data management routines are not needed, since R has its own sophisticated data subsetting tools.

Virtual topology is a standard feature of MPI. Topologies provide a high-level method for managing CPU groups without dealing with them directly. The *Cartesian* topology implementation will be the target of future version of **Rmpi**.

MPI profiling is an interesting area that may enhance MPI programming in R. It remains to be seen if the MPE (Multi-Processing Environment) library can be implemented in **Rmpi** or whether it will best be implemented as a separate package.

Other exotic advanced features of MPI under consideration are Parallel I/O and Remote Memory Access (RMA) for one-sided communication.

With parallel programming, debugging remains a challenging research area. Deadlock (i.e., a situation arising when a failed task leaves a thread waiting) and race conditions (i.e., bugs caused by failing to account for dependence on the relative timing of events) are always issues regardless of whether one is working at a low level (C(++) or Fortran) or at a high level (R). The standard MPI references can provide help.

Acknowledgements

Rmpi is primarily implemented on the SASWulf Beowulf cluster funded by NSERC equipment grants. Support from NSERC is gratefully acknowledged.

Thanks to my colleagues Drs. John Braun and Duncan Murdoch for proofreading this article. Their effects have made this article more readable.

Bibliography

A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Snuderam. *PVM: Parallel Virtual Machine. A user's guide and tutorial for networked parallel computing*. The MIT Press, Massachusetts, 1994.

W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI-The Complete Reference: Volume 2, MPI-2 Extensions*. The MIT Press, Massachusetts, 1998.

W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, Massachusetts, 1999a.

W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. The MIT Press, Massachusetts, 1999b.

Michael Na Li and A.J. Rossini. RPVM: Cluster statistical computing in R. *R News*, 1(3):4–7, September 2001. URL http://CRAN.R-project.org/doc/Rnews/.

M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI-The Complete Reference: Volume 1, The MPI Core*. The MIT Press, Massachusetts, 1998.

Hao Yu
University of Western Ontario
hyu@stats.uwo.ca

The grid Graphics Package

by Paul Murrell

Introduction

The **grid** package provides an alternative set of graphics functions within R. This article focuses on **grid** as a drawing tool. For this purpose, **grid** provides two main services:

1. the production of low-level to medium-level graphical components such as lines, rectangles, data symbols, and axes.

2. sophisticated support for arranging graphical components.

The features of **grid** are demonstrated via several examples including code. Not all of the details of the code are explained in the text so a close consideration of the output and the code that produced it, plus reference to the on-line help for specific **grid** functions, may be required to gain a complete understanding.

The functions in **grid** do not provide complete high-level graphical components such as scatterplots or barplots. Instead, **grid** is designed to make it very easy to build such things from their basic components. This has three main aims:

1. The removal of some of the inconvenient constraints imposed by R's default graphical functions (e.g., the fact that you cannot draw anything other than text relative to the coordinate system of the margins of a plot).

2. The development of functions to produce high-level graphical components which would not be very easy to produce using R's default graphical functions (e.g., the **lattice** add-on package for R, which is described in a companion article in this issue).

3. The rapid development of novel graphical displays.

Drawing primitives

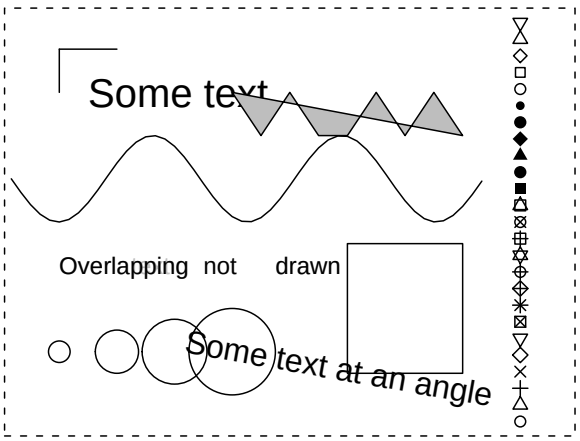

Figure 1: Example output from the **grid** primitive functions.

