Writing **grid** Code

Paul Murrell

November 11, 2026

The **grid** system contains a degree of complexity in order to allow things like editing graphical objects, "packing" graphical objects, and so on. This means that many of the predefined Grid graphics functions are relatively complicated[1].

One design aim of **grid** is to allow users to create simple graphics simply and not to force them to use complicated concepts or write complicated code unless they actually need to. Along similar lines, it is intended that people should be able to prototype even complex graphics very simply and then refine the implementation into a more sophisticated form if necessary.

With the predefined graphics functions being fully-developed and complicated implementations, there is a lack of examples of simple, prototype code. Furthermore, given that the aim is to allow a range of ways to produce the same graphical output, there is a need for examples which demonstrate the various stages, from simple to complex, that a piece of **grid** code can go through.

This document describes the construction of a scatterplot object, like that shown below, going from the simplest, prototype implementation to the most complex and sophisticated. It demonstrates that if you only want simple graphics output then you can do it pretty simply and quickly. It also demonstrates how to write functions that allow your graphics to be used by other people. Finally, it demonstrates how to make your graphics fully interactive (or at least as interactive as Grid will let you make it).

This document should be read *after* the **grid** Users' Guide. Here we are assuming that the reader has an understanding of viewports, layouts, and units. For the later sections of the document, it will also be helpful to have an understanding of R's `S3` object system.

Procedural grid

The simplest way to produce graphical output in Grid is just like producing standard R graphical output. You simply issue a series of graphics commands and each command adds more ink to the plot. The purpose of the commands is simply to produce graphics output; in particular, we are not concerned with any values returned by the plotting functions. I will call this *procedural graphics*.

[1] Although there are exceptions; some functions, such as `grid.show.viewport`, are purely for producing illustrative diagrams and remain simple and procedural.

In order to draw a simple scatterplot, we can issue a series of commands which draw the various components of the plot.

Here are some random data to plot.

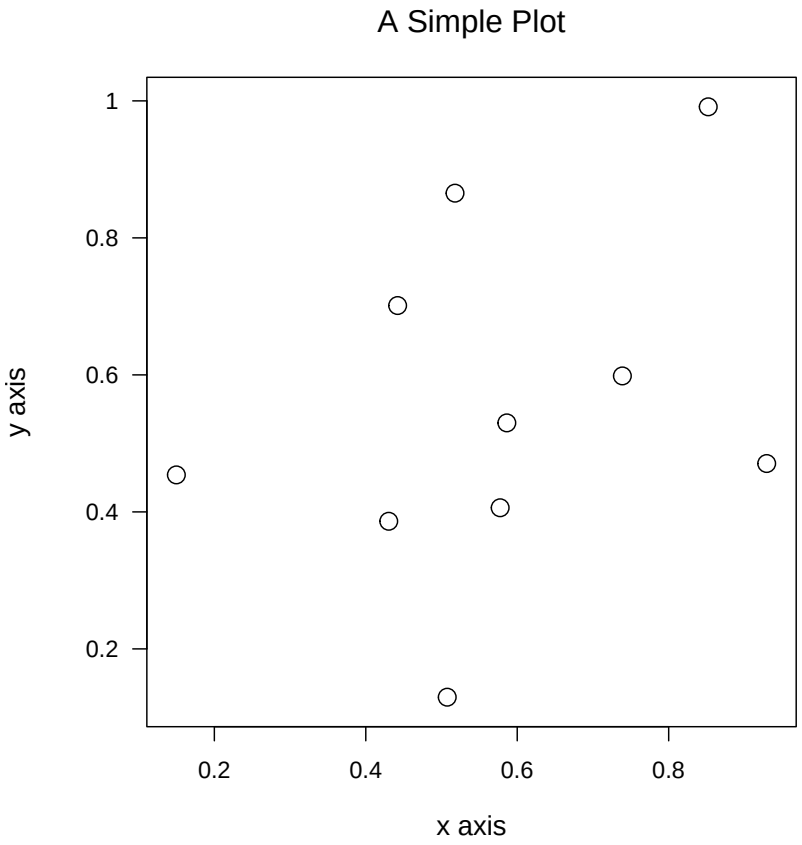
```
> x <- runif(10)
> y <- runif(10)
```

The first step in creating the plot involves defining a "data" region. This is a region which has sensible scales on the axes for plotting the data and margins around the outside for the axes to fit in, with a space for a title at the top.

```
> data.vp <- viewport(x = unit(5, "lines"),
+                     y = unit(4, "lines"),
+                     width = unit(1, "npc") - unit(7, "lines"),
+                     height = unit(1, "npc") - unit(7, "lines"),
+                     just = c("left", "bottom"),
+                     xscale = range(x) + c(-0.05, 0.05)*diff(range(x)),
+                     yscale = range(y) + c(-0.05, 0.05)*diff(range(y)))
```

Now we create the data region and draw the components of the plot relative to it: points, axes, labels, and a title.

