`grid` Graphics

Paul Murrell

March 12, 2003

`grid` is *not* a set of high-level plotting functions. It is a set of low-level graphics functions and is intended to provide facilities for producing complete plots with only a moderate amount of code and a trivial amount of mental effort.

Other packages (such as `lattice`) can be written using `grid` to provide high-level plotting functions for more convenient production of complete plots. Ideally these should still allow a great deal of annotation and customisation via raw `grid` functions.

The structure of this document deliberately mirrors the structure of the document "Traditional `S` Graphics" in order to help the transition from standard `R` graphics to `grid`.

There are three things to know in order to construct a statistical graphic (plot):

1. How to create and control different graphics regions and coordinate systems.
2. How to control which graphics region and coordinate system graphics output goes into.
3. How to produce graphics output (lines, points, text, ...) including how to control its appearance (colour, line type, line width, ...).

1 Creating and Controlling Graphics Regions and Coordinate Systems

In `grid` there can be any number of graphics regions. A graphics region is referred to as a *viewport* and is created using the `viewport()` function. A viewport can be positioned anywhere on a graphics device (page, window, ...), it can be rotated, and it can be clipped to. The following code describes a viewport which is centred within the page, and is half the width of the page, one quarter of the height of the page, and rotated 45°; Figure 1 shows a diagram of this viewport.

```
> viewport(x = 0.5, y = 0.5, width = 0.5, height = 0.25, angle = 45)
```

The object returned by the `viewport()` function is only a *description* of a graphics region. A graphics region is only created on a graphics device when a viewport is "pushed" onto that device. This is achieved using the `push.viewport()` function. Each device has only one "current viewport" (by default this is the entire device), but it maintains a "stack" of viewports that have been pushed. The top viewport on this stack is the current viewport. The `push.viewport()` function places a viewport on the top of the stack, and the `pop.viewport()` function removes a viewport from the top of the stack. This means that there is always only one graphics region to draw into, but it is possible to return to a previous graphics region through the appropriate set of pushing and popping viewports. The following code creates a graphics region in the top-left corner of the page, does some drawing, then creates another region in the bottom-right corner of the page, does some drawing there, *then* returns to the first graphics region and does some more drawing (see Figure 2).

```
> grid.rect(gp = gpar(lty = "dashed"))
> vp1 <- viewport(x = 0, y = 0.5, w = 0.5, h = 0.5, just = c("left",
+     "bottom"))
> vp2 <- viewport(x = 0.5, y = 0, w = 0.5, h = 0.5, just = c("left",
+     "bottom"))
> push.viewport(vp1)
> grid.rect(gp = gpar(col = "grey"))
> grid.text("Some drawing in graphics region 1", y = 0.8)
> pop.viewport()
> push.viewport(vp2)
> grid.rect(gp = gpar(col = "grey"))
> grid.text("Some drawing in graphics region 2", y = 0.8)
> pop.viewport()
> push.viewport(vp1)
> grid.text("MORE drawing in graphics region 1", y = 0.2)
> pop.viewport()
```

When several viewports are pushed onto the viewport stack, later viewports are located and sized within the context of earlier viewports. The following code gives an example; a viewport is defined which is one-quarter the size of its parent (half the width and half the height), and this viewport is pushed twice. The first time it gets pushed it is quarter of the size of the page, the second time it is quarter of the size of *the previous viewport on the stack*. Figure 3 shows the output of these commands.

```
> grid.rect(gp = gpar(lty = "dashed"))
> vp <- viewport(width = 0.5, height = 0.5)
> push.viewport(vp)
> grid.rect(gp = gpar(col = "grey"))
> grid.text("quarter of the page", y = 0.85)
> push.viewport(vp)
> grid.rect()
> grid.text("quarter of the\nprevious viewport")
> pop.viewport(2)
```

Each viewport has a number of coordinate systems available. The full set is described in Table 1, but there are four main types: absolute coordinates (e.g., “inches”, “cm”) allow locations and sizes in terms of physical coordinates – there is no dependence on the size of the page; normalised coordinates (e.g., “npc”) allow locations and sizes as a proportion of the page size (or the current viewport); relative coordinates (i.e., “native”) allow locations and sizes relative to a user-defined set of x- and y-ranges; referential coordinates (e.g., “strwidth”) where locations and sizes are based on the size of some other graphical object.

It is possible to specify the coordinate system for relative coordinates, but all other coordinate systems are implicitly defined based on the location and size of the viewport and/or the size of other graphical objects.

2 Directing Graphics Output into Different Graphics Regions and Coordinate Systems

Graphics output is always relative to the current viewport (on the current device). Selecting which region you want is a matter of pushing and popping the appropriate viewports.