grid provides the standard set of basic graphical components: lines, text, rectangles, circles, polygons, and data symbols. A slight difference from the base graphics is that a set of text may be drawn such that any overlapping text is omitted. Also, **grid** has the notion of a "current location" and provides a command for resetting this location and a command for drawing a line from the previous location. The following set of code is from a sample **grid** session; the output produced by this code is shown in Figure 1.

```
grid.move.to(0.1, 0.8)
grid.line.to(0.1, 0.9)
grid.line.to(0.2, 0.9)
grid.text("Some text", x=0.15, y=0.8,
          just=c("left", "center"),
          gp=gpar(fontsize=20))
grid.text("Some text at an angle", x=0.85, y=0.1,
          just=c("right", "center"), rot=350,
          gp=gpar(fontsize=16))
grid.text(c("Overlapping", "text", "", "drawn"),
          0.1 + 0:3/8, 0.4, gp=gpar(col="grey"),
          just=c("left", "center"))
grid.text(c("Overlapping", "text", "not",
            "drawn"),
          0.1 + 0:3/8, 0.4,
          just=c("left", "center"),
          check.overlap=TRUE)
grid.circle(1:4/10, 0.2, r=1:4/40)
grid.points(rep(0.9, 25), 1:25/26, pch=1:25,
            size=unit(0.1, "inches"))
grid.polygon(0.4 + 0:8/20,
             0.6 + c(2,1,2,1,1,2,1,2,1)/10,
             gp=gpar(fill="grey"))
grid.rect(x=0.7, y=0.3, w=0.2, h=0.3)
grid.lines(x=1:50/60,
           y=0.6 + 0.1*sin(seq(-3*pi, 3*pi,
                               length=60)))
```

The functions are very similar to the base R counterparts, however, one important difference is in the way that graphical parameters, such as line colour and line thickness, are specified. There are default graphical parameter settings and any setting may be overridden by specifying a value via the `gp` argument, using the `gpar` function. There is a much smaller set of graphical parameters available:

`lty` line type (e.g., `"solid"` or `"dashed"`).

`lwd` line width.

`col` "foreground" colour for drawing borders.

`fill` "background" colour for filling shapes.

`font` a number indicating plain, bold, italic, or bold-italic.

`fontsize` the point size of the font.

`lineheight` the height of a line of text given as a multiple of the point size.

There may be additions to this list in future versions of **grid**, but it will remain much smaller than the list available in R's `par` command. Parameters such as `pch` are provided separately only where they are needed (e.g., in `grid.points()`).

Coordinate systems

All of the drawing in Figure 1 occurred within a so-called "normalised" coordinate system where the bottom-left corner of the device is represented by the location $(0, 0)$ and the top-right corner is represented by the location $(1, 1)$. For producing even simple statistical graphics, a surprising number of coordinate systems are required. **grid** provides a simple mechanism for specifying coordinate systems within rectangular regions, based on the notion of a *viewport*.

grid maintains a "stack" of viewports, which allows control over the context within which drawing occurs. There is always a default top-level viewport on the stack which provides the normalised coordinate system described above. The following commands specify a new coordinate system in which the y-axis scale is from -10 to 10 and the x-axis scale is from 0 to 5 (the call to `grid.newpage()` clears the device and resets the viewport stack):

```
grid.newpage()
push.viewport(viewport(yscale=c(-10, 10),
                       xscale=c(0, 5)))
```

All drawing operations occur within the context of the viewport at the top of the viewport stack (the *current viewport*). For example, the following command draws symbols relative to the new x- and y-scales:

```
grid.points(1:4, c(-3, 0, 3, 9))
```

In addition to x- and y-scales, a **grid** viewport can have a location and size, which position the viewport within the context of the previous viewport on the stack. For example, the following commands draw the points in a viewport that occupies only the central half of the device (the important bits are the specifications `width=0.5` and `height=0.5`):

```
grid.newpage()
push.viewport(
  viewport(width=0.5, height=0.5,
           yscale=c(-10, 10), xscale=c(0, 5)))
grid.points(1:4, c(-3, 0, 3, 9))
grid.rect(gp=gpar(lty="dashed"))
```