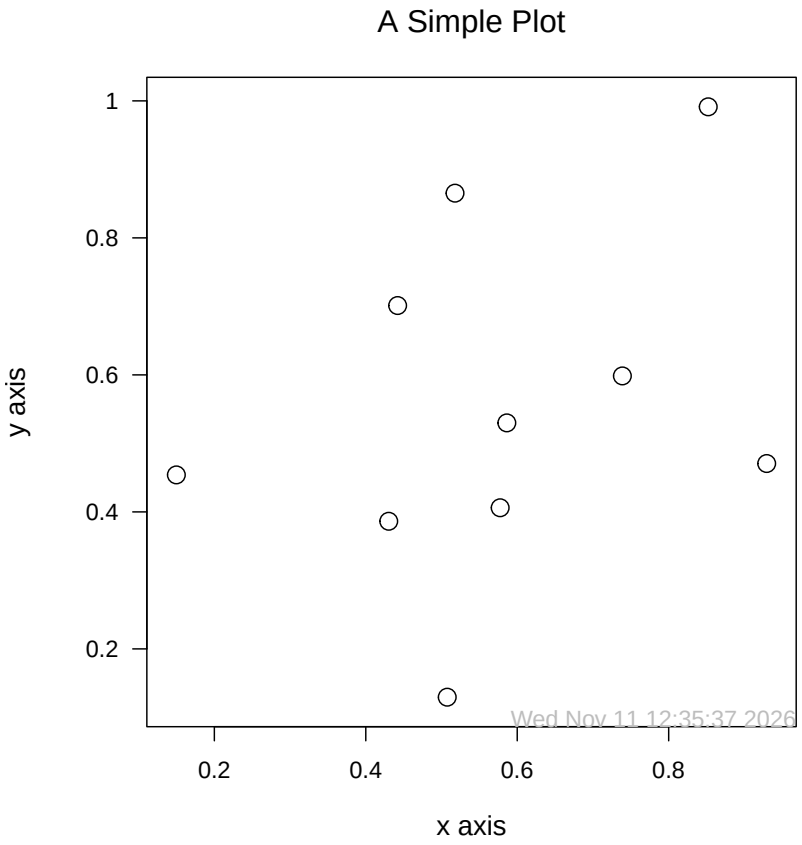
```
> pushViewport(data.vp)
> grid.points(x, y)
> grid.rect()
> grid.xaxis()
> grid.yaxis()
> grid.text("x axis", y = unit(-3, "lines"),
+           gp = gpar(fontsize = 14))
> grid.text("y axis", x = unit(-4, "lines"),
+           gp = gpar(fontsize = 14), rot = 90)
> grid.text("A Simple Plot",
+           y = unit(1, "npc") + unit(1.5, "lines"),
+           gp = gpar(fontsize = 16))
> popViewport()
```



Facilitating Annotation

Issuing a series of commands to produce a plot, like in the previous section, allows the user to have a great deal of flexibility. It is always possible to recreate viewports in order to add further annotations. For example, the following code recreates the data region in order to place the date at the bottom right corner.

```
> pushViewport(data.vp)
> grid.text(date(), x = unit(1, "npc"), y = 0,
+           just = c("right", "bottom"), gp = gpar(col="grey"))
> popViewport()
```



When more complex arrangements of viewports are involved, there may be a bewildering array of viewports created, which may make it difficult for other users to revisit a particular region of a plot. A `lattice` plot is a good example. In such cases, it will be more cooperative to use `upViewport()` rather than `popViewport()` and leave the viewports that were created during the drawing of the plot. Other users can then use `vpPath`s to navigate to the desired region. For example, here is a slight modification of the original series of commands, where the original data viewport is given a name and `upViewport()` is used at the end.

```
> data.vp <- viewport(name = "dataregion",
+                     x = unit(5, "lines"),
+                     y = unit(4, "lines"),
+                     width = unit(1, "npc") - unit(7, "lines"),
+                     height = unit(1, "npc") - unit(7, "lines"),
+                     just = c("left", "bottom"),
+                     xscale = range(x) + c(-0.05, 0.05)*diff(range(x)),
+                     yscale = range(y) + c(-0.05, 0.05)*diff(range(y)))
> pushViewport(data.vp)
> grid.points(x, y)
> grid.rect()
> grid.xaxis()
> grid.yaxis()
> grid.text("x axis", y = unit(-3, "lines"),
```

```
+           gp = gpar(fontsize = 14))
> grid.text("y axis", x = unit(-4, "lines"),
+           gp = gpar(fontsize = 14), rot = 90)
> grid.text("A Simple Plot",
+           y = unit(1, "npc") + unit(1.5, "lines"),
+           gp = gpar(fontsize = 16))
> upViewport()
```

The date is now added using `downViewport()` to get to the data region.

```
> downViewport("dataregion")
> grid.text(date(), x = unit(1, "npc"), y  =  0,
+           just = c("right", "bottom"), gp = gpar(col = "grey"))
> upViewport()
```

