

Integrating Grid Graphics Code with Base Graphics Code

Paul Murrell

July 23, 2003

The `grid` graphics package is much more powerful than base graphics when it comes to combining and arranging graphical elements. The major weakness of `grid` is that there is very little implemented yet in terms of high-level plots¹. Furthermore, the task of reimplementing the very many high-level plots already implemented in base graphics (i.e., which users depend on having) is an extremely daunting one. It would be nice to be able to combine the ready-made base plots with the sophisticated arrangement features of `grid`.

This document describes the `gridBase` package which provides some support for combining `grid` and base graphics output.

The `gridBase` Functions

There are two `gridBase` functions:

`baseViewports` can be used to create `grid` viewports that correspond to the current base plot; you can then annotate a base plot using `grid` functions and, importantly, position them using `grid` units and possibly further `grid` viewports.

This function returns a list of three `grid` viewports. The first corresponds to the base “inner” region. It is relative to the entire device; it only makes sense to push this viewport from the “top level” (i.e., only when no other viewports have been pushed). The second viewport corresponds to the base “figure” region and is relative to the inner region; it only makes sense to push it after the first viewport has been pushed. The third viewport corresponds to the base “plot” region and is relative to the figure region; it only makes sense to push it after the other two have been pushed in the correct order.

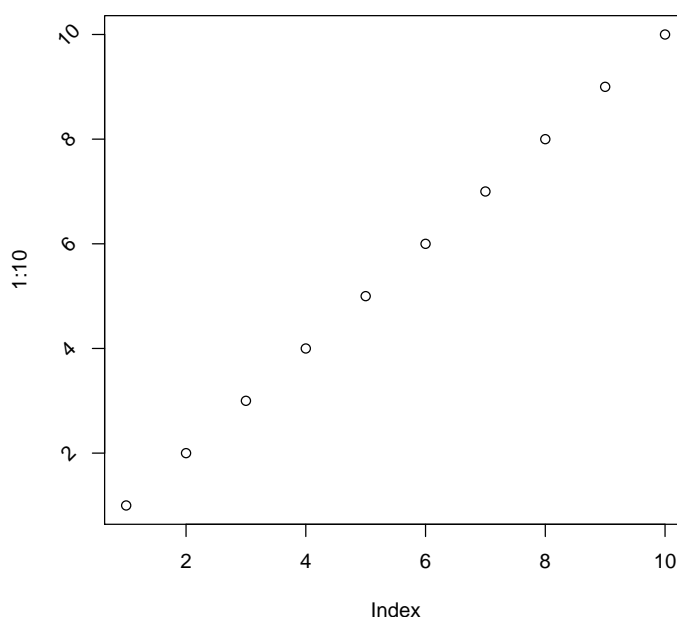
A simple application of this facility involves adding text to the margins of a base plot at an arbitrary orientation. The base function `mtext` allows text to be located in terms of a number of lines away from the plot region,

¹No disrespect intended to Deepayan’s `lattice`, Frank Harrell’s `Hmisc`, M. Kondrin’s `Disgrace`, and possibly others I don’t know about!

but only at rotations of 0 or 90 degrees. The base `text` function allows arbitrary rotations, but only locates text relative to the user coordinate system in effect in the plot region (which is inconvenient for locating text in the margins of the plot). By contrast, the `grid` function `grid.text` allows arbitrary rotations and can be used in any `grid` viewport. The following code creates a base plot, leaving off the tick labels, then uses `baseViewports` to create grid viewports that correspond to the base plot, pushes those viewports, then draws rotated labels using `grid.text` ²:

```
> plot(1:10, axes = FALSE)
> axis(1)
> axis(2, at = seq(2, 10, 2), labels = FALSE)
> box()
> vps <- baseViewports()
> par(new = TRUE)
> push.viewport(vps$inner, vps$figure, vps$plot)
> grid.text(seq(2, 10, 2), x = unit(-1.5, "lines"), y = unit(seq(2,
+ 10, 2), "native"), rot = 45)
> pop.viewport(3)
```