The margins around a plot in R are often specified in terms of lines of text. **grid** viewports provide a number of coordinate systems in addition to the normalised one and that defined by the x- and y-scales. When specifying the location and size of a graphical primitive or viewport, it is possible to select which coordinate system to use by specifying the location and/or size using a *unit* object. The following commands draw the points in a viewport with a margin given in lines of text. An x-axis and a y-axis, some labels, and a border are added to make something looking like a standard R plot (the output is shown in Figure 2). The important bits in the following code involve the use of the `unit()` function:

```
grid.newpage()
plot.vp <-
  viewport(x=unit(4, "lines"),
           y=unit(4, "lines"),
           width=unit(1, "npc") -
                 unit(4 + 2, "lines"),
           height=unit(1, "npc") -
                  unit(4 + 3, "lines"),
           just=c("left", "bottom"),
           yscale=c(-10.5, 10.5),
           xscale=c(-0.5, 5.5))
push.viewport(plot.vp)
grid.points(1:4, c(-3, 0, 3, 9))
grid.xaxis()
grid.text("X Variable",
          y=unit(-3, "lines"))
grid.yaxis()
grid.text("Y Variable",
          x=unit(-3, "lines"), rot=90)
grid.rect()
grid.text("Plot Title",
          y=unit(1, "npc") + unit(2, "lines"),
          gp=gpar(fontsize=14))
```

In terms of the arrangement of the graphical components, these few lines of code reproduce most of the layout of standard R plots. Another basic **grid** feature, *layouts*[1], allows a simple emulation of R's multiple rows and columns of plots.

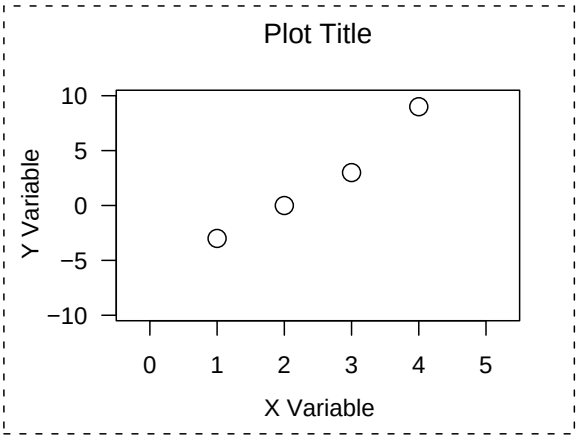

Figure 2: A standard scatterplot produced by **grid**.

If a viewport has a layout specified, then the next viewport in the stack can specify its position within that layout rather than using unit objects. For example, the following code draws the plot in the previous example within a 1-row by 2-column array (and an outer margin is added for good measure; the output is shown in Figure 3). The important bits to look for involve the specification of a layout for the first viewport, `layout=grid.layout(1, 2)`, and the specification of a position within that layout for the next viewport that is pushed onto the viewport stack, `layout.pos.col=1`:

```
grid.newpage()
push.viewport(
  viewport(x=unit(1, "lines"),
           y=unit(1, "lines"),
           width=unit(1, "npc") -
                 unit(2, "lines"),
           height=unit(1, "npc") -
                  unit(2, "lines"),
           just=c("left", "bottom"),
           layout=grid.layout(1, 2),
           gp=gpar(fontsize=6)))
grid.rect(gp=gpar(lty="dashed"))
push.viewport(viewport(layout.pos.col=1))
grid.rect(gp=gpar(lty="dashed"))
push.viewport(plot.vp)
grid.points(1:4, c(-3, 0, 3, 9))
grid.xaxis()
grid.text("X Variable", y=unit(-3, "lines"))
grid.yaxis()
grid.text("Y Variable", x=unit(-3, "lines"),
          rot=90)
grid.rect()
grid.text("Plot Title",
          y=unit(1, "npc") + unit(2, "lines"),
          gp=gpar(fontsize=8))
```

It should be noted that the `fontsize=6` setting in the first viewport overrides the default setting for all subsequent viewports in the stack and for all graphical components within the contexts that these view-

[1]These are similar to the facility provided by the base `layout` function, which mostly follows the description in Murrell, Paul R. (1999), but there is additional flexibility provided by the addition of extra units and these layouts can be nested by specifying multiple viewports in the viewport stack each with its own layout.

ports provide. This "inheritance" of graphical context from parent viewports is true for all graphical parameter settings.

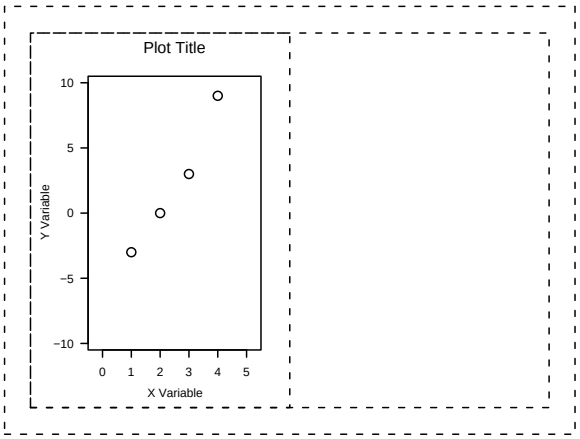

Figure 3: A **grid** scatterplot within a 1-by-2 array, within an outer margin.