Writing a grid Function

Here is the scatterplot code wrapped up as a simple function.

```
> splot <- function(x = runif(10), y = runif(10), title = "A Simple Plot") {
+     data.vp <- viewport(name = "dataregion",
+                         x = unit(5, "lines"),
+                         y = unit(4, "lines"),
+                         width = unit(1, "npc") - unit(7, "lines"),
+                         height = unit(1, "npc") - unit(7, "lines"),
+                         just = c("left", "bottom"),
+                         xscale = range(x) + c(-.05, .05)*diff(range(x)),
+                         yscale = range(y) + c(-.05, .05)*diff(range(y)))
+     pushViewport(data.vp)
+     grid.points(x, y)
+     grid.rect()
+     grid.xaxis()
+     grid.yaxis()
+     grid.text("y axis", x = unit(-4, "lines"),
+               gp = gpar(fontsize = 14), rot = 90)
+     grid.text(title, y = unit(1, "npc") + unit(1.5, "lines"),
+               gp = gpar(fontsize = 16))
+     upViewport()
+ }
```

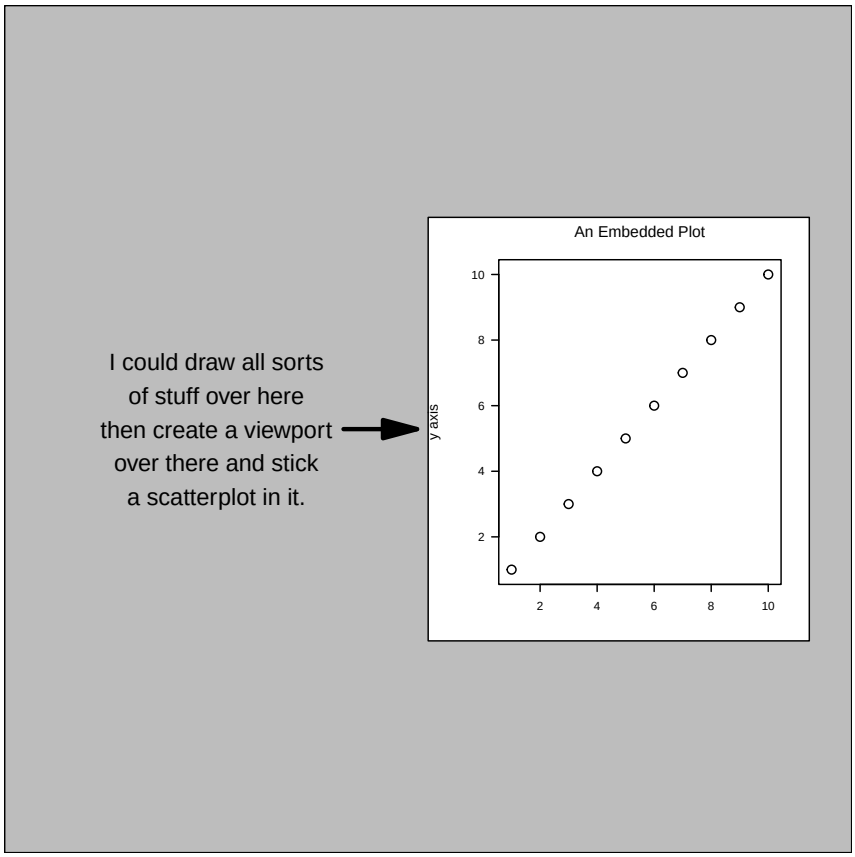
There are several advantages to creating a function:

1. We get the standard advantages of a function: we can reuse and maintain the plot code more easily.
2. We can slightly generalise the plot. In this case, we can use it for different data and have a different title. We could add more arguments to allow different margins, control over the axis scales, and so on.

3. The plot can be embedded in other graphics output.

Here is an example which uses the `splot()` function to create a slightly modified scatterplot, embedded within other **grid** output.

```
> grid.rect(gp = gpar(fill = "grey"))
> message <-
+     paste("I could draw all sorts",
+           "of stuff over here",
+           "then create a viewport",
+           "over there and stick",
+           "a scatterplot in it.", sep = "\n")
> grid.text(message, x = 0.25)
> grid.lines(x = unit.c(unit(0.25, "npc") + 0.5*stringWidth(message) +
+            unit(2, "mm"),
+            unit(0.5, "npc") - unit(2, "mm")),
+            y = 0.5,
+            arrow = arrow(angle = 15, type = "closed"),
+            gp = gpar(lwd = 3, fill = "black"))
> pushViewport(viewport(x = 0.5, height = 0.5, width = 0.45, just = "left",
+                       gp = gpar(cex = 0.5)))
> grid.rect(gp = gpar(fill = "white"))
> splot(1:10, 1:10, title = "An Embedded Plot")
> upViewport()
```



It is still straightforward to annotate the scatterplot as long as we have enough information about the viewports. In this case, a non-strict `downViewport()` will still work (though note that `upViewport(0)` is required to get right back to the top level).

