



Current and Future Developments in R Graphics

Paul Murrell

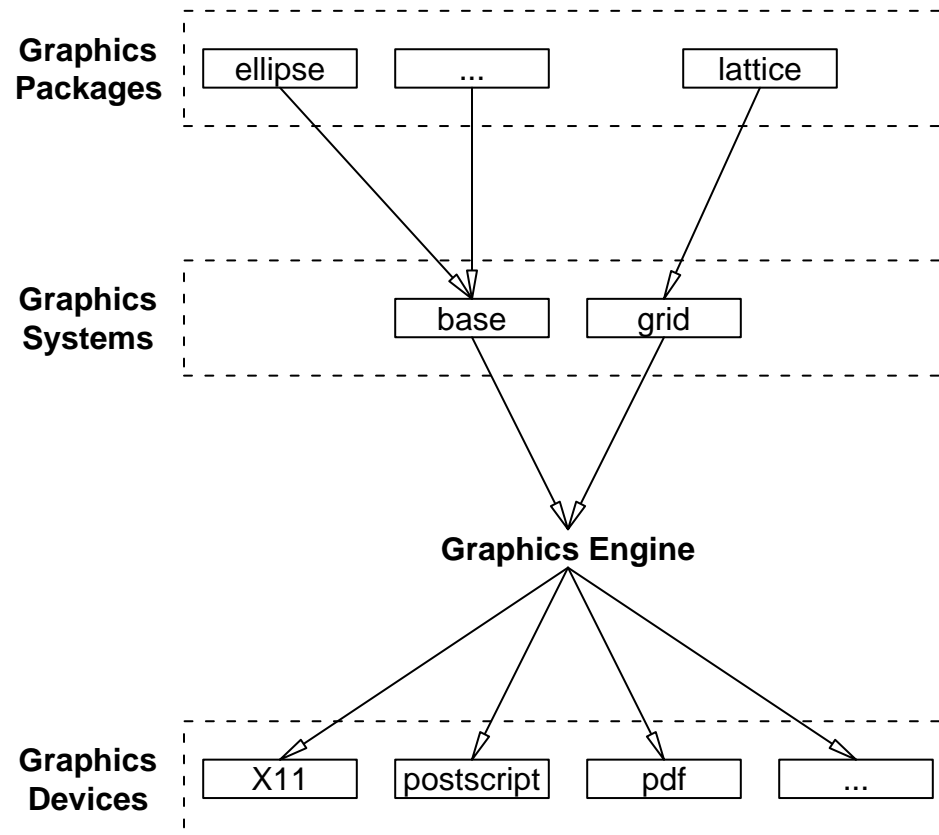
The University of Auckland

Tokyo 2003

General Remarks

Some of the R code examples in this talk may only run properly on the **development version** of R.

The Structure of R Graphics



Graphical Parameters in R

Graphical Parameters in R

R provides a basic set of graphical parameters for controlling the appearance of graphical output:

<code>col</code>	Colour of lines, text, ...
<code>lwd</code>	Width of lines
<code>lty</code>	Line type (solid, dashed, ...)
<code>font</code>	Font face (plain, bold, italic, ...)
<code>cex</code>	Font size multiplier
<code>lineheight</code>	Line Height (multiple of the font size)

There is a need to make this list much more complete.

Graphical Parameters in grid

The `grid` graphics system adds a few more standard parameters ...

<code>fill</code>	Fill colour for rectangles, circles, and polygons
<code>fontfamily</code>	Font family (Helvetica, Hershey, ...)
<code>fontsize</code>	Font size (points)
<code>alpha</code>	Alpha transparency

... but not all of these are fully supported and there are still some important ones missing.

<code>linecap</code>	Should the end of a line be “round” or “square”?
<code>linejoin</code>	Should a join be “smooth” or “pointy”

Fonts in R

Specifying a **font family** can be quite complicated and depends a lot on the platform and/or output format.