There are three important features of this emulation of R's plot layout capabilities:

1. the control of the layout was performed completely within R code.
2. the amount of R code required was small.
3. the layout is actually more flexible than the R equivalent.

In order to illustrate the last point, the following examples add a couple of simple annotations to Figure 3 which are inconvenient or impossible to achieve in R.

The first three come from some requests to the R-help mailing list over over the last couple of years: drawing a plotting symbol that is a fixed number of millimeters square; drawing text within a plot in a location that is "in the top-left corner" rather than relative to the current axis scales; and drawing text at a location in the plot margins with an arbitrary rotation. The final example demonstrates the ability to draw from one coordinate system to another.

```
grid.rect(x=unit(1:4, "native"),
          width=unit(4, "mm"),
          y=unit(c(-3, 0, 3, 9), "native"),
          height=unit(4, "mm"))
grid.text("Text in\nthe upper-left\ncorner",
          x=unit(1, "mm"),
          y=unit(1, "npc") - unit(1, "mm"),
          just=c("left", "top"),
          gp=gpar(font=3))
grid.yaxis(main=FALSE, label=FALSE)
grid.text(c("-ten", "-five", "zero",
            "five", "ten"),
          x=unit(1, "npc") + unit(0.8, "lines"),
          y=unit(seq(-10, 10, 5), "native"),
          just=c("left", "centre"), rot=70)
pop.viewport(2)
push.viewport(viewport(layout.pos.col=2))
push.viewport(plot.vp)
grid.rect()
grid.points(1:4, c(-8, -3, -2, 4))

line.between <- function(x1, y1, x2, y2) {
  grid.move.to(unit(x2, "native"),
               unit(y2, "native"))
  pop.viewport(2)
  push.viewport(viewport(layout.pos.col=1))
  push.viewport(plot.vp)
  grid.line.to(unit(x1, "native"),
               unit(y1, "native"),
               gp=gpar(col="grey"))
  pop.viewport(2)
  push.viewport(viewport(layout.pos.col=2))
  push.viewport(plot.vp)
}
line.between(1, -3, 1, -8)
line.between(2, 0, 2, -3)
line.between(3, 3, 3, -2)
line.between(4, 9, 4, 4)
```

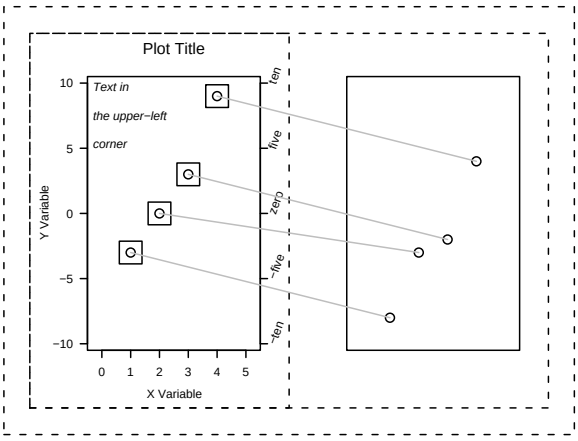

Figure 4: A **grid** scatterplot with simple annotations.

Applications of grid

R's base graphics are based on the notion of a plot which is surrounded by margins for axes and labels. Many statistical graphics cannot be conveniently described using such a plot as the basic building block. A good example, is the Trellis Graphics system (Becker et al., 1996; Cleveland, 1993), particularly the multipanel Trellis displays. Here, the more natural building block is a "panel", which consists of a plot plus one or more "strips" above it. The construction of such a panel is straightforward using grid, as the following code demonstrates (the output is shown in Figure 5:

```
grid.newpage()
lyt <-
  grid.layout(3, 1,
    heights=unit(c(1.5, 1.5, 1),
                 c("lines", "lines", "null")))
push.viewport(viewport(width=0.7,
```

```
                      height=0.7,
                      layout=lyt,
                      xscale=c(0.5, 8.5)))
push.viewport(viewport(layout.pos.row=1))
grid.rect(gp=gpar(fill="light green"))
grid.text("Strip 1")
pop.viewport()
push.viewport(viewport(layout.pos.row=2))
grid.rect(gp=gpar(fill="orange"))
grid.text("Strip 2")
pop.viewport()
push.viewport(viewport(layout.pos.row=3,
                      xscale=c(0.5, 8.5),
                      yscale=c(.1, .9)))
grid.rect()
grid.grill()
grid.points(unit(runif(5, 1, 8), "native"),
            unit(runif(5, .2, .8), "native"),
            gp=gpar(col="blue"))
grid.yaxis()
grid.yaxis(main=FALSE, label=FALSE)
pop.viewport()
grid.xaxis()
grid.xaxis(main=FALSE, label=FALSE)
```