NULL

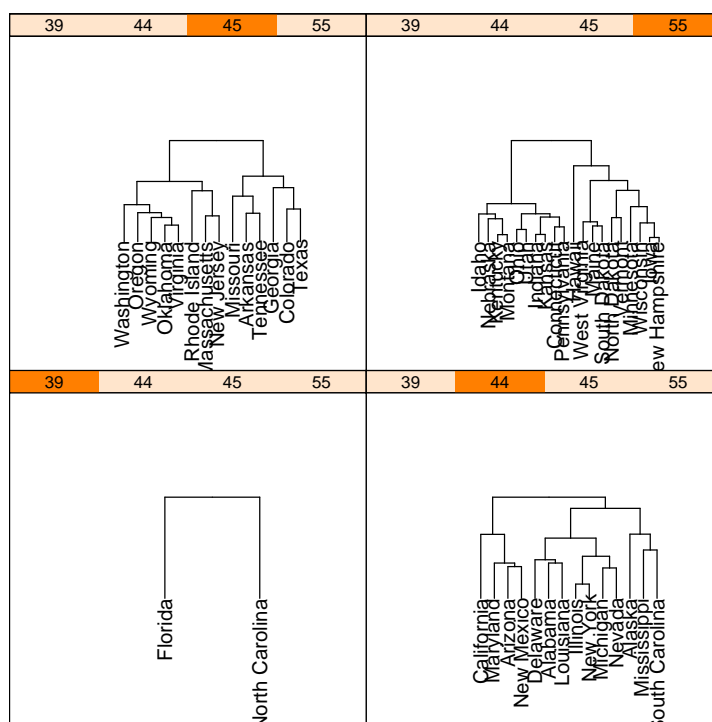


`gridOMI` can be used to set the base `par(omi)` parameter so that base plots will draw within the current grid viewport. In other words, this sets the base “inner” region to correspond to the current grid viewport.

²The `par(new=TRUE)` is necessary currently because the first grid action will try to move to a new page; it should be possible to remove that step in future versions of R.

The main usefulness of this function is to allow you to create a complex layout using `grid` and then draw a base plot within relevant elements of that complex layout. The following example uses this idea to create a lattice plot where the panels contain dendrograms.

```
> library(mva)
> data(USArrests)
> hc <- hclust(dist(USArrests), "ave")
> dend1 <- as.dendrogram(hc)
> dend2 <- cut(dend1, h = 70)
> library(lattice)
> x <- 1:4
> y <- 1:4
> height <- factor(round(unlist(lapply(dend2$lower, attr, "height"))))
> dendpanel <- function(x, y, subscripts, ...) {
+   gridOMI()
+   par(new = TRUE)
+   plot(dend2$lower[[subscripts]], axes = FALSE)
+ }
> xyplot(y ~ x | height, subscripts = TRUE, xlab = "", ylab = "",
+   strip = function(...) {
+     strip.default(style = 4, ...)
+   }, scales = list(draw = FALSE), panel = dendpanel)
```



gridBase by Example

We will now look at a reasonably complete example to show how **gridBase** helps us to solve the following problem, described in an email to R-help (18 July 2003) from Adam Langley:

“I am looking at a way of plotting a series of pie charts at specified locations on an existing plot. The size of the pie chart would be proportion to the magnitude of the total value of each vector (x) and the values in x are displayed as the areas of pie slices.”