It would be nice if the **fontfamily** parameter was device-independent. One way to achieve this is to have a “font database” for each device to translate the font family specification into a device-specific format.

```
grid.text("Hello", gp=gpar(fontfamily="sans"))

ps.options(fontdb=list(sans="Helvetica", serif="Times",
                      symbol="Symbol", mono="Courier"))

x11.options(fontdb=list(sans="*-helvetica-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*",
                       serif="*-times-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*",
                       symbol="*-symbol-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*",
                       mono="*-courier-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*"))
```

Specifying Graphical Parameters

The `base` graphics system provides the `par()` function for “permanently” setting graphical parameters.

It is also possible to specify graphical parameters “temporarily” by providing them as arguments to individual functions.

This approach has several problems:

- It is difficult to specify where a setting should apply when using the `par` function; there has consequently been an “explosion” of parameters
- The implementation of graphical parameters for individual functions has become inconsistent.

Problems with par

There is a parameter for each part of a graph.

fg	"foreground" colour
col	plotting symbols
col.axis	axis tick labels
col.lab	axis labels
col.main	plot titles
col.sub	plot sub-titles

This can make it difficult to affect all of a graph.

```
> par(fg = "grey", col = "grey", col.axis = "grey", col.lab = "grey",  
+     col.main = "grey", col.sub = "grey")  
> plot(1:10, main = "My Plot", sub = "Subtitle")
```

This approach is not extensible to new structures.

Inconsistent parameters

- `rect()`: `col` gives the fill colour and `border` gives the edge colour.

```
> rect(0, 0, 1, 1, border = "red", col = "blue")
```

- `points()`: `col` gives the edge colour for plotting symbols and `bg` gives the fill colour for open plotting symbols.

```
> points(0:1, pch = 22, col = "red", bg = "blue")
```

- `box()`: `col` gives the edge colour and there is no way to specify the fill colour.

Specifying Graphical Parameters in grid

In `grid` there is a fixed set of graphical parameters. The `gpar()` function is used to specify graphical parameter settings.

```
> gpar(col = "red", fill = "blue")
$col
[1] "red"

$fill
[1] "blue"

attr("class")
[1] "gpar"
```

All functions have a `gp` argument which can be the result of a call to `gpar`.

```
> grid.rect(gp = gpar(col = "red", fill = "blue"))
```

Specifying Graphical Parameters in grid

Graphical parameter settings are “inherited”.

```
> pushViewport(viewport(gp = gpar(fill = "blue")))
viewport [GRIDVP1]
> grid.rect(gp = gpar(col = "red"))
> popViewport()
viewport [ROOT]
```

This approach allows graphical parameter settings to affect either just a part of a graph, or a whole image.

This approach also extends to new structures.

Interactive Graphics in R

Interactive Graphics in R

R graphics provides a lot of power and flexibility for producing **presentation graphics** (i.e., static plots).

R **base** graphics in particular are not designed to support any serious interactive behaviour.

- `identify()`
- `locator()`

R's graphics structure is designed for generality (multiple devices) and to be programmed at the user (interpreted) level. This means that it is **slower** than other systems and makes it unsuitable for highly dynamic applications.

Interactive Graphics in Other Software

One approach to providing interactive graphics in R is to “leave it to the experts”; use R for numerical computations and communicate with third-party software to produce interactive graphical displays.

- GGobi

<http://www.ggobi.org/>

- iPlots

<http://www1.math.uni-augsburg.de/iPlots/>

Dynamic Graphics in SVG

Another approach is to use R only to **describe** an image and offload the **rendering** to third-party software.

The **gridSVG** package converts a **grid** description of an image into an SVG document.

<http://www.stat.auckland.ac.nz/~paul/>

```
> pushViewport(viewport(gp = gpar(col = "black", fill = NA)))
> t <- seq(0, 2 * pi, length = 30)
> x <- 20 * cos(t)
> v <- -20 * sin(t)
> grid.circle(x = grid.animate(unit(x, "native"), duration = 5,
+   rep = TRUE), y = grid.animate(unit(v, "native"), duration = 5,
+   rep = TRUE), r = unit(1, "mm"), gp = gpar(fill = "black"))
> popViewport()
> gridToSVG()
```


Interactive Graphics in grid

The design of the `grid` graphics system provides greater support (than the `base` system) for interactive graphics.

