DSC 2001 Proceedings of the 2nd International Workshop on Distributed Statistical Computing
March 15-17, Vienna, Austria
http://www.ci.tuwien.ac.at/Conferences/DSC-2001
K. Hornik & F. Leisch (eds.) *ISSN 1609-395X*

R Lattice Graphics[*]

Paul Murrell[†]

Abstract

Lattice is an add-on package or library for the R statistical computing environment. It provides an alternative set of user-level functions for producing graphical output. Compared to the base R graphical functions, the Lattice functions provide greater control over the specification of where graphical output appears on the page. In addition, Lattice graphics functions return graphical objects, which may be used to interactively edit the graphical output.

1 What is Lattice ?

Lattice is an add-on library for the R statistical computing environment. It provides a set of user-level graphics functions as alternatives to the R base graphics functions such as `plot()`, `text()`, `points()`, etc for the construction of statistical graphics (plots).

2 Who needs Lattice ?

There were two main reasons for producing an alternative graphics system for R:

1. The original motivation for Lattice graphics was to provide support for the production of graphical displays similar to those produced by the Trellis graphics package in the S-Plus system (see Figure 1).

[*]After this talk was delivered at DSC 2001, it was decided that the package implementing Trellis-like functionality in R should take the name “lattice”. Consequently, the R package described in this paper will no longer be called “lattice”, but will take a new name, possibly “G2”.

[†]The University of Auckland

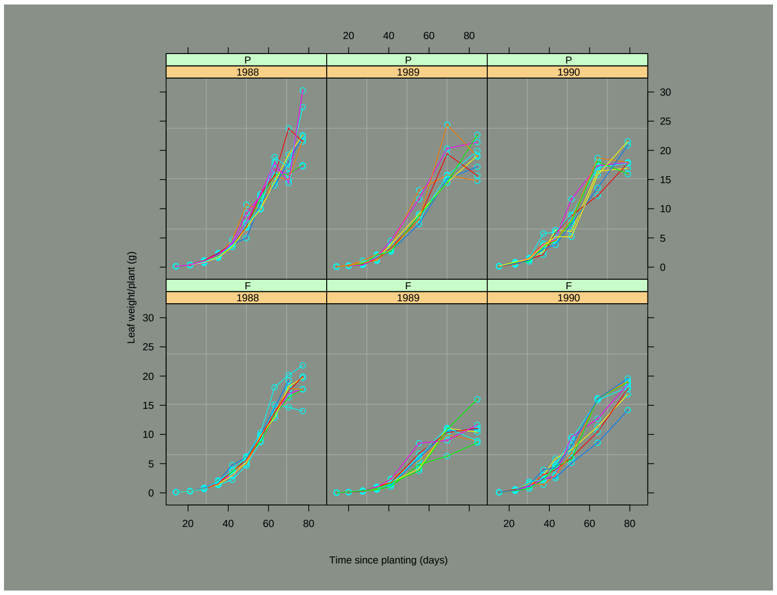

Figure 1: A Trellis-like plot.

These sorts of plots are difficult to produce using the base R graphics functions.

2. The base R graphics system does not provide support for interactive techniques such as selecting elements of a graph for editing, or brushing and linking plots. Lattice was designed to provide support for this sort of simple interaction.

3 What can Lattice Graphics do ?

The Lattice graphics package is designed to provide enormous power and flexibility for producing statistical graphics. The following sections describe the tools provided for the user.

3.1 Viewports

The base R graphics system is based on the concept of a plot region, surrounded by a margin. There may be multiple plots, each with their own margins, and all of them surrounded by a further "outer" margin (see Figure 2).

Lattice is based instead upon the concept of a generic drawing region. This will sometimes correspond to a plot region, where data symbols and lines are drawn, but it may also correspond to an entire plot, a plot margin, a legend, a collection of plots, or just a data symbol.