The selection of which coordinate system to use within the current viewport is made using the `unit()` function. The `unit()` function creates an object which is a combination of a value and a coordinate system (plus some extra information for certain coordinate systems). Here are some examples from the `help(unit)` page:

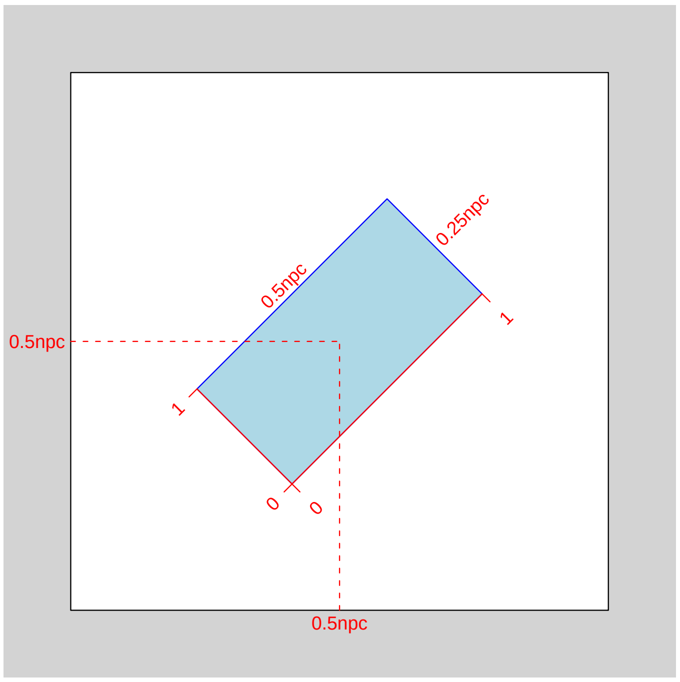

Figure 1: A diagram of a simple **grid** viewport (produced using the `grid.show.viewport()` function.

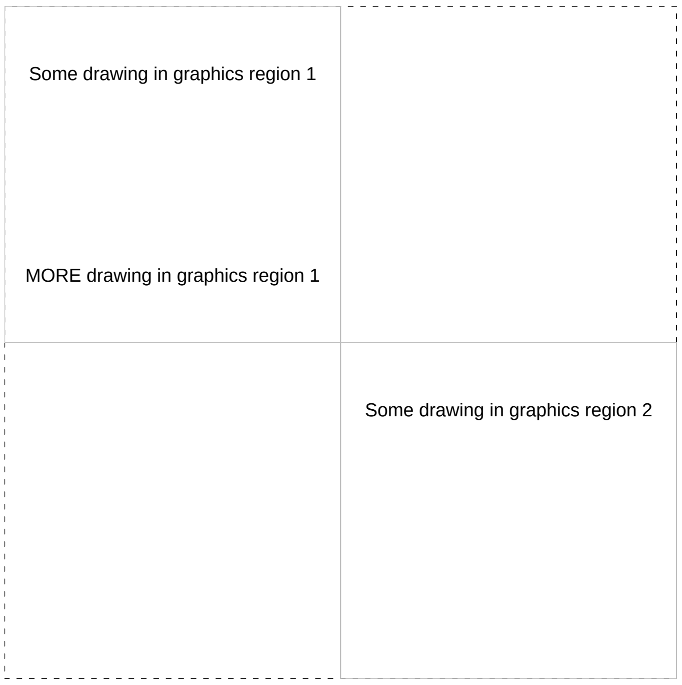
Some drawing in graphics region 1
MORE drawing in graphics region 1
Some drawing in graphics region 2

Figure 2: Defining and drawing in multiple graphics regions.

Coordinate System Name	Description
`"npc"`	Normalised Parent Coordinates. Treats the bottom-left corner of the current viewport as the location $(0, 0)$ and the top-right corner as $(1, 1)$.
`"native"`	Locations and sizes are relative to the x- and y-scales for the current viewport.
`"inches"`	Locations and sizes are in terms of physical inches. For locations, $(0, 0)$ is at the bottom-left of the viewport.
`"cm"`	Same as `"inches"`, except in centimetres.
`"mm"`	Millimetres.
`"points"`	Points. There are 72.27 points per inch.
`"bigpts"`	Big points. There are 72 big points per inch.
`"picas"`	Picas. There are 12 points per pica.
`"dida"`	Dida. 1157 dida equals 1238 points.
`"cicero"`	Cicero. There are 12 dida per cicero.
`"scaledpts"`	Scaled points. There are 65536 scaled points per point.
`"char"`	Locations and sizes are specified in terms of multiples of the current nominal `fontheight`.
`"lines"`	Locations and sizes are specified in terms of multiples of the height of a line of text (dependent on both the current `fontsize` and the current `lineheight`).
`"snpc"`	Square Normalised Parent Coordinates. Locations and size are expressed as a proportion of the *smaller* of the width and height of the current viewport.
`"strwidth"`	Locations and sizes are expressed as multiples of the width of a given string (dependent on the string and the current `fontsize`).
`"strheight"`	Locations and sizes are expressed as multiples of the height of a given string (dependent on the string and the current `fontsize`).
`"grobwidth"`	Locations and sizes are expressed as multiples of the width of a given graphical object (dependent on the current state of the graphical object).
`"grobheight"`	Locations and sizes are expressed as multiples of the height of a given graphical object (dependent on the current state of the graphical object).