`grid` graphics functions return a `graphical object` as well as producing graphical output. This object can be edited in order to modify the graphical output.

```
> gr <- grid.rect()
> grid.get(gr, "width")
[1] 1npc
> grid.edit(gr, width = unit(0.5, "npc"))
NULL
> grid.get(gr, "width")
[1] 0.5npc
```

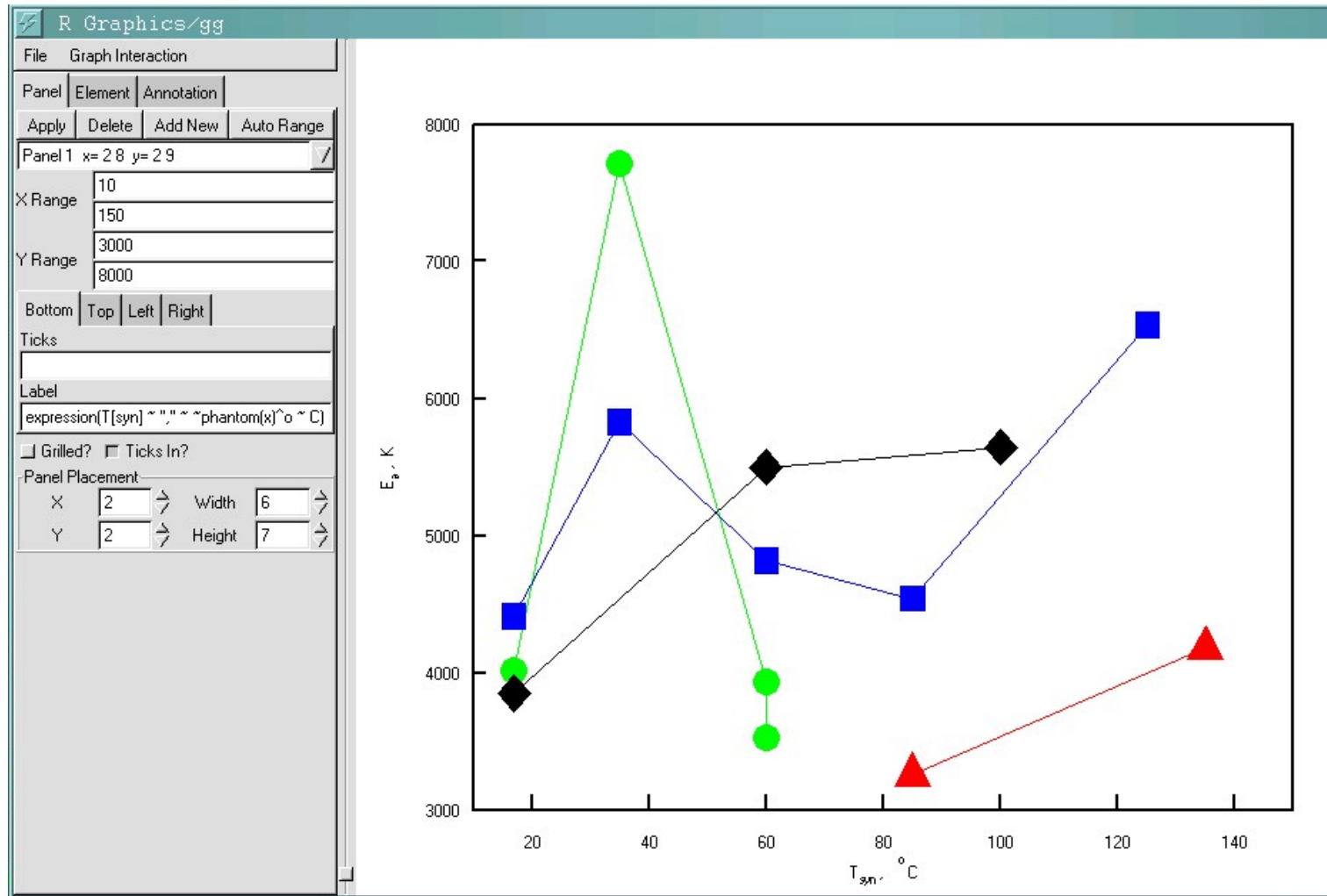
Interactive Graphics in grid

M Kondrin has made use of this feature of `grid` to develop an interactive graph editor for simple plot types.

