Integrating Grid Graphics Output with Base Graphics Output

Paul Murrell

March 19, 2012

The `grid` graphics package[1] is much more powerful than the standard `R` graphics system (hereafter "base graphics") when it comes to combining and arranging graphical elements. It is possible to create a greater variety of graphs more easily with `grid` (see, for example, Deepayan Sarkar's `lattice` package[2]). However, there are very many plots based on base graphics (e.g., biplots), that have not been implemented in `grid`, and the task of reimplementing these in `grid` is extremely daunting. 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.

Annotating base graphics using `grid`

The `gridBase` package has one function, `baseViewports()`, that supports adding `grid` output to a base graphics plot. This function creates a set of `grid` viewports that correspond to the current base plot. This allows simple annotations such as adding lines and text using `grid`'s units to locate them relative to a wide variety of coordinate systems, or something more complex involving pushing further `grid` viewports.

`baseViewports()` returns a list of three grid viewports. The first corresponds to the base "inner" region. This viewport 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 "inner" 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 viewports 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, 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. In the following code we first create a base plot, leaving off the tick labels.

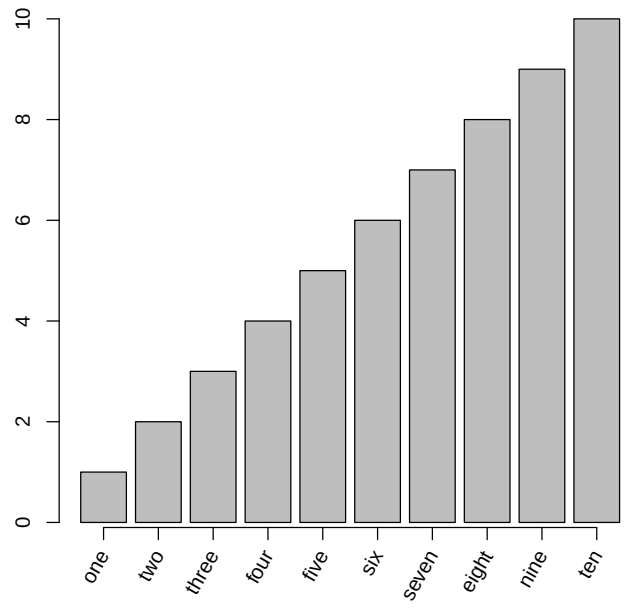
```
> midpts <- barplot(1:10, axes=FALSE)
> axis(2)
> axis(1, at=midpts, labels=FALSE)
>
```

Next we use `baseViewports()` to create grid viewports that correspond to the base plot and we push those viewports.

```
> vps <- baseViewports()
> pushViewport(vps$inner, vps$figure, vps$plot)
>
```

Finally, we draw rotated labels using `grid.text()` (and pop the viewports to clean up after ourselves).

```
> grid.text(c("one", "two", "three", "four", "five",
+             "six", "seven", "eight", "nine", "ten"),
+           x=unit(midpts, "native"), y=unit(-1, "lines"),
+           just="right", rot=60)
> popViewport(3)
>
```



The next example is a bit more complicated because it involves embedding `grid` viewports within a base graphics plot. The dataset is a snapshot of wind speed, wind direction, and temperature at several weather stations in the South China Sea, south west of Japan[1]. `grid` is used to produce novel plotting symbols for a standard base plot.

First of all, we need to define the novel plotting symbol. This consists of a dot at the data location, with a thermometer extending "below" and an arrow extending "above". The thermometer is used to encode temperature and the arrow is used to indicate wind speed (both scaled to [0, 1]).

```
> novelsym <- function(speed, temp,
+                      width=unit(3, "mm"),
+                      length=unit(0.5, "inches")) {
+   grid.rect(height=length, y=0.5,
+             just="top", width=width,
+             gp=gpar(fill="white"))
+   grid.rect(height=temp*length,
+             y=unit(0.5, "npc") - length,
+             width=width,
+             just="bottom", gp=gpar(fill="grey"))
+   grid.lines(x=0.5,
+              y=unit.c(unit(0.5, "npc"), unit(0.5, "npc") + speed*length),
+              arrow=arrow(length=unit(3, "mm"), type="closed"),
```

[1]Obtained from the CODIAC web site: `http://www.joss.ucar.edu/codiac/codiac-www.html`. The file `chinasea.txt` is in the `gridBase/doc` directory.