Table 1: The full set of coordinate systems available in `grid` viewports.

```
> unit(1, "npc")

[1] 1npc

> unit(1:3/4, "npc")

[1] 0.25npc 0.5npc  0.75npc

> unit(1:3/4, "npc")[2]

[1] 0.5npc

> unit(1:3/4, "npc") + unit(1, "inches")

[1] 0.25npc+1inches 0.5npc+1inches  0.75npc+1inches

> min(unit(0.5, "npc"), unit(1, "inches"))

[1] min(0.5npc, 1inches)

> unit.c(unit(0.5, "npc"), unit(2, "inches") + unit(1:3/4, "npc"),
+     unit(1, "strwidth", "hi there"))

[1] 0.5npc          2inches+0.25npc 2inches+0.5npc  2inches+0.75npc
[5] 1strwidth
```

Notice that unit objects are treated much like numeric vectors. You can index a unit object, it is possible to do simple arithmetic (including `min` and `max`), and there are several unit-versions of common functions (e.g., `unit.c`, `unit.rep`, and `unit.length`; `unit.pmin`, and `unit.pmax`)).

`grid` functions that have arguments specifying locations and sizes typically assume a default coordinate system is being used. Most often this default is "npc". In other words, if a raw numeric value, `x`, is specified this is implicitly taken to mean `unit(x, "npc")`. The `viewport()` function is one that assumes "npc" coordinates, so in all of the viewport examples to this point, we have only used "npc" coordinates to position viewports within the page or within each other. It is also possible to position viewports using any of the coordinate systems described in Table 1. The following code first pushes a viewport with a user-defined x-scale, then pushes another viewport which is centred at the x-value 60 and half-way up the first viewport, and is 3 inches high[1] and as wide as the text "coordinates for everyone". Figure 4 shows the resulting output.

```
> push.viewport(viewport(y = unit(3, "lines"), width = 0.9, height = 0.8,
+     just = "bottom", xscale = c(0, 100)))
> grid.rect(gp = gpar(col = "grey"))
> grid.xaxis()
> push.viewport(viewport(x = unit(60, "native"), y = unit(0.5,
+     "npc"), width = unit(1, "strwidth", "coordinates for everyone"),
+     height = unit(3, "inches")))
> grid.rect()
> grid.text("coordinates for everyone")
> pop.viewport(2)
```

[1]If you want to check the figure, the scaling factor is 3.5/6 (i.e., the rectangle in the figure should be 1.75" or 3.94cm high).

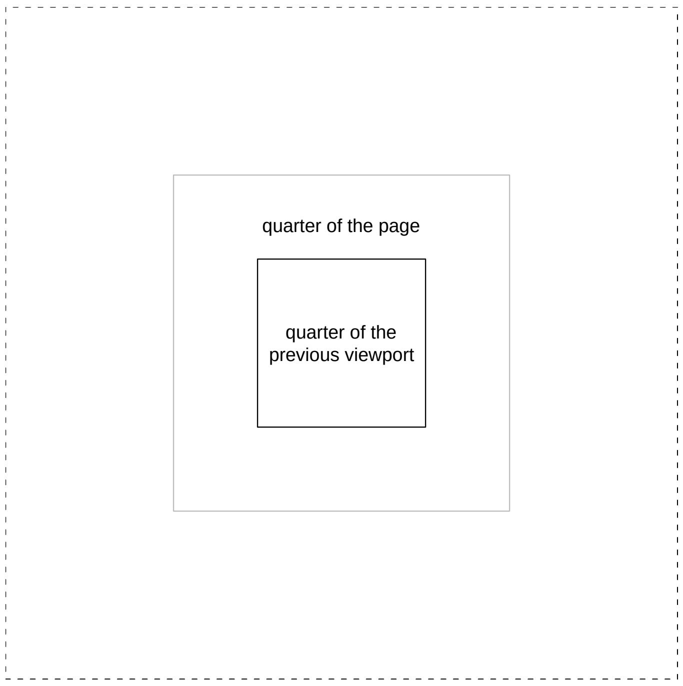

Figure 3: The result of pushing the same viewport onto the viewport stack twice.