This is available as an R add-on package called `Disgrace`.

<http://www.hppi.troitsk.ru/Kondrin/disgrace.htm>

Interactive Graphics in grid



Integrating grid and base Graphics

Integrating grid and base Graphics

R now has two distinct graphics systems: the “traditional” S graphics system (what I call **base** graphics); and the new **grid** system.

grid and **base** each maintain a separate set of coordinate systems. This makes it difficult to combine output from the two graphics systems.

For example, calls to **base** graphics functions within a **lattice** panel function do not work.

```
> data(ethanol)
> EE <- equal.count(ethanol$E, number = 9, overlap = 1/4)
> xyplot(NOx ~ C | EE, data = ethanol, panel = function(x, y, ...) {
+   panel.xyplot(x, y)
+   abline(h = 2)
+ })
```

← →

Integrating grid and base Graphics

It would be nice to be able to combine the output from `base` and `grid` graphics:

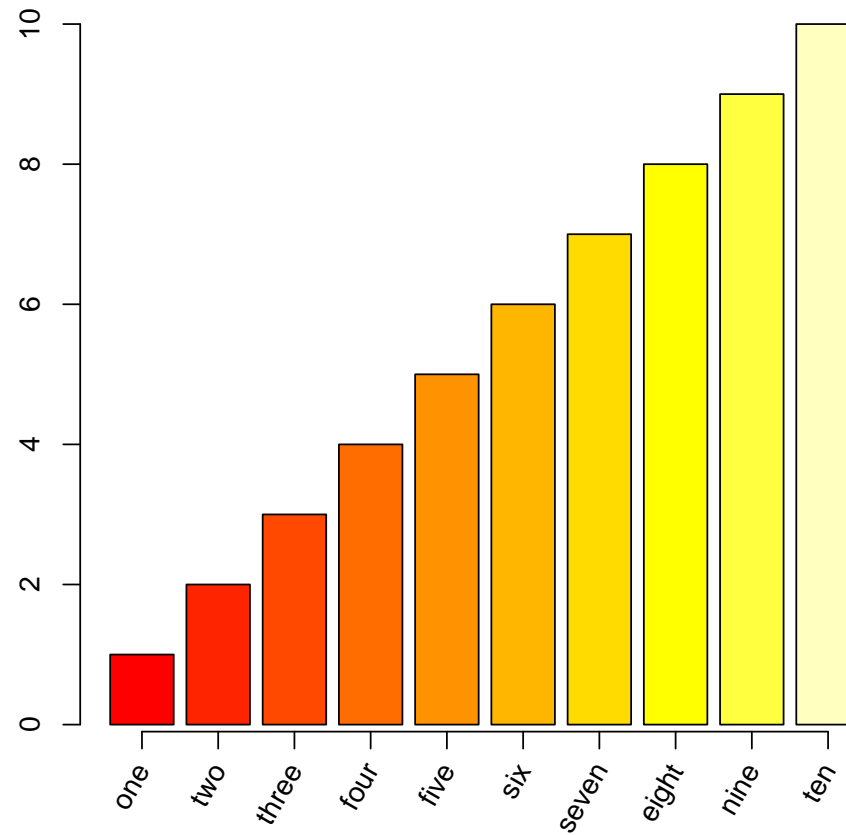
- to be able to annotate `base` plots using `grid`'s `units`.
- to be able to arrange `base` plots using `grid`'s `viewports`.