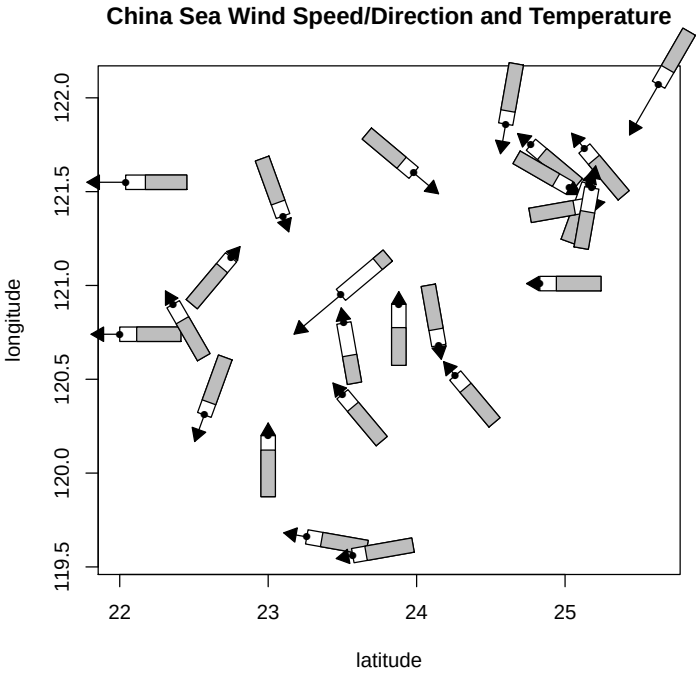
```
+              gp=gpar(fill="black"))
+   grid.points(unit(0.5, "npc"), unit(0.5, "npc"), size=unit(2, "mm"),
+               pch=16)
+ }
>
```

Now we read in the data and generate a base plot, but plot no points.

```
> chinasea <- read.table(system.file("doc", "chinasea.txt",
+                                    package="gridBase"),
+                        header=TRUE)
> plot(chinasea$lat, chinasea$long, type="n",
+   xlab="latitude", ylab="longitude",
+   main="China Sea Wind Speed/Direction and Temperature")
>
```

Now we use `baseViewports()` to align a `grid` viewport with the plot region, and draw the symbols by creating a `grid` viewport per (x, y) location (we rotate the viewport to represent the wind direction).

```
> speed <- 0.8*chinasea$speed/14 + 0.2
> temp <- chinasea$temp/40
> vps <- baseViewports()
> pushViewport(vps$inner, vps$figure, vps$plot)
> for (i in 1:25) {
+   pushViewport(viewport(x=unit(chinasea$lat[i], "native"),
+                         y=unit(chinasea$long[i], "native"),
+                         angle=chinasea$dir[i]))
+   novelsym(speed[i], temp[i])
+   popViewport()
+ }
> popViewport(3)
>
```



Embedding base graphics plots in `grid` viewports

`gridBase` provides several functions for adding base graphics output to `grid` output. There are three functions that allow base plotting regions to be aligned with the current `grid` viewport; this makes it possible to draw one or more base graphics plots within a `grid` viewport. The fourth function provides a set of graphical parameter settings so that base `par()` settings can be made to correspond to some of[2] the current `grid` graphical parameter settings.

The first three functions are `gridOMI()`, `gridFIG()`, and `gridPLT()`. They return the appropriate `par()` values for setting the base "inner", "figure", and "plot" regions, respectively.