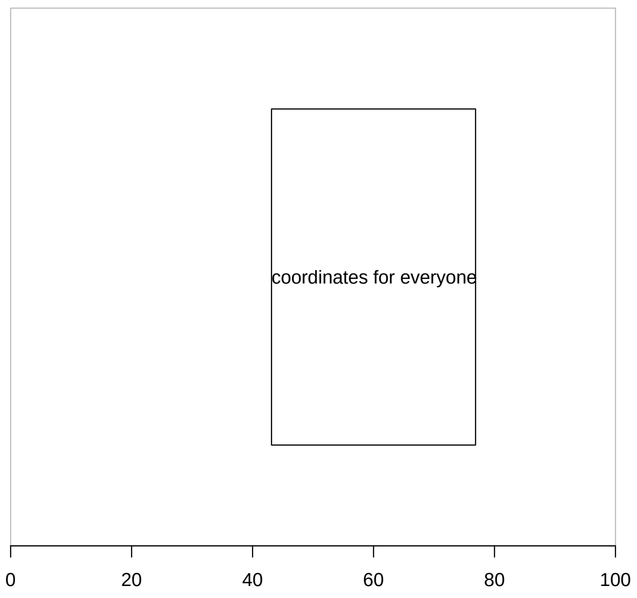

Figure 4: A viewport positioned using a variety of coordinate systems.

2.1 Layouts

`grid` provides an alternative method for positioning viewports within each other based on *layouts*[2]. A layout may be specified for any viewport. Any viewport pushed immediately after a viewport containing a layout may specify its location with respect to that layout. In the following simple example, a viewport is pushed with a layout with 4 rows and 5 columns, then another viewport is pushed which occupies the second and third columns of the third row of the layout.

```
> push.viewport(viewport(layout = grid.layout(4, 5)))
> grid.rect(gp = gpar(col = "grey"))
> grid.segments(c(1:4/5, rep(0, 3)), c(rep(0, 4), 1:3/4), c(1:4/5,
+     rep(1, 3)), c(rep(1, 4), 1:3/4), gp = gpar(col = "grey"))
> push.viewport(viewport(layout.pos.col = 2:3, layout.pos.row = 3))
> grid.rect(gp = gpar(lwd = 3))
> pop.viewport(2)
```

Layouts introduce a special sort of unit called "null". These can be used in layouts to specify relative column-widths or row-heights. In the following, slightly more complex, example, the layout specifies something similar to a standard plot arrangement; there are bottom and left margins 3 lines of text wide, top and right margins 1cm wide, and two rows and columns within these margins where the bottom row is twice the height of the top row (see Figure 6).

```
> grid.show.layout(grid.layout(4, 4, widths = unit(c(3, 1, 1, 1),
+     c("lines", "null", "null", "cm")), heights = c(1, 1, 2, 3),
+     c("cm", "null", "null", "lines")))
```

3 Producing Graphics Output

`grid` provides a standard set of graphical primitives: lines, text, points, rectangles, polygons, and circles. There are also two higher level components: x- and y-axes. Table 2 lists the `grid` functions that produce these primitives.

NOTE: all of these graphical primitives are available in all graphical regions and coordinate systems.

`grid.text`	Can specify angle of rotation.
`grid.rect`	
`grid.circle`	
`grid.polygon`	
`grid.points`	Can specify type of plotting symbol.
`grid.lines`	
`grid.segments`	
`grid.grill`	Convenience function for drawing grid lines
`grid.move.to`	
`grid.line.to`	
`grid.xaxis`	Top or bottom axis
`grid.yaxis`	Left or right axis

Table 2: `grid` graphical primitives.

[2]The primary reference for layouts is [1]. Layouts in `grid` represent an extension of the idea to allow a greater range of units for specifying row heights and column widths. `grid` also differs in the way that children of the layout specify their location within the layout.

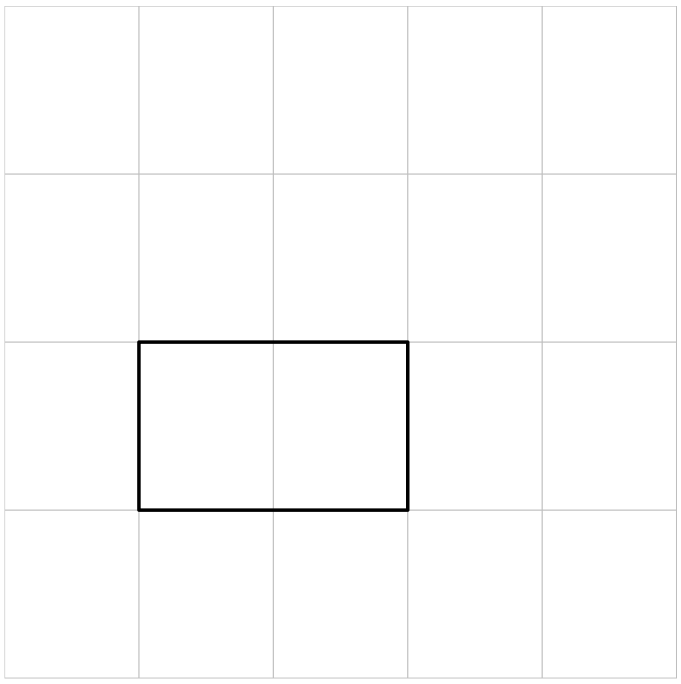

Figure 5: A viewport positioned using a layout.