The gridBase package

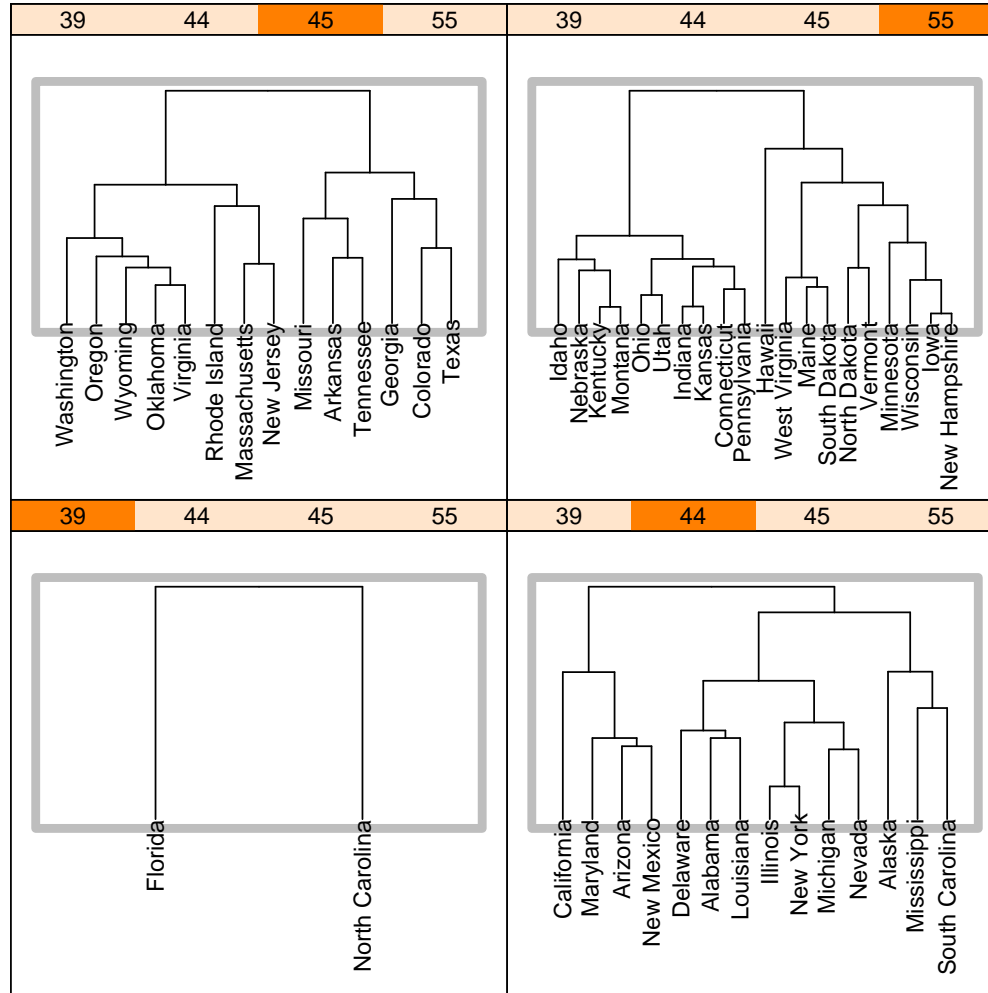
The `gridBase` package (available from [CRAN](#)) provides functions for aligning the `grid` and `base` coordinate systems.

```
> midpts <- barplot(1:10, axes = FALSE)
> axis(2)
> axis(1, at = midpts, labels = FALSE)
> vps <- baseViewports()
> par(new = TRUE)
> pushViewport(vps$inner, vps$figure, vps$plot)
> 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)
```