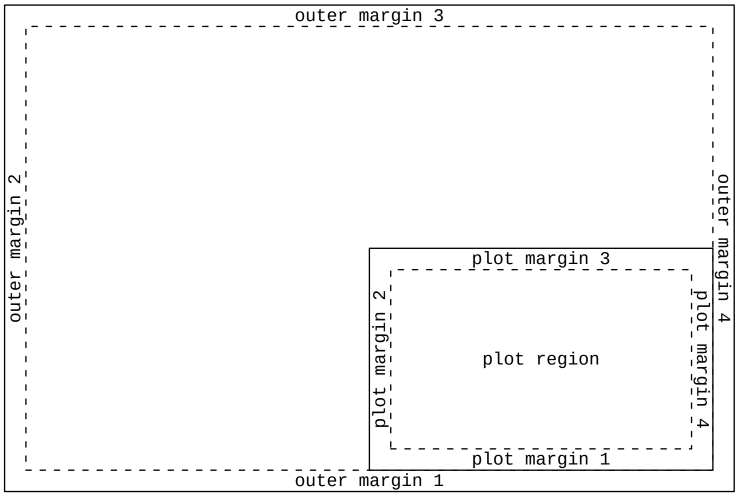

Figure 2: The base R graphics system.

A Lattice *viewport* is a rectangular region with a user-defined location, justification, and size. For example, Figure 3 shows the region defined by the command:

```
lviewport(x=unit(1, "npc") - unit(1, "inches"), y=0.5,
          width=0.2, height=0.5, just=c("right", "centre")))
```

3.2 Coordinate Systems and Units

The base R graphics functions are related to a specific coordinate system as well as to a specific task. For example, the `text()` function can be used to draw text within the plot region, relative to the coordinates of the x- and y-axes ("user coordinates"). On the other hand, the function `mtext()` produces text output in the margins of a plot (or in the outer margins) using a coordinate system relevant to the margins.

In Lattice, a large range of coordinate systems are available in **all** viewports. The user selects which coordinate system to use by specifying a *unit* for each value. For example, the following statement specifies a value of 1cm: `unit(1, "cm")`.

Some of the coordinate systems available are:

Normalised Parent Coordinates (NPC): the origin of the viewport is at $(0, 0)$ and the viewport has a width and height of 1. This is useful for things like centering objects. For example, the location $(0.5, 0.5)$ in NPC is always the centre of the viewport.

Physical Coordinates (inches, centimetres, ...): self-explanatory. These are useful for specifying objects of absolute size, for example, to produce a plot of a fixed size for inclusion in a document.

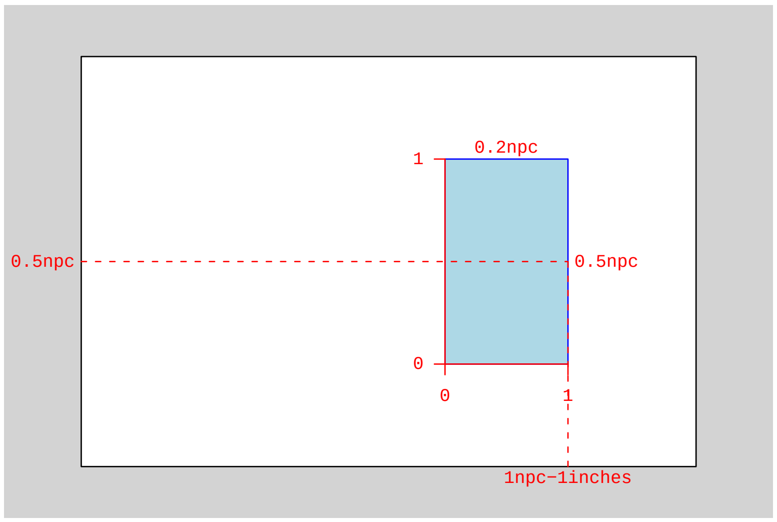

Figure 3: Diagram of a Lattice viewport.

Native Coordinates: All viewports have x- and y-scales. This coordinate system is relative to those scales. This is the coordinate system to use for drawing points and lines on a plot.

Character-based, Line-based, and String-Width/Height-based Coordinates: Values in these coordinate systems are multiples of the nominal font size, or the vertical distance between the baselines of two lines of text, or the width/height of a piece of text (respectively). These are useful for arranging pieces of text relative to each other, or other objects relative to text.

3.3 Layouts and Nesting Viewports

The wide selection of coordinate systems allows users to specify the location of a child object within a parent viewport. There is also support for the parent viewport to specify the location of its children.