Creating grid Graphical Objects

A **grid** function like the one in the previous section provides output which is very flexible and can be annotated in arbitrary ways and can be embedded within other output. This is likely to satisfy most uses.

However, there are some things that cannot be done (or at least would be extremely hard to do) with such a function. The output produced by the function cannot be addressed as a coherent whole. It is not possible, for example, to to change the `x` and `y` data used in the plot and have the points and axes update automatically. There is no scatterplot object to save; the individual components exist, but they are not bound together as a whole. If/when these sorts of issues become important, it becomes necessary to create a **grid** graphical object (a `grob`) to represent the plot.

The first step is to write a function which will create a `grob` – a *constructor* function. In most cases, this will involve creating a special sort of `grob` called a `gTree`; this is just a `grob` that can have other `grob`s as children. Here's an example for creating an `splot grob`.

I have put bits of the construction into separate functions, for reasons which will become apparent later.

```
> splot.data.vp <- function(x, y) {
+   viewport(name = "dataregion",
+            x = unit(5, "lines"),
+            y = unit(4, "lines"),
+            width = unit(1, "npc") - unit(7, "lines"),
+            height = unit(1, "npc") - unit(7, "lines"),
+            just = c("left", "bottom"),
+            xscale = range(x) + c(-.05, .05)*diff(range(x)),
+            yscale = range(y) + c(-.05, .05)*diff(range(y)))
+ }
> splot.title <- function(title) {
+       textGrob(title, name = "title",
+                y = unit(1, "npc") + unit(1.5, "lines"),
+                gp = gpar(fontsize = 16), vp = "dataregion")
+ }
> splot <- function(x, y, title, name=NULL, draw=TRUE, gp=gpar(), vp=NULL) {
+     spg <- gTree(x = x, y = y, title = title, name = name,
+                  childrenvp  =  splot.data.vp(x, y),
+                  children = gList(rectGrob(name = "border",
+                                            vp = "dataregion"),
+                  xaxisGrob(name = "xaxis", vp = "dataregion"),
+                  yaxisGrob(name = "yaxis", vp = "dataregion"),
+                  pointsGrob(x, y, name = "points", vp = "dataregion"),
+                  textGrob("x axis", y = unit(-3, "lines"), name = "xlab",
+                           gp = gpar(fontsize = 14), vp = "dataregion"),
+                  textGrob("y axis", x = unit(-4, "lines"), name = "ylab",
+                           gp = gpar(fontsize = 14), rot = 90,
+                           vp = "dataregion"),
+                  splot.title(title)),
+                  gp = gp, vp = vp,
+                  cl = "splot")
+     if (draw) grid.draw(spg)
+     spg
+ }
```

There are four important additions to the argument list compared to the original `splot()` function:

1. The `name` argument allows a string identifier to be associated with the scatterplot object we create. This is important for being able to specify the scatterplot when we try to edit it after drawing it and/or when it is part of a larger `grob` (see later examples).

2. The `draw` argument makes it possible to use the function in a procedural manner as before:

   ```
   > splot(1:10, 1:10, "Same as Before", name = "splot")
   ```

```
splot[splot]
```

```
> downViewport("dataregion")
> grid.text(date(), x = unit(1, "npc"), y = 0,
+           just = c("right", "bottom"), gp = gpar(col = "grey"))
> upViewport(0)
```

3. The `gp` argument allows the user to supply `gpar()` settings for the scatterplot as a whole.

4. The `vp` argument allows the user to supply a viewport for the `splot grob` to be drawn in. This is especially useful for specifying a `vpPath` when the `splot` is used as a component of another `grob` (see scatterplot matrix example below).

The important parts of the `gTree` definition are:

1. The `children` argument provides a list of `grob`s which are part of the scatterplot. When the scatterplot is drawn, all children will be drawn. Notice that instead of the procedural `grid.*()` functions we use `*Grob()` functions which just produce `grob`s and do not perform any drawing. Also notice that I have given each of the children a name; this will make it possible to access the components of the scatterplot (see later examples).