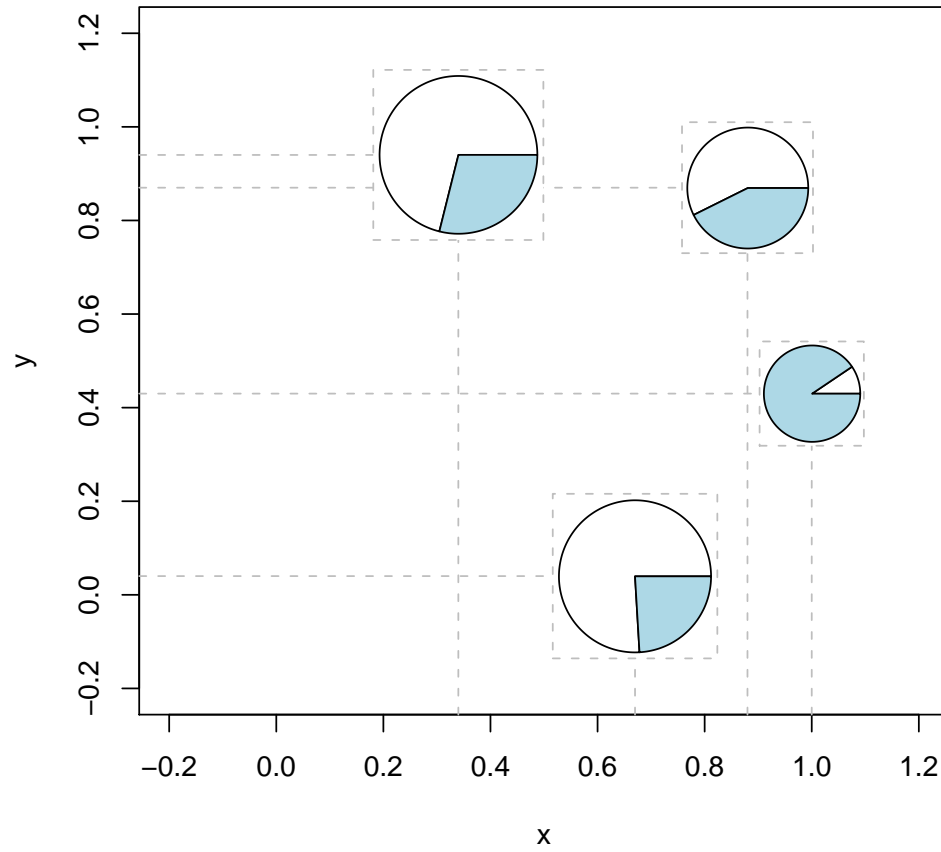
The gridBase package



The gridBase package



The gridBase package



Display Lists in R

Display Lists in R

The `graphics engine` maintains a display list; a record of all graphical output.

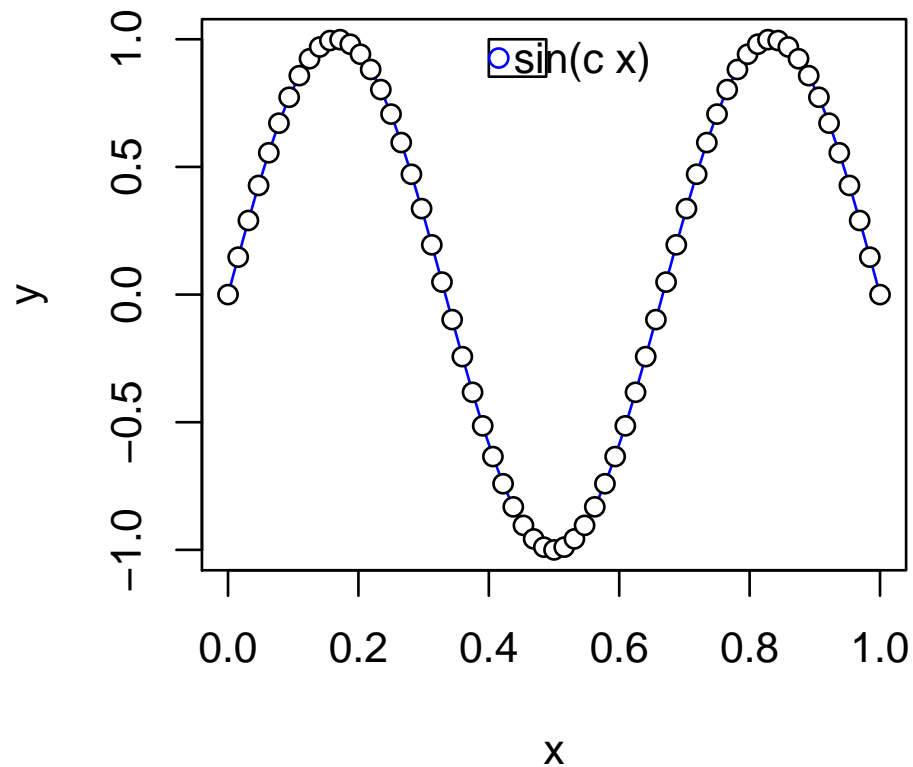
The display list is used for redrawing output when a device is resized or when copying between devices.