The user is able to define a *layout* [1] for a viewport. This is essentially a division of the viewport into rows and columns of different sizes. Children of the viewport are allocated some subset of the cells within the layout. For example, Figure 4 shows a simple layout with three columns and four rows, with a viewport occupying the middle two rows of the central column.

The above example also demonstrates the fact that viewports can be nested within other viewports. In other words, it is possible to create a hierarchy of viewports. This is extremely useful because many (if not all) statistical graphics consist of a hierarchy of graphical objects. For example, a page may contain multiple plots, and each plot may consist of a plot region plus margins.

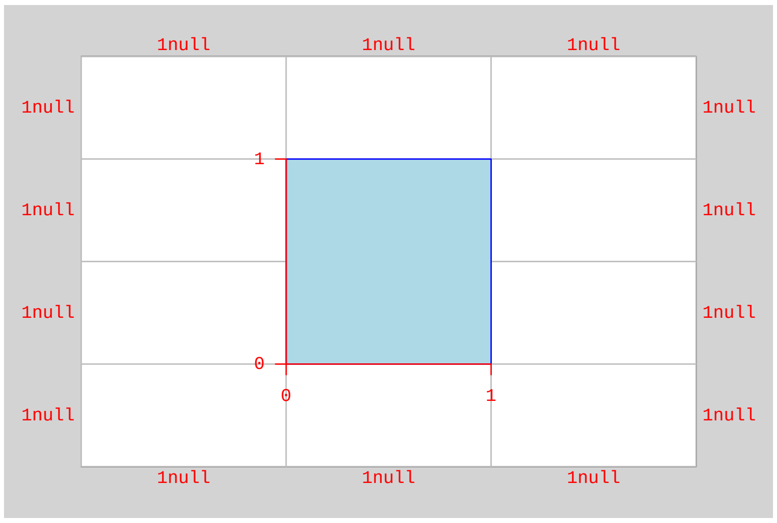

Figure 4: A Lattice viewport occupying cells within a Lattice layout.

Conceptually, this nesting allows the user to focus on the most convenient way of producing each component of a graphic without having to be concerned with the final location or size of the graphic. The classic example is a plot legend; this can be constructed within its own viewport, using convenient coordinates within that viewport such as line-based coordinates and string-width-based coordinates, and then the legend viewport can be positioned within a plot.

3.4 Interaction and Customisation

The value returned by all Lattice graphics functions is an R object representing the graphical output produced by the function. For example, the command `xa <- lxaxis()` draws an x-axis and returns an R object representing that axis. It is then possible to perform operations on the returned object. For example, the command `ledit(xa, at=c(1, 5, 9))` changes the location (and number) of the tick marks on the axis.

Some Lattice graphics objects have a hierarchical structure. For example, an axis contains multiple Lattice text objects representing the labels on the axes. It is important that the user can gain access to this structure so Lattice allows the user to specify sub-components of objects. For example, the command `ledit(xa, "labels", rot=45)` changes the angle of rotation of the labels of the x-axis.

3.5 Extensibility and Ease-of-use

Lattice provides the user with more powerful control over the graphics output, but this comes with additional complexity. Lattice has been designed so that the additional power and complexity it provides is optional; simple things are still simple to do.

Lattice graphics functions can be used simply to produce output; for example, the command `ltext("hi")` produces the text output "hi". With a little more work, the functions can be used simply to produce graphical objects for further manipulation; for example, the command `txt <- ltext("hi", draw=F)` produces no output, but assigns a text object to the variable `txt`.

As with the base R graphics system, Lattice is designed to be used programmatically by the user. That is, users are able to extend the system to produce new statistical graphics functions.

The user can write graphics functions which are designed to be used only for their graphical output. For example, the following function definition produces output, but does not return (all of) the objects associated with that output.

```
my.func <- function() {
    ltext("hi")
    lrect(width=unit(1, "strwidth", "hi"))
}
```

A little more work is required to write a function which returns a meaningful graphical object. For example:

```
my.func <- function() {
    txt <- ltext("hi")
    box <- lrect(width=unit(1, "strwidth", "hi"))
    lgrob(list(txt, box), "boxed.text")
}
```

3.6 Some Experimental Ideas

Two Lattice features are still at an early stage of development.