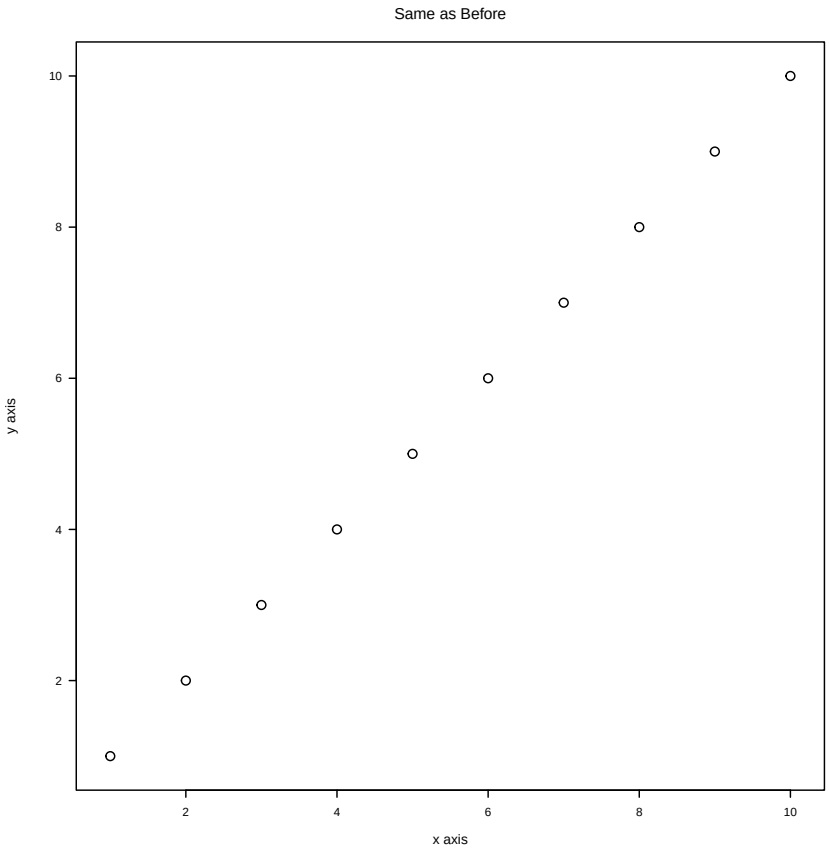
2. The `childrenvp` argument provides a viewport (or `vpStack`, `vpList`, or `vpTree`) which will be pushed before the children are drawn. The difference between this argument and the `vp` argument common to all `grob`s is that the `vp` is pushed before drawing the children and then popped after, whereas the `childrenvp` gets pushed *and* then a call to `upViewport()` is made before the children are drawn. This allows the children to simply specify the viewport they should be drawn in by way of a `vpPath` in their `vp` argument. In this way, viewports remain available for further annotation such as we have already seen in procedural code.

3. The `gp` and `vp` arguments are automatically handled by the `gTree` drawing methods so that `gpar()` settings will be enforced and the viewport will be pushed when the `splot` is drawn.

4. The `cl` argument means that the `grob` created is a special sort of `grob` called `splot`. This will allow us to write methods specifically for our scatterplot (see later examples).

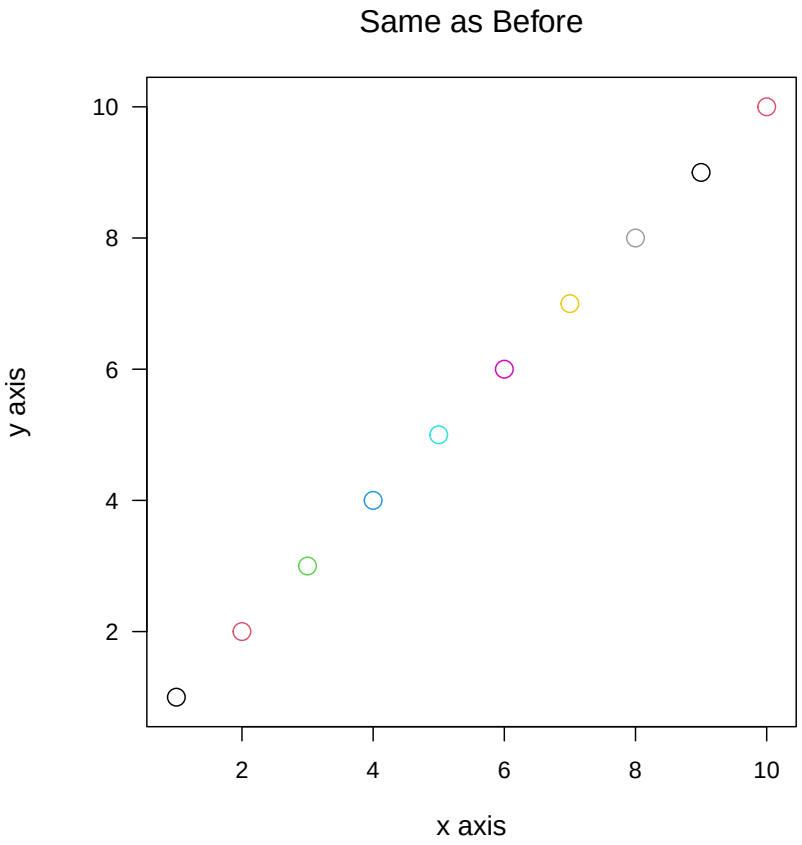
Now that we have a `grob`, there are some more interesting things that we can do with it. First of all, the `splot grob` provides a container for the `grob`s which make up the scatterplot. If we modify the `splot grob`, it affects all of the children.

```
> splot(1:10, 1:10, "Same as Before", name = "splot")
> grid.edit("splot", gp = gpar(cex=0.5))
```