This panel can now be included in a higher-level layout to produce an array of panels just as a scatterplot was placed within an array previously.

An important point about this example is that, once the command

```
push.viewport(viewport(layout.pos.row=1))
```

has been issued, drawing can occur within the context of the top strip, with absolutely no regard for any other coordinate systems in the graphic. For example, lines and rectangles can be drawn relative to an x-scale within this strip to indicate the value of a third conditioning variable and a text label can be drawn relative to the normalised coordinates within the strip — e.g., at location `x=unit(0, "npc")` with `just=c("left", "centre")` to left-align a label.

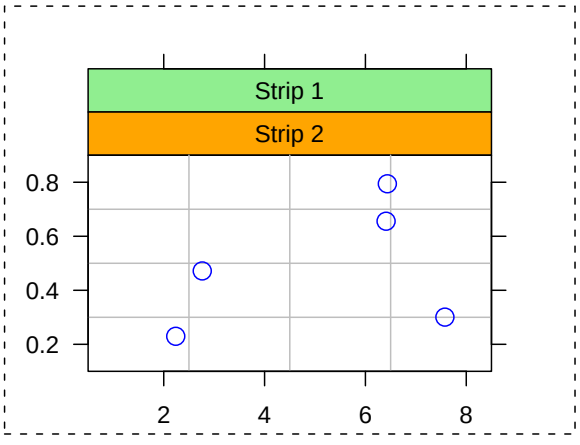

Figure 5: A Trellis-like panel produced using **grid**.

Given this sort of flexibility and power for combining graphical components, it becomes possible to seriously consider generating novel statistical graphics and producing custom images for the needs of individual data sets.

The following example, uses data from a study which was conducted to investigate the speed of cars at a location in Auckland, New Zealand. The variable of interest was the proportion of cars travelling above 60kph. This variable was recorded every hour for several days. In addition, the total volume of cars per hour was recorded. The researchers wanted to produce a graphic which presented the proportion data in a top panel and the volume data in a bottom panel as well as indicate day/night transitions using white versus grey bands and weekends or public holidays using white versus black strips.

The following code produces the desired graph, which is shown in Figure 6.

```
n <- dim(cardata)[1]
xrange <- c(0, n+1)
grid.newpage()
push.viewport(
  viewport(x=unit(3, "lines"),
           width=unit(1, "npc") -
                 unit(4, "lines"),
           y=unit(3, "lines"),
           height=unit(1, "npc") -
                  unit(5, "lines"),
           just=c("left", "bottom"),
           layout=grid.layout(5, 1,
                    heights=unit(rep(3, 1),
                      rep(c("mm", "null"),
                          length=5))),
           xscale=xrange, gp=gpar(fontsize=8)))
grid.rect(x=unit((1:n)[cardata$day == "night"],
                 "native"),
          width=unit(1, "native"),
          gp=gpar(col=NULL, fill="light grey"))
grid.rect()
grid.xaxis(at=seq(1, n, 24), label=FALSE)
grid.text(cardata$weekday[seq(1, n, 24)],
          x=unit(seq(1, n, 24)+12.5, "native"),
          y=unit(-1, "lines"))
draw.workday <- function(row) {
    push.viewport(viewport(layout.pos.row=row,
                           xscale=xrange))
    grid.rect(gp=gpar(fill="white"))
    x <- (1:n)[cardata$workday == "yes"]
    grid.rect(x=unit(x, "native"),
              width=unit(1, "native"),
              gp=gpar(fill="black"))
    pop.viewport()
}
draw.workday(1)
push.viewport(viewport(layout.pos.row=2,
                       xscale=xrange,
                       yscale=c(0.5, 1)))
grid.lines(unit(1:n, "native"),
           unit(cardata$prop, "native"))
grid.yaxis()
pop.viewport()
draw.workday(3)
push.viewport(
```

```
  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