Rotated Viewports: It is possible to specify an angle of rotation for a viewport. Combined with the notion of designing components without having to worry about where they will end up, this makes it relatively simple to produce some quite interesting plots. For example, Figure 5 shows a scatterplot with a rotated boxplot indicating the spread of the vertical distances from the points to the line "$x = y$".

Frames and Packing: I have already described the Lattice support for having a child object specify its location within a parent viewport (coordinate systems) and the support for having a parent viewport specify the location of its children (layouts). In some cases, it is useful to have more of a "discussion" occur between parent and child. For example, when adding a legend to a plot, it is useful to be able to specify (from the parent) that the legend should be located to the right of the plot, but it is also useful to be able to consult the legend to determine how much width it requires.

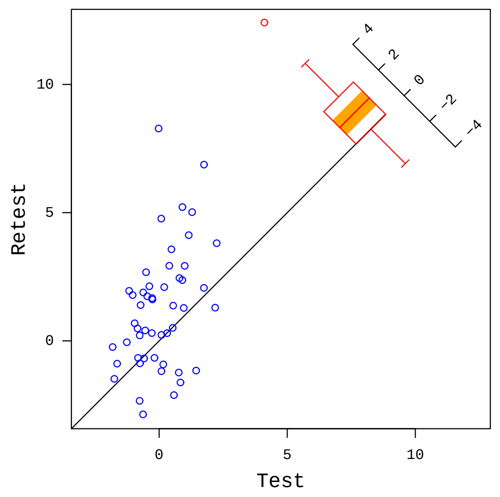

Figure 5: A plot using a rotated viewport.

This sort of problem is often encountered in the construction of a Graphical User Interface. The common solution in such situations is to define a parent *frame* and *pack* the children within the frame, effectively specifying an approximate location for each child, but consulting the child for the required size.

I have begun to develop this sort of an interface for constructing graphical objects in Lattice. For example, the following series of commands were used to produce Figure 6.

```
lf <- lframe()
lpack(lf, plot)
lpack(lf, llegend(pch, labels, draw=F),
      height=unit(1, "null"), side="right")
```

4 Summary

Lattice currently has most of the support required for producing very complex arrangements of graphical components, including Trellis-like layouts; Deepayan Sarkar, a student at the University of Wisconsin is using Lattice to write a package which produces Trellis-like plots (Deepayan's package was used to produce Figure 1). In addition, Lattice graphics functions provide basic support for simple interaction by returning graphical objects and allowing those objects to be edited.

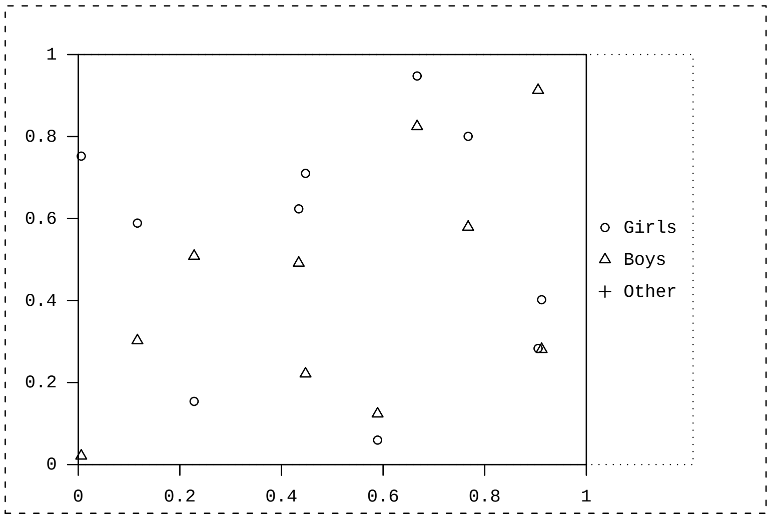

Figure 6: A plot and legend produced by packing graphical objects into a graphical frame.

There are still some major features missing, including clipping to viewports and multiple devices. Much of the code also requires tidying and the addition of argument type-checking. Nevertheless, I plan to make a beta version available on the "Devel" section of CRAN (`http://lib.stat.cmu.edu/R/CRAN/`) in the near future.

References

[1] Paul R. Murrell. Layouts: A mechanism for arranging plots on a page. *Journal of Computational and Graphical Statistics*, 1999.