We can access elements of the `splot grob` to edit them individually.

```
> splot(1:10, 1:10, "Same as Before", name = "splot")
> grid.edit(gPath("splot", "points"), gp = gpar(col = 1:10))
```

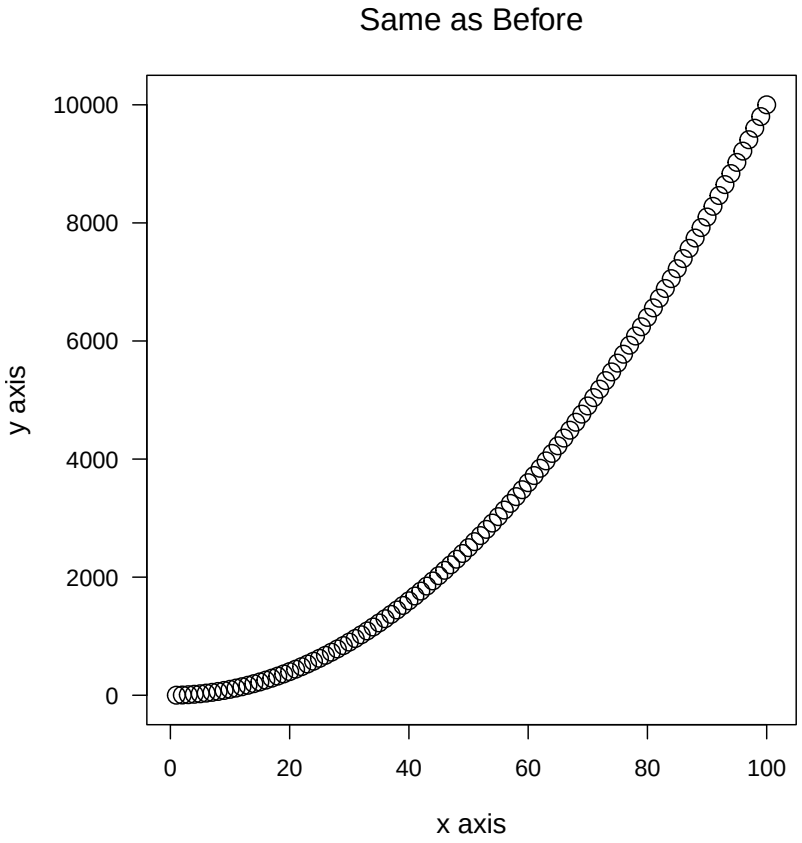


With a little more work we can make the scatterplot a bit more dynamic. The following describes a `editDetails()` method for the `splot grob`. This will be called whenever a scatterplot is edited and will update the components of the scatterplot.

```
> editDetails.splot <- function(x, specs) {
+     if (any(c("x", "y") %in% names(specs))) {
+         if (is.null(specs$x)) xx <- x$x else xx <- specs$x
+         if (is.null(specs$y)) yy <- x$y else yy <- specs$y
+         x$childrenvp <- splot.data.vp(xx, yy)
+         x <- addGrob(x, pointsGrob(xx, yy, name = "points",
+                                    vp = "dataregion"))
+     }
+   x
+ }
> splot(1:10, 1:10, "Same as Before", name = "splot")
```

```
splot[splot]
```

```
> grid.edit("splot", x = 1:100, y = (1:100)^2)
```



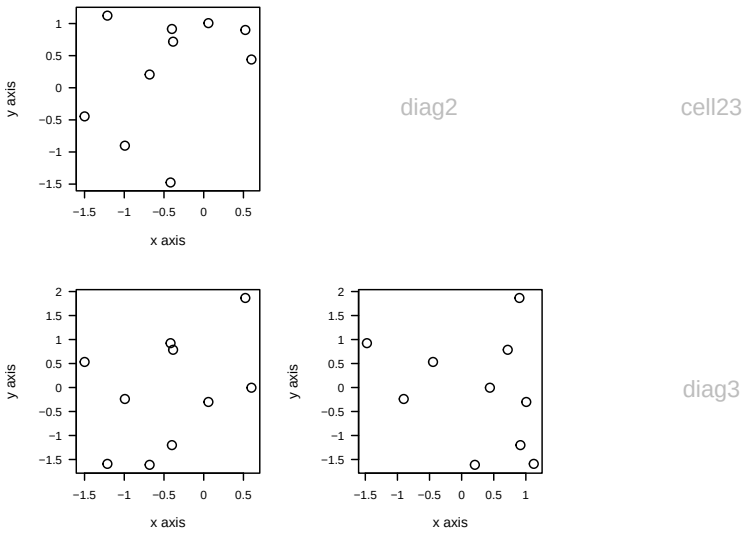
The `splot grob` can also be used in the construction of other `grob`s. Here's a simple scatterplot matrix `grob`[2].

```
> cellname <- function(i, j) paste("cell", i, j, sep = "")
> splom.vpTree <- function(n) {
+     vplist <- vector("list", n^2)
+     for (i in 1:n)
+         for (j in 1:n)
+             vplist[[(i - 1)*n + j]] <-
+               viewport(layout.pos.row = i, layout.pos.col = j,
+                        name = cellname(i, j))
+     vpTree(viewport(layout = grid.layout(n, n), name = "cellgrid"),
+     do.call("vpList", vplist))
+ }
> cellpath <- function(i, j) vpPath("cellgrid", cellname(i, j))
> splom <- function(df, name = NULL, draw = TRUE) {
+     n <- dim(df)[2]
+     glist <- vector("list", n*n)
+     for (i in 1:n)
```

