The Grid Graphics Package

Version 0.5

Table of Contents

1 Introduction

Grid graphics is an alternative graphics interface to the base R graphics (`plot()`, `par()`, `lines()`, ...). It is designed to allow greater low-level power and customisation for graphics. Grid itself mostly consists of the *basis* for producing flash high-level plots; there are a couple of examples of higher-level plotting functions (e.g., `grid.xaxis` and `grid.yaxis`, and more spectacularly, `grid.show.layout` and `grid.show.viewport`). The `lattice` package provides plenty of examples of what can be achieved.

This user guide focuses on the low-level functionality; the purpose of this package is to provide greater power and flexibility for producing plots when the standard high-level plots are not sufficient.

This guide provides an explanation of the system through examples. It is best used while sitting at a console with R running.

A more detailed and complete description of individual functions is available in the form of Rd files (see `help(Grid)` and `library(help=grid)`).

2 Getting Started

Load the Grid library with the command `library(grid)`.

3 Tutorial

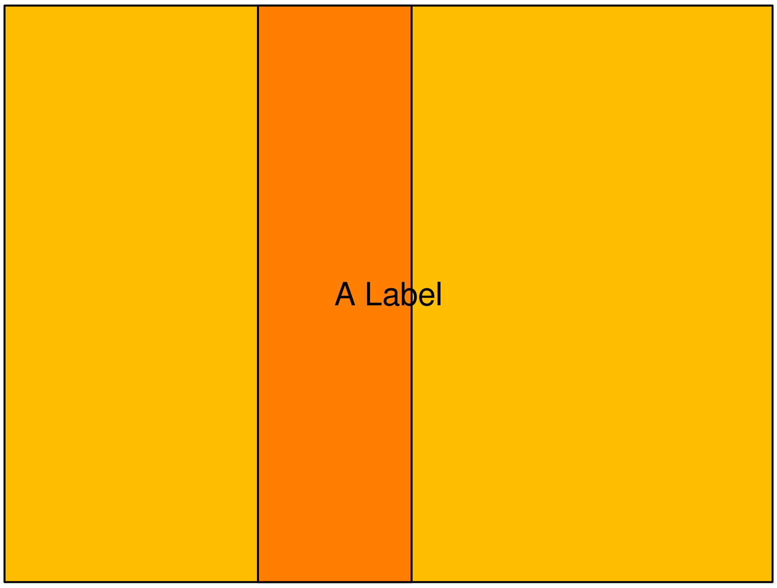
We will now look at some very simple Grid graphics drawing functions. This section assumes that you have loaded Grid graphics.

3.1 Baby Steps

First of all, try typing the following commands:

```
grid.rect(x=0.5, y=0.5, h=1, w=1,
          just="centre", gp=gpar(border=NULL, fill="#FFBF00"))
grid.rect(x=0.33, y=0, h=1, w=0.2,
          just=c("left", "bottom"), gp=gpar(border=NULL, fill="#FF8000"))
grid.text(x=0.5, y=0.5, "A Label")
```

The window is now a light orange colour with a darker orange vertical strip and the text '`A Label`' in the centre.



The first rectangle is centred half way across and half way up the device (`just = "centre"`, `x = 0.5`, and `y = 0.5`) and is as wide as and as high as the device (`w = 1` and `h = 1`). In other words, it occupies the entire device.

The second rectangle has its bottom-left corner at the bottom of the device and about a third of the way across (`just = c("left", "bottom")`, `x = 0.33`, and `y = 0`) and is as high as the device, but only one fifth as wide (`w = 0.2` and `h = 1`).

These locations and dimensions are all defined as proportions of the device; try resizing the window and notice that the text remains centred, while the location and width of vertical strip changes as the width of the device changes.

In grid, there are a number of coordinate systems available; this one is known as Normalised Parent Coordinates (NPC). It is not called Normalised *Device* Coordinates because drawing is not always relative to the entire device surface. In fact, drawing always occurs within a structure called a *viewport*.

A viewport is a region on a graphics device which has a number of coordinate systems associated with it. Grid has a default viewport, which occupies the entire device region. It is possible to specify which viewport you want to draw into, but if you do not, Grid just uses the default viewport.

The following code draws the same rectangles and text as in the previous example, but within a viewport that occupies only part of the device.

```
grid.newpage()
vp <- viewport(x=0.5, y=1, w=1, h=unit(1, "lines"),
                    just=c("centre", "top"))
push.viewport(vp)
grid.rect(gp=gpar(border=NULL, fill="#FFBF00"))
grid.rect(x=0.33, y=0, h=1, w=0.2,
          just=c("left", "bottom"),
          gp=gpar(border=NULL, fill="#FF8000"))
grid.text(x=0.5, y=0.5, "A Label")
pop.viewport()
```

The screen probably looks a bit of a mess. Type `grid.newpage()` and then enter the above commands again. Grid does not have the concept of a '`plot`' so you have to clear the device with a call to `grid.newpage()` to start again with a blank device.

The same rectangles are now drawn within a viewport which is only 1 line of text high (`h = unit(1, "lines")`) and flush with the top of the device (`just = c("centre", "top")`). The function `push.viewport` is used to set the current viewport; the function `viewport` is used to create a viewport description. Grid maintains a stack of viewports and we "pop" the viewport at the end to restore the system to the state it was before we started drawing.

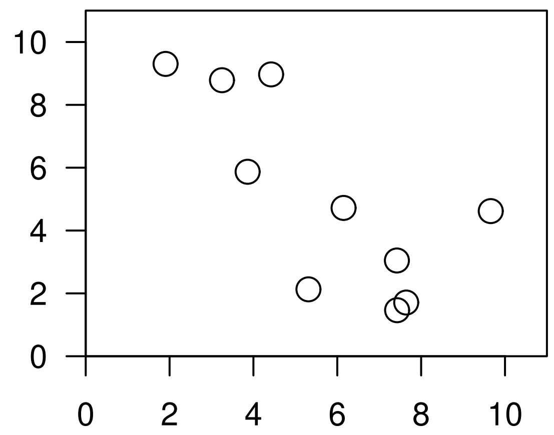
This example demonstrates that drawing is relative to the viewport rather than the device. For example, the text is centred within the viewport. This also demonstrates how you specify a unit or coordinate system explicitly using the `unit` function. In `"lines"` units, locations and dimensions are multiples of the height of one line of text.

Notice also that the first call to `grid.rect` has not specified the values of `x`, `y`, `w`, `h`, and `just`. This is because the values we want are actually just the default values supplied by the `grid.rect` function.

3.2 A Simple Plot

Now try the following commands:

```
vp <- viewport(x=0.5, y=0.5, w=0.6, h=0.6,
                    xscale=c(0, 11), yscale=c(0, 11))
push.viewport(vp)
grid.rect()
grid.points(runif(10, 1, 10), runif(10, 1, 10))
grid.xaxis()
grid.yaxis()
pop.viewport()
```