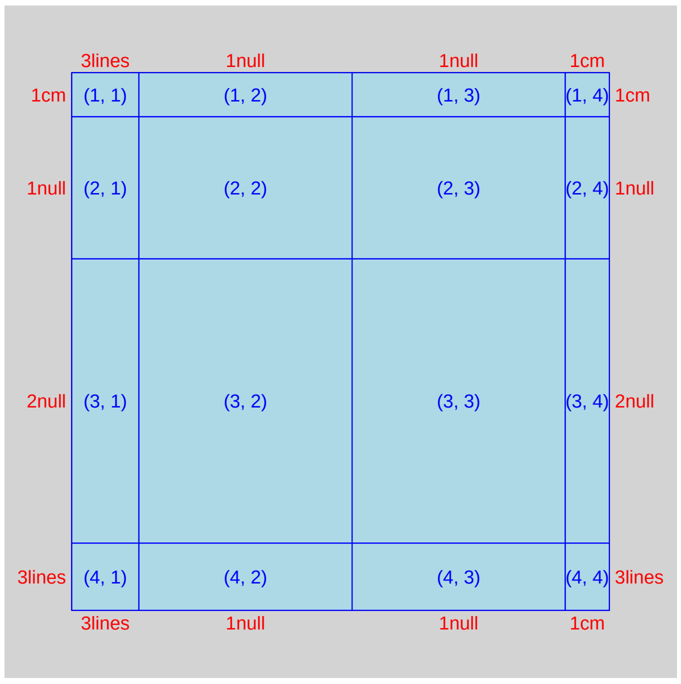

Figure 6: A more complex layout.

3.1 Controlling the Appearance of Graphics Output

grid recognises a fixed set of graphical parameters for modifying the appearance of graphical output (see Table 3).

col	colour of lines, text, ...
fill	colour for filling rectangles, circles, polygons, ...
lwd	line width
lty	line type
fontsize	The size of text (in points)
fontface	The font face (bold, italic, ...)
fontfamily	The font family

Table 3: grid graphical parameters.

Graphical parameter settings may be specified for both viewports and graphical objects. A graphical parameter setting for a viewport will hold for all graphical output within that viewport *and* for all viewports subsequently pushed onto the viewport stack, *unless* the graphical object or viewport specifies a different parameter setting.

A description of graphical parameter settings is created using the gpar() function, and this description is associated with a viewport or a graphical object via the gp argument. The following code demonstrates the effect of graphical parameter settings (see Figure 7).

```
> push.viewport(viewport(gp = gpar(fill = "grey", fontface = "italic")))
> grid.rect()
> grid.rect(width = 0.8, height = 0.6, gp = gpar(fill = "white"))
> grid.text("This text and the inner rectangle\nhave specified their own gpar settings",
+     y = 0.75, gp = gpar(fontface = "plain"))
> grid.text("This text and the outer rectangle\naccept the gpar settings of the viewport",
+     y = 0.25)
> pop.viewport()
```

4 Plots from First Principles

This is the only way to produce a plot using grid itself. However, just like in standard graphics, it is quite straightforward to create a simple plot by hand. The following code produces the equivalent of a standard plot(1:10) (see Figure 8).

```
> grid.rect(gp = gpar(lty = "dashed"))
> x <- y <- 1:10
> push.viewport(plotViewport(c(5.1, 4.1, 4.1, 2.1)))
> push.viewport(dataViewport(x, y))
> grid.rect()
> grid.xaxis()
> grid.yaxis()
> grid.points(x, y)
> grid.text("1:10", x = unit(-3, "lines"), rot = 90)
> pop.viewport(2)
```

Now consider a more complex example, where we create a barplot with a legend (see Figure 9). There are two main parts to this because grid has no predefined barplot function; the construction of the barplot will itself be instructive, so we will start with just that.

The data to be plotted are as follows: we have four measures to represent at four levels; the data are in a matrix with the measures for each level in a column.