[2] **Warning:** As the number of `grob`s in a `gTree` gets larger the construction of the `gTree` will get slow. If this happens, the best solution is to just use a **grid** function rather than a `gTree`, and wait for me to implement some ideas for speeding things up!

```
+         for (j in 1:n) {
+             glist[[(i - 1)*n + j]] <-if (i == j)
+                 textGrob(paste("diag", i, sep = ""),
+                          gp = gpar(col = "grey"), vp = cellpath(i, j))
+             else if (j > i)
+                 textGrob(cellname(i, j),
+                          name = cellname(i, j),
+                          gp = gpar(col = "grey"), vp = cellpath(i, j))
+             else
+                 splot(df[,j], df[,i], "",
+                       name = paste("plot", i, j, sep = ""),
+                       vp = cellpath(i, j),
+                       gp = gpar(cex = 0.5), draw = FALSE)
+         }
+     smg <- gTree(name = name, childrenvp = splom.vpTree(n),
+                  children = do.call("gList", glist))
+     if (draw) grid.draw(smg)
+     smg
+ }
> df <- data.frame(x = rnorm(10), y = rnorm(10), z = rnorm(10))
> splom(df)

gTree[GRID.gTree.377]
```

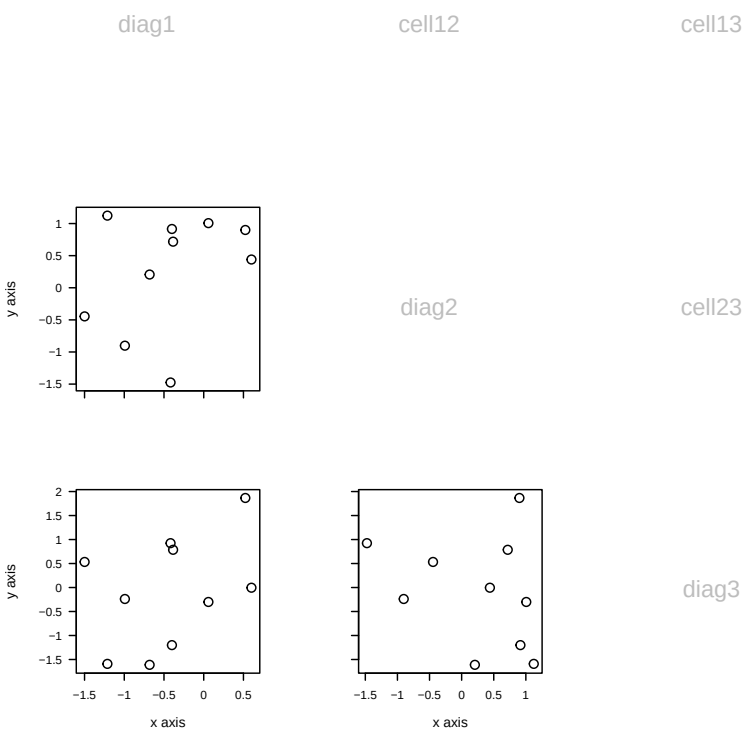


This `grob` can be edited as usual:

```
> splom(df)
```

```
gTree[GRID.gTree.381]
```

```
> grid.edit("plot21::xlab", label = "", redraw = FALSE)
> grid.edit("plot32::ylab", label = "", redraw = FALSE)
> grid.edit("plot21::xaxis", label = FALSE, redraw = FALSE)
> grid.edit("plot32::yaxis", label = FALSE)
```