First of all, let's construct some random data, consisting of four (x, y) values, and four (z_1, z_2) values :

```
> x <- runif(4)
> y <- runif(4)
> z <- matrix(runif(4 * 2), ncol = 2)
```

Before we start any plotting, we'll save the current **par** settings so that at the end we can “undo” some of the complicated settings that we need to apply.

```
> oldpar <- par(no.readonly = TRUE)
```

Now we do a standard base plot of the (x, y) values, but do not plot anything at these locations (we're just setting up the user coordinate system).

```
> plot(x, y, xlim = c(-0.2, 1.2), ylim = c(-0.2, 1.2), type = "n")
```

Now we make use of **baseViewports**. This will create a list of grid viewports that correspond to the inner, figure, and plot regions set up by the base plot. By pushing these viewports, we establish a grid viewport that aligns exactly with the plot region created by the base plot, including a (grid) “native” coordinate system that matches the (base) user coordinate system³.

```
> vps <- baseViewports()
> par(new = TRUE)
> push.viewport(vps$inner, vps$figure, vps$plot)
> grid.grill(h = y, v = x, default.units = "native")
```

Before we draw the pie charts, we need to perform a couple of calculations to determine their size. In this case, we specify that the largest pie will be 1" in diameter and the others will be a proportion of that size based on $\sum_i z_i / \max(\sum_i z_i)$

³The **grid.grill** call is just drawing some grey lines to show that the pie charts we end up with are centred correctly at the appropriate (x, y) locations.

```

> maxpiesize <- unit(1, "inches")
> totals <- apply(z, 1, sum)
> sizemult <- totals/max(totals)

```

We now enter a loop to draw a pie at each (x, y) location representing the corresponding (z_1, z_2) values. The first step is to create a grid viewport at the (x, y) location, then we use `gridOMI` to set the base inner region to correspond to the grid viewport. With that done, we can use the base `pie` function to draw a pie chart within the grid viewport⁴.

```

> for (i in 1:4) {
+   push.viewport(viewport(x = unit(x[i], "native"), y = unit(y[i],
+     "native"), width = sizemult[i] * maxpiesize, height = sizemult[i] *
+     maxpiesize))
+   grid.rect(gp = gpar(col = "grey", fill = "white", lty = "dashed"))
+   gridOMI()
+   par(mar = rep(0, 4), new = TRUE)
+   pie(z[i, ], radius = 1, labels = rep("", 2))
+   pop.viewport()
+ }

```

Finally, we clean up after ourselves by popping the grid viewports and restoring the initial `par` settings.

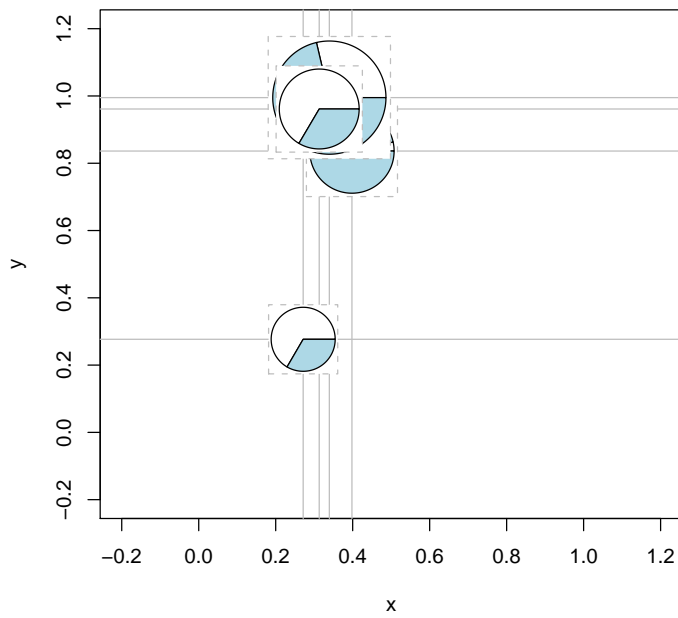
```

> pop.viewport(3)
> par(oldpar)

```

The final plot looks like this:

⁴We draw a `grid.rect` with a dashed border just to show the extent of each grid viewport. We also use `par(mar)` to set the plot margins to zero; we are not drawing labels on the pie charts and we want the pie chart to completely fill the grid viewport. It is crucial that we again call `par(new=TRUE)` so that we do not move on to a new page.



Problems and Limitations