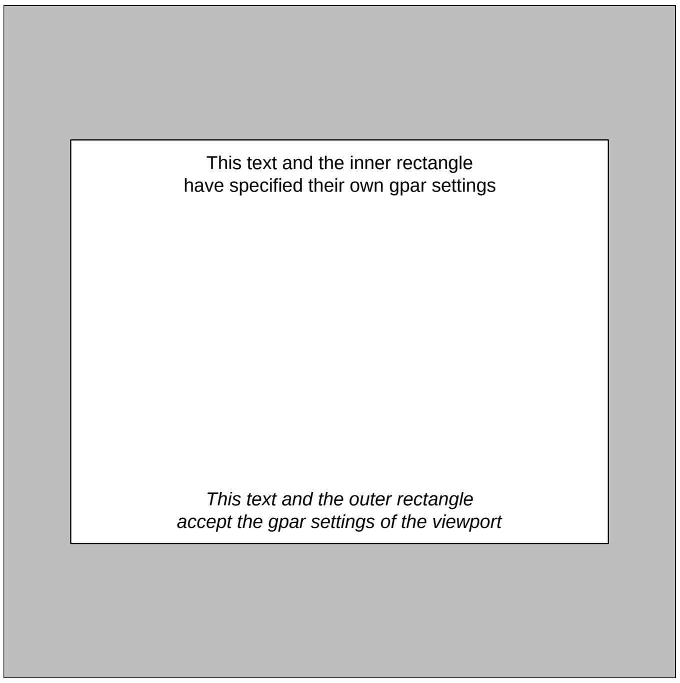
This text and the inner rectangle
have specified their own gpar settings
This text and the outer rectangle
accept the gpar settings of the viewport

Figure 7: The effect of different graphical parameter settings.

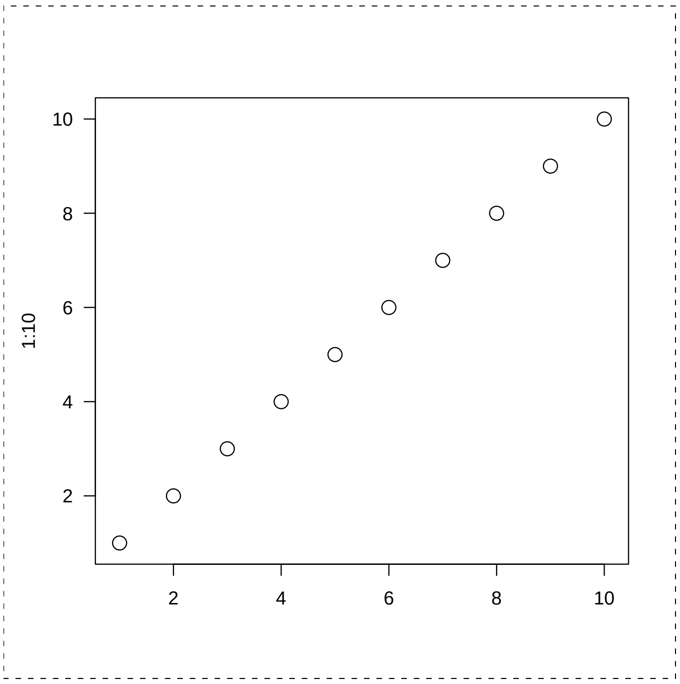

Figure 8: The grid equivalent of `plot(1:10)`.

```
> barData <- matrix(sample(1:4, 16, replace = T), ncol = 4)
```

We will use colours to differentiate the measures.

```
> boxColours <- 1:4
```

We create the barplot within a function so that we can easily reproduce it when we combine it with the legend.

```
> bp <- function(barData) {
+     nbars <- dim(barData)[2]
+     nmeasures <- dim(barData)[1]
+     barTotals <- rbind(rep(0, nbars), apply(barData, 2, cumsum))
+     barYscale <- c(0, max(barTotals) * 1.05)
+     push.viewport(plotViewport(c(5, 4, 4, 1), yscale = barYscale,
+         layout = grid.layout(1, nbars)))
+     grid.rect()
+     grid.yaxis()
+     for (i in 1:nbars) {
+         push.viewport(viewport(layout.pos.col = i, yscale = barYscale))
+         grid.rect(x = rep(0.5, nmeasures), y = unit(barTotals[1:nmeasures,
+             i], "native"), height = unit(diff(barTotals[, i]),
+             "native"), width = 0.8, just = "bottom", gp = gpar(fill = boxColours))
+         pop.viewport()
+     }
+     pop.viewport()
+ }
```

Now we turn our attention to the legend. We need some labels and we will enforce the constraint that the boxes in the legend should be 0.5" square:

```
> legLabels <- c("Group A", "Group B", "Group C", "Something Longer")
> boxSize <- unit(0.5, "inches")
```

The following draws the legend elements in a column, with each element consisting of a box with a label beneath.

```
> leg <- function(legLabels) {
+     nlabels <- length(legLabels)
+     push.viewport(viewport(layout = grid.layout(nlabels, 1)))
+     for (i in 1:nlabels) {
+         push.viewport(viewport(layout.pos.row = i))
+         grid.rect(width = boxSize, height = boxSize, just = "bottom",
+             gp = gpar(fill = boxColours[i]))
+         grid.text(legLabels[i], y = unit(0.5, "npc") - unit(1,
+             "lines"))
+         pop.viewport()
+     }
+     pop.viewport()
+ }
```