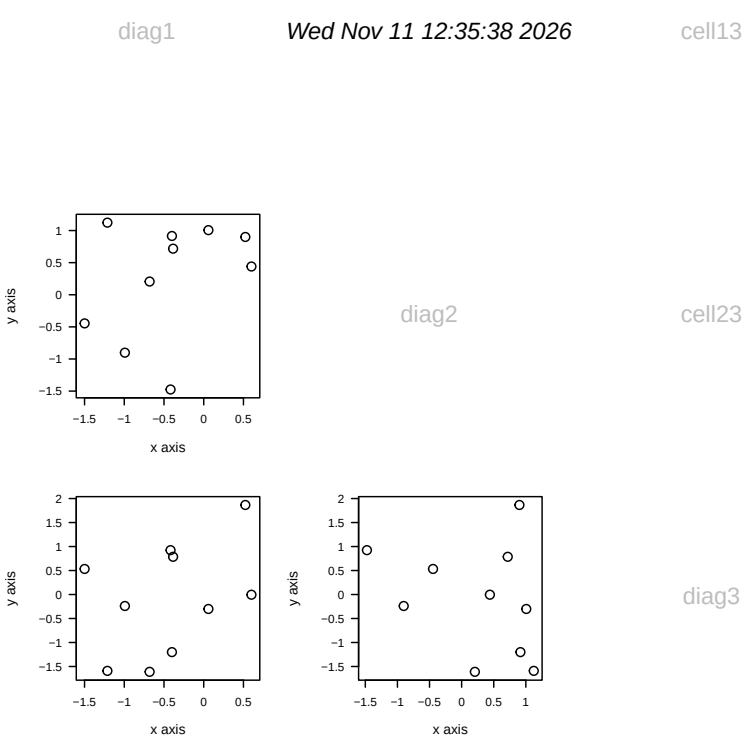


But of more interest, because this is a `grob`, is the *programmatic* interface. With a `grob` (as opposed to a function) it is possible to modify the description of what is being drawn via an API (as opposed to having to edit the original code). In the following, we remove one of the "spare" cell labels and put in its place the current date.

```
> splom(df, name = "splom")
```

```
gTree[splom]
```

```
> grid.remove("cell12")
> grid.add("splom", textGrob(date(), name = "date",
+                            gp = gpar(fontface = "italic"),
+                            vp = "cellgrid::cell12"))
```



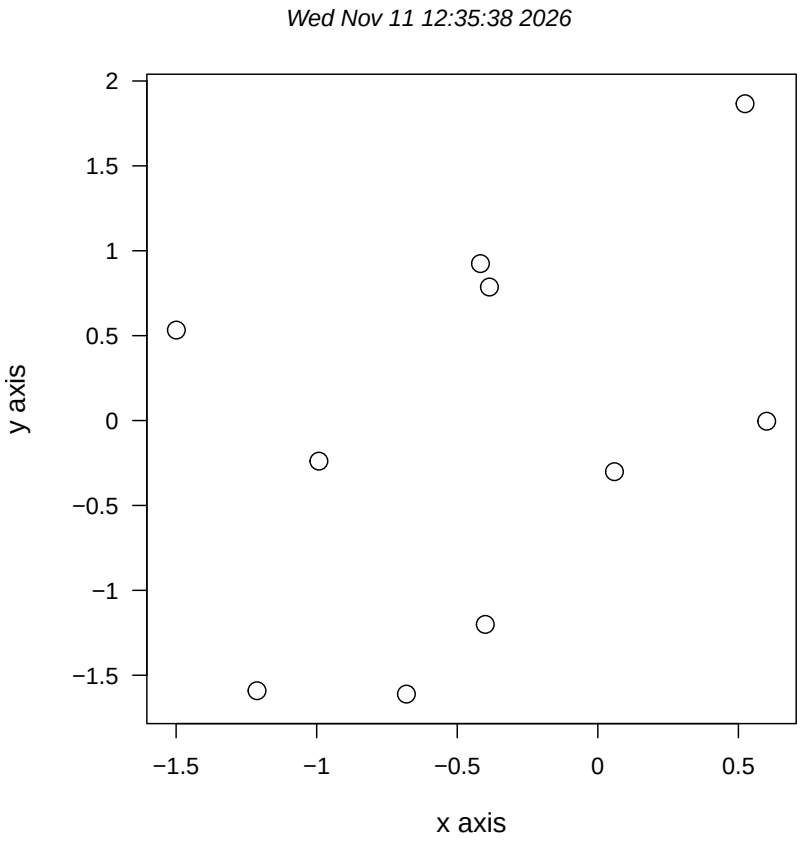
With the date added as a component of the scatterplot matrix, it is saved as part of the matrix. The next sequence saves the scatterplot matrix, loads it again, extracts the bottom-left plot and the date and just draws those two objects together.

```
> splom(df, name = "splom")
```

```
gTree[splom]
```

```
> grid.remove("cell12")
> grid.add("splom", textGrob(date(), name = "date",
+                            gp = gpar(fontface = "italic"),
+                            vp = "cellgrid::cell12"))
> smg <- grid.get("splom")
> save(smg, file = "splom.RData")
> load("splom.RData")
> plot <- getGrob(smg, "plot31")
> date <- getGrob(smg, "date")
> plot <- editGrob(plot, vp = NULL, gp = gpar(cex = 1))
> date <- editGrob(date, y = unit(1, "npc") - unit(1, "lines"), vp = NULL)
> grid.newpage()
> grid.draw(plot)
```

```
> grid.draw(date)
>
```



All of this may seem a bit irrelevant to interactive use, but it does provide a basis for creating an editable plot interface as used in M. Kondrin's **Rgrace** package (available on CRAN 2005–7).