The main usefulness of these functions is to allow you to create a complex layout using `grid` and then draw a base plot within relevant elements of that layout. The following example uses this idea to create a `lattice` plot where the panels contain dendrograms drawn using base graphics functions[3].

First of all, we create a dendrogram and cut it into four subtrees[4].

```
> data(USArrests)
```

[2]Only `lwd`, `lty`, `col` are available yet. More should be available in future versions.

[3]Recall that `lattice` is built on `grid` so the panel region in a `lattice` plot is a `grid` viewport.

[4]the data and cluster analysis are copied from the example in `help(plot.dendrogram)`.

```
>      hc <- hclust(dist(USArrests), "ave")
>      dend1 <- as.dendrogram(hc)
>      dend2 <- cut(dend1, h=70)
>
```

Now we create some dummy variables which correspond to the four subtrees.

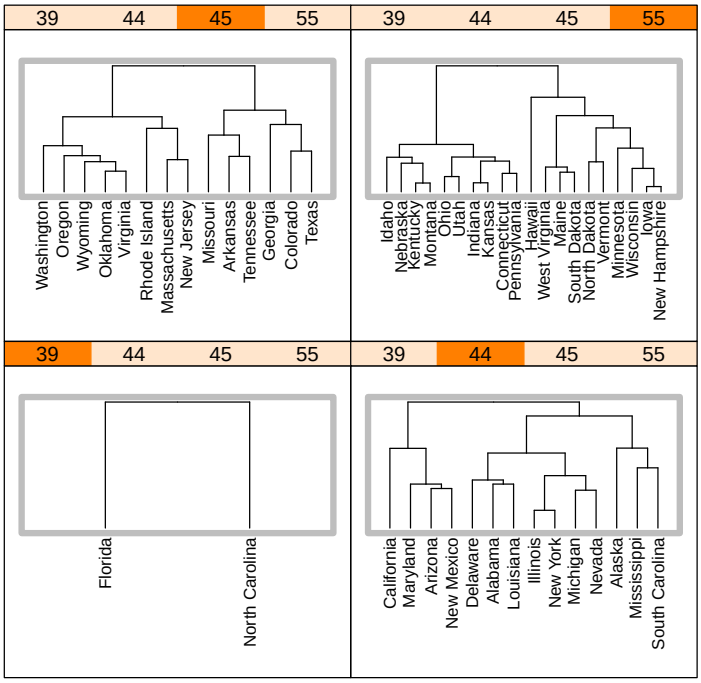
```
> x <- 1:4
> y <- 1:4
> height <- factor(round(unlist(lapply(dend2$lower, attr, "height"))))
>
```

Next we define a `lattice` panel function to draw the dendrograms. The first thing this panel function does is push a viewport that is smaller than the viewport `lattice` creates for the panel; the purpose is to ensure there is enough room for the labels on the dendrogram. The `space` variable contains a measure of the length of the longest label. The panel function then calls `gridPLT()` and makes the base plot region correspond to the viewport we have just pushed. Finally, we call the base `plot()` function to draw the dendrogram (and pop the viewport we pushed)[5].

```
> space <- max(unit(rep(1, 50), "strwidth",
+              as.list(rownames(USArrests))))
> dendpanel <- function(x, y, subscripts, ...) {
+   pushViewport(viewport(y=space, width=0.9,
+                         height=unit(0.9, "npc") - space,
+                         just="bottom"))
+   grid.rect(gp=gpar(col="grey", lwd=5))
+   par(plt=gridPLT(), new=TRUE, ps=10)
+   plot(dend2$lower[[subscripts]], axes=FALSE)
+   popViewport()
+ }
>
```

Finally, we draw a `lattice` xyplot, using `lattice` to set up the arrangement of panels and strips and our panel function to draw a base dendrogram in each panel.

[5]The `grid.rect()` call is just to show the extent of the extra viewport we pushed.