Now that we have the two components, we can arrange them together to form a complete image. Notice that we can perform some calculations to make sure that we leave enough room for the legend, including 1 line of text as left and right margins. We also impose top and bottom margins on the legend to match the plot margins.

```
> grid.rect(gp = gpar(lty = "dashed"))
> legend.width <- max(unit(rep(1, length(legLabels)), "strwidth",
+     as.list(legLabels)) + unit(2, "lines"), unit(0.5, "inches") +
+     unit(2, "lines"))
> push.viewport(viewport(layout = grid.layout(1, 2, widths = unit.c(unit(1,
+     "null"), legend.width))))
> push.viewport(viewport(layout.pos.col = 1))
> bp(barData)
> pop.viewport()
> push.viewport(viewport(layout.pos.col = 2))
> push.viewport(plotViewport(c(5, 0, 4, 0)))
> leg(legLabels)
> pop.viewport(3)
```

5 grid and lattice

The lattice package is built on top of grid and provides a quite sophisticated example of writing high-level plotting functions using grid. Because lattice consists of grid calls, it is possible to both add grid output to lattice output, and lattice output to grid output.

5.1 Adding grid to lattice

Panel functions in lattice can include grid calls. The following example adds a horizontal line at 0 to a standard xyplot (see Figure 10):

```
> x <- rnorm(100)
> y <- rnorm(100)
> g <- sample(1:8, 100, replace = T)
> print.trellis(xyplot(y ~ x | g, panel = function(x, y) {
+     panel.xyplot(x, y)
+     grid.lines(unit(c(0, 1), "npc"), unit(0, "native"), gp = gpar(col = "grey"))
+ }))
```

The following example writes a left-justified label in each strip (see Figure 11):

```
> x <- rnorm(100)
> y <- rnorm(100)
> g <- sample(1:8, 100, replace = T)
> print.trellis(xyplot(y ~ x | g, strip = function(which.given,
+     which.panel, var.name, factor.levels, shingle.intervals,
+     strip.names = "", style, bg, fg, par.strip.text) {
+     grid.text(paste("Variable ", which.given, ": Level ", which.panel[which.given],
+         sep = ""), unit(1, "mm"), 0.5, just = "left")
+ }))
```

5.2 Adding lattice to grid

It is also possible to use a lattice plot as an element of a grid image. The following example splits up the page so that there is an xyplot beside a panel of text (see Figure 12). First of all, the lattice plot is created, but not drawn. grid is used to create some regions and the lattice plot is drawn into one of those regions.

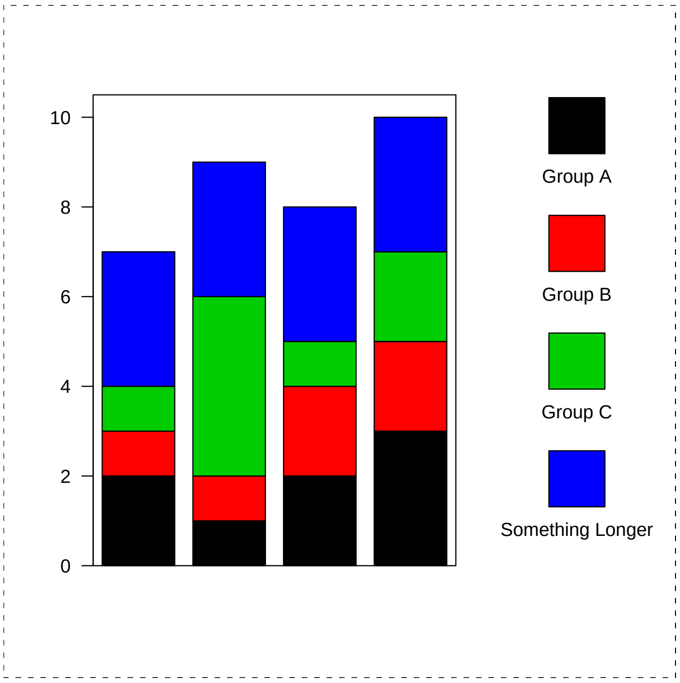

Figure 9: A barplot plus legend from first principles using `grid`.