The `graphics engine`'s display list has some severe limitations:

- graphical output is only recorded at the lowest level (lines, rectangles, ...), not at a high level (axes, plots, ...).
- calculations leading up to graphical output are not recorded. For example, any output from the `legend()` function does not resize well.

Resizing legend() output

The `legend()` function performs calculations to locate and draw a border around the legend. If the device is resized, these calculations become incorrect.



The grid display list

The `grid` graphics system maintains its own display list. This is used for redrawing when a `grid` graphical object is edited and has fewer limitations:

- graphical output is recorded at a high level (axes, plots, ...). This is used by the `gridSVG` package to associate attributes (hyperlinks, transparency) with high-level objects in an image.
- calculations leading up to graphical output can be recorded. If an object has a `draw.details` method, that will be run when the display list redraws an image.

gridSVG hyperlinks

The following code associates a **URL** with a complete set of data points.

```
> grid.hyperlink(grid.points(iris$Sepal.Length, iris$Sepal.Width),  
+ href = "http://130.203.156.63/chu/course/iris.htm#iris1")
```

Safely resizing legend() output

If the `grid` display list is used and the legend function is placed within a `draw.details` method then the output will survive a device resize.

```
> draw.details.mylegend <- function(grob, grob.wrapped, recording) {  
+   x <- 0:64/64  
+   y <- sin(3 * pi * x)  
+   plot(x, y, type = "l", col = "blue")  
+   points(x, y, pch = 21, bg = "white")  
+   legend(0.4, 1, "sin(c x)", pch = 21, pt.bg = "white", lty = 1,  
+         col = "blue")  
+ }  
> engine.display.list(FALSE)  
NULL  
> dev.control(displaylist = "inhibit")  
> grid.grob(list(), "mylegend")
```


The grid display list

Unfortunately, the **grid** display list is **slower** than the **graphics engine's** display list.

Saving Graphics in R

Saving Graphics in R

R graphics can be “saved” in several different ways:

- An R code script

```
plot(1:10)
```

- A device-dependent format (PostScript, PDF, ...)

```
> postscript()  
> plot(1:10)  
> dev.off()
```

- Saving the display list

```
> plot(1:10)  
> myplot <- recordPlot()  
> save(myplot, file = "myplot.Rdata")
```

Saving Graphics in R

There are downsides to each of these:

- An R code script is easy to reproduce and it can be edited further, but it is difficult to interact with programmatically.
- A device-dependent format (PostScript, PDF, ...) is not (fully) editable.
- The display list is a binary, undocumented format with no API.

Saving Graphics in grid

WARNING: example code from here on will not work in ANY publicly available version of `grid`

`grid` produces objects representing the drawn image, which can be saved.

These are subtly different from an R script; instead of a “procedural” description of a graph, they provide an “object-oriented” description.

An object-oriented description of a graph is easier to **programmatically** query, edit, and generally interact with.

Saving Graphics in grid

An R script provides a procedural description.

```
pushViewport(plotViewport(c(5, 4, 4, 2), name="plotRegion"))
grid.rect()
pushViewport(viewport(name="dataRegion"))
grid.xaxis()
grid.yaxis()
popViewport(2)
```

Modifying the image must occur “manually”.

```
pushViewport(plotViewport(c(5, 4, 4, 2), name="plotRegion"))
grid.rect()
pushViewport(viewport(name="dataRegion"))
grid.xaxis()
grid.yaxis()
  grid.points()
popViewport(2)
```

Saving Graphics in grid

A `grid` object provides an object-oriented description.

```
> gplot <- gTree(childrenvp = vpTree(plotViewport(c(5, 4, 4, 2),
+   name = "plotRegion"), vpList(viewport(name = "dataRegion"))),
+   children = gList(xaxisGrob(vp = "plotRegion::dataRegion"),
+     yaxisGrob(vp = "plotRegion::dataRegion"), rectGrob(vp = "plotRegion")))
> grid.draw(gplot)
> save(gplot, file = "gplot1")
```

An API can be provided for interacting with this object.

```
> load("gplot1")
> gplot <- addGrob(gplot, pointsGrob(vp = "plotRegion::dataRegion"))
> grid.newpage()
> grid.draw(gplot)
```