The `gridPLT()` function is useful for embedding just the plot region of a base graphics function (i.e., without labels and axes; another example of this usage is given in the next section). If labelling and axes are to be included it will make more sense to use `gridFIG()`. The `gridOMI()` function has pretty much the same effect as `gridFIG()` except that it allows for the possibility of embedding multiple base plots at once. In the following code, a `lattice` plot is placed alongside base diagnostic plots arranged in a 2-by-2 array.

We use the data from page 93 of "An Introduction to Generalized Linear Models" (Annette Dobson, 1990).

```
>       counts <- c(18,17,15,20,10,20,25,13,12)
>       outcome <- gl(3,1,9)
>       treatment <- gl(3,3)
>
```

We create two regions using `grid` viewports; the left region is for the `lattice` plot and the right region is for the diagnostic plots. There is a middle column of 1cm to provide a gap between the two regions.

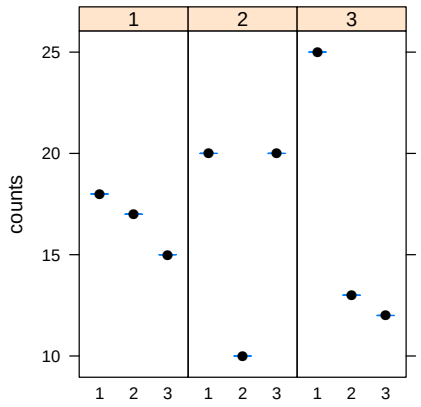
```
> pushViewport(viewport(layout=grid.layout(1, 3,
+   widths=unit(rep(1, 3), c("null", "cm", "null")))))
>
```

We draw a `lattice` plot in the left region.

```
> pushViewport(viewport(layout.pos.col=1))
> library(lattice)
> bwplot <- bwplot(counts ~ outcome | treatment)
> print(bwplot, newpage=FALSE)
> popViewport()
>
```

We draw the diagnostic plots in the right region. Here we use `gridOMI()` to set the base inner region and `par(mfrow)` and `par(mfg)` to insert multiple plots[6].

```
> pushViewport(viewport(layout.pos.col=3))
>       glm.D93 <- glm(counts ~ outcome + treatment, family=poisson())
>       par(omi=gridOMI(), mfrow=c(2, 2), new=TRUE)
>       par(cex=0.5, mar=c(5, 4, 1, 2))
>       par(mfg=c(1, 1))
>       plot(glm.D93, caption="", ask=FALSE)
> popViewport(2)
>
```



Notice that because there is only ever one current `grid` viewport, it only makes sense to use one of `gridOMI()`, `gridFIG()`, or `gridPLT()`. In other words, it only makes sense to align either the inner region, or the figure region, or the plot region with the current `grid` viewport.

A more complex example

We will now look at a reasonably complex example involving embedding base graphics within grid viewports which are themselves embedded within a base plot. This example is motivated by the following problem[7]:

[6] We use `par(mfrow)` to specify the 2-by-2 array and `par(mfg)` to start at position $(1, 1)$ in the array.

[7] This description is from an email to R-help from Adam Langley, 18 July 2003

> 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, we construct some fake data, consisting of four (x, y) values, and four (z_1, z_2) values :

```
> x <- c(0.88, 1.00, 0.67, 0.34)
> y <- c(0.87, 0.43, 0.04, 0.94)
> z <- matrix(runif(4*2), ncol=2)
>
```

Before we start any plotting, we 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 system[8].

```
> vps <- baseViewports()
> pushViewport(vps$inner, vps$figure, vps$plot)
> grid.segments(x0=unit(c(rep(0, 4), x),
+                       rep(c("npc", "native"), each=4)),
+               x1=unit(c(x, x), rep("native", 8)),
+               y0=unit(c(y, rep(0, 4)),
+                       rep(c("native", "npc"), each=4)),
+               y1=unit(c(y, y), rep("native", 8)),
+               gp=gpar(lty="dashed", col="grey"))
>
```

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 $\sum_i z_{.i} / \max\left(\sum_i z_{.i}\right)$

[8] The `grid.segments` call is just drawing some dashed 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 `gridPLT()` to set the base plot 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 viewport[9].