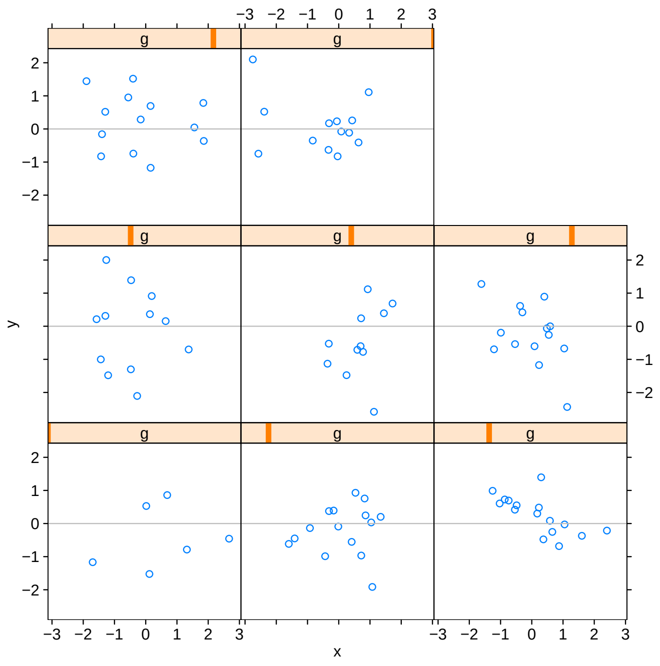

Figure 10: A `lattice` panel function using `grid`.

```
> x <- rnorm(100)
> y <- rnorm(100)
> g <- sample(1:8, 100, replace = T)
> someText <- "A panel of text\nproduced using\nraw grid code\nthat describes\nthe plot\nto the r
> latticePlot <- xyplot(y ~ x | g, layout = c(2, 4))
> grid.rect(gp = gpar(lty = "dashed"))
> push.viewport(viewport(layout = grid.layout(1, 2, widths = unit.c(unit(1,
+     "strwidth", someText) + unit(2, "cm"), unit(1, "null")))))
> push.viewport(viewport(layout.pos.col = 1))
> grid.rect(gp = gpar(fill = "light grey"))
> grid.text(someText, x = unit(1, "cm"), y = unit(1, "npc") - unit(1,
+     "inches"), just = c("left", "top"))
> pop.viewport()
> push.viewport(viewport(layout.pos.col = 2))
> print.trellis(latticePlot, newpage = FALSE)
> pop.viewport(2)
```

References

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

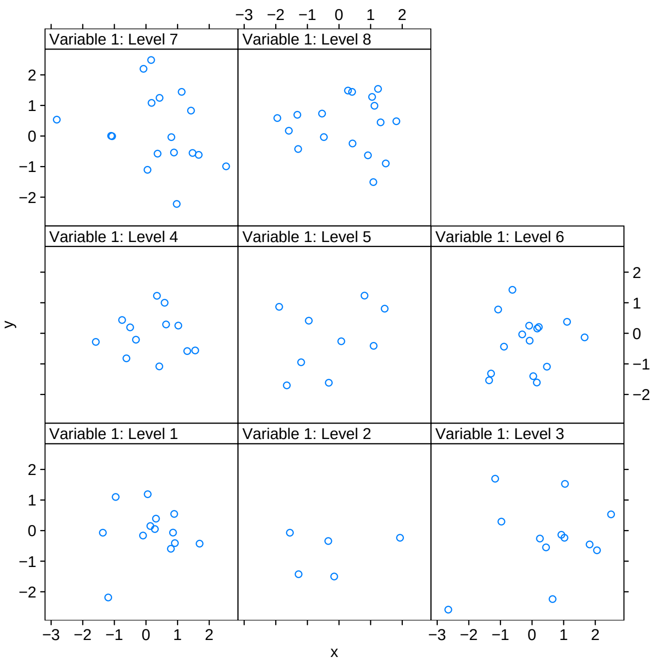

Figure 11: A `lattice` strip function using `grid`.

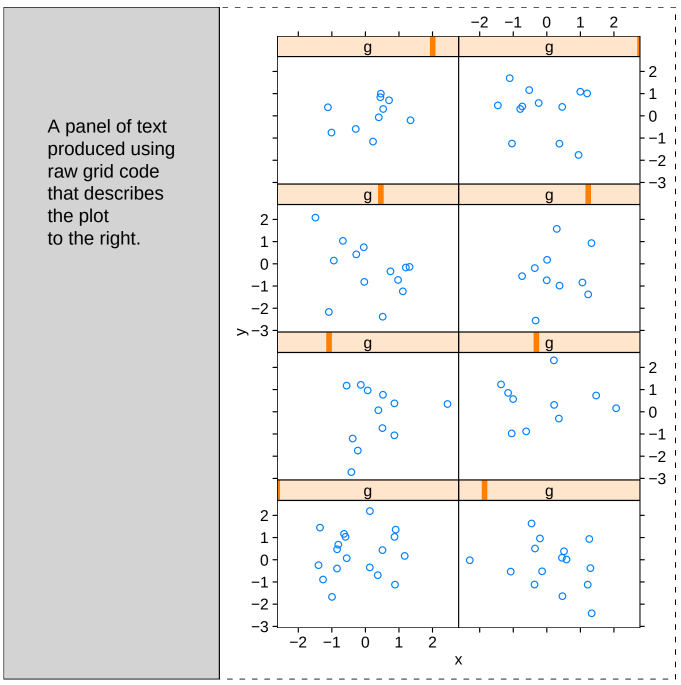

Figure 12: A `lattice` plot used as a component of a larger `grid` image.