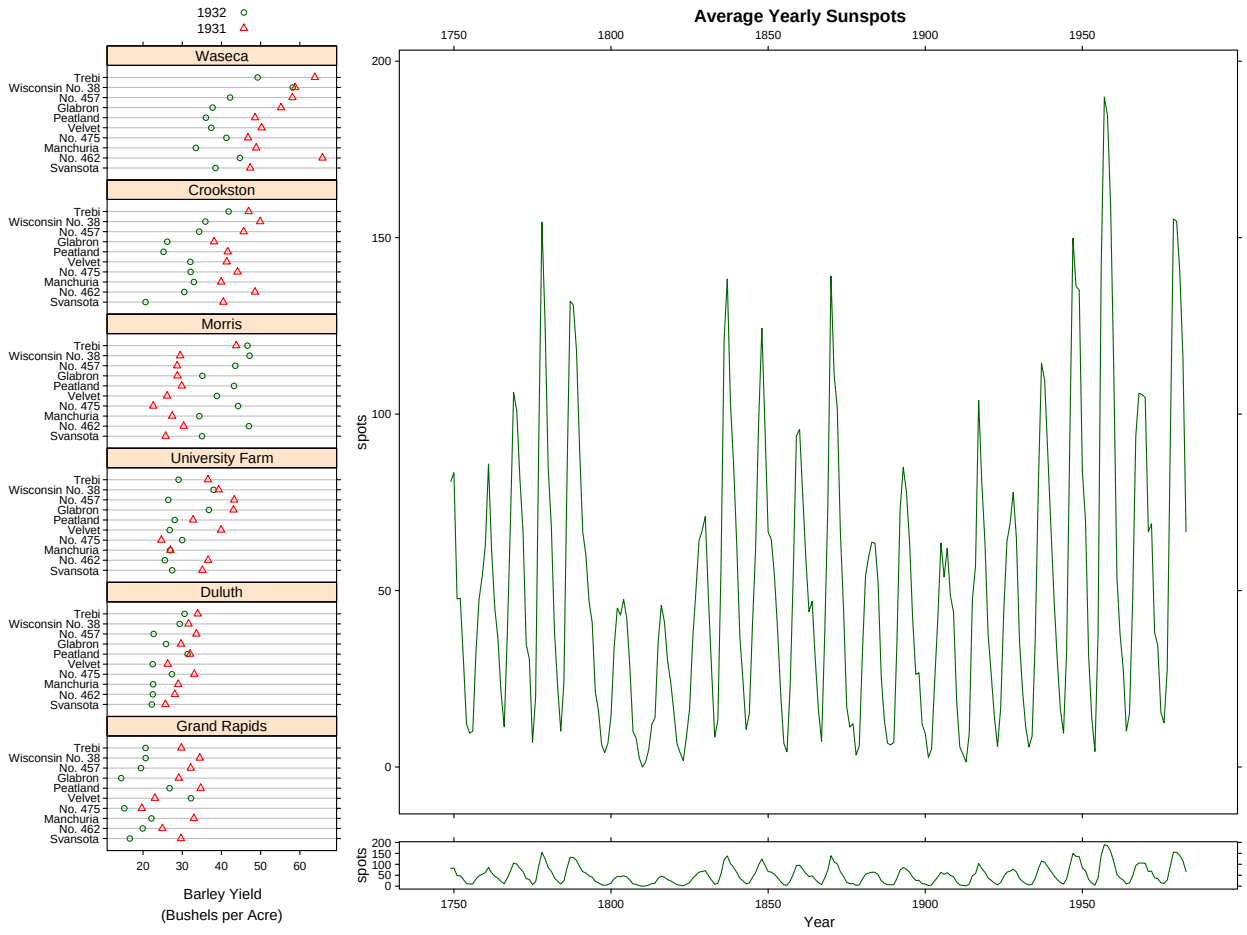

Figure 3: Two well known examples from Cleveland (1993). The left figure shows the yield of 10 different varieties of barley in 6 farms recorded for two consecutive years. The data has been around since Fisher, but this plot suggested a possible mistake in recording the data from Morris. The plots on the right show the usefulness of banking. The top figure is what an usual time series plot of the average yearly sunspots from 1749 to 1983 would look like. The bottom plot uses banking to adjust the aspect ratio. The resulting plot emphasizes an important feature, namely, that the ascending slope approaching a peak is steeper than the descending slope. [This plot was created using `print.trellis` for the layout. See accompanying code for details. The colour schemes used here, as well as in the other plots, are not the defaults. See the section on **Settings** for references on how to set 'themes'.]

```
data(barley)
plot1 <- dotplot(variety ~ yield | site, data = barley, groups = year, aspect = .5,
                 panel = "panel.superpose", panel.groups = "panel.dotplot", layout = c(1, 6),
                 col.line = c("grey", "transparent"), xlab = "Barley Yield\n(Bushels per Acre)",
                 key = list( text = list(c("1932", "1931")),
                 points = Rows(trellis.par.get("superpose.symbol"), 1:2)))
data(sunspots)
spots <- by(sunspots, gl(235, 12, lab = 1749:1983), mean)
plot2 <- xyplot(spots~1749:1983, xlab = "", type = "l", main = "Average Yearly Sunspots",
                scales = list(x = list(alternating = 2)),
plot3 <- xyplot(spots~1749:1983, xlab = "Year", type = "l", aspect = "xy")
print(plot1, position = c(0, 0, .3, ,1), more = TRUE)
print(plot2, position = c(.28, .12, 1, 1), more = TRUE)
print(plot3, position = c(.28, 0, 1, .13))
```