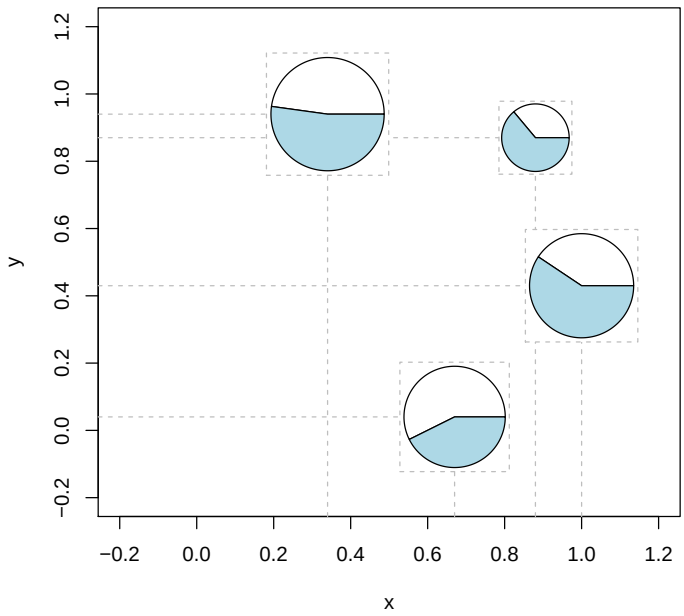
```
> for (i in 1:4) {
+   pushViewport(viewport(x=unit(x[i], "native"),
+                         y=unit(y[i], "native"),
+                         width=sizemult[i]*maxpiesize,
+                         height=sizemult[i]*maxpiesize))
+   grid.rect(gp=gpar(col="grey", fill="white", lty="dashed"))
+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

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

```
> popViewport(3)
> par(oldpar)
>
```

The final plot is shown below.

[9] We draw a `grid.rect` with a dashed border just to show the extent of each 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

The functions provided by the `gridBase` package allow the user to mix output from two quite different graphics systems and there are limits to how much the systems can be combined. It is important that users are aware that they are mixing two not wholly compatible systems (which is why these functions are provided in a separate package) and it is of course important to know what the limitations are:

- The `gridBase` functions attempt to match `grid` graphics settings with base graphics settings (and vice versa). This is only possible under certain conditions. For a start, it is only possible if the device size does not change. If these functions are used to draw into a window, then the window is resized, the base and `grid` settings will almost certainly no longer match and the graph will become a complete mess. This also applies to copying output between devices of different sizes.
- It is not possible to embed base graphics output within a `grid` viewport that is rotated.
- There are certain base graphics functions which modify settings like `par(omi)` and `par(fig)` themselves (e.g., `coplot()`). Output from these functions may not embed properly within `grid` viewports.

- `grid` output cannot be saved and restored so any attempts to save a mixture of `grid` and base output are likely to end in disappointment.

Summary

The functions in the `gridBase` package provide a simple mechanism for combining base graphics output with `grid` graphics output for static, fixed-size plots. This is not a full integration of the two graphics systems, but it does provide a useful bridge between the existing large body of base graphics functions and the powerful new features of `grid`.

Availability

The `grid` package is now part of the base distribution of `R` (from `R` version 1.8.0).
Additional information on `grid` is available from:
`http://www.stat.auckland.ac.nz/ paul/grid/grid.html`.
The `gridBase` package is available from `CRAN` (e.g., `http://cran.us.r-project.org`).

References

[1] Paul Murrell. The grid graphics package. *R News*, 2(2):14–19, June 2002.

[2] Deepayan Sarkar. Lattice. *R News*, 2(2):19–23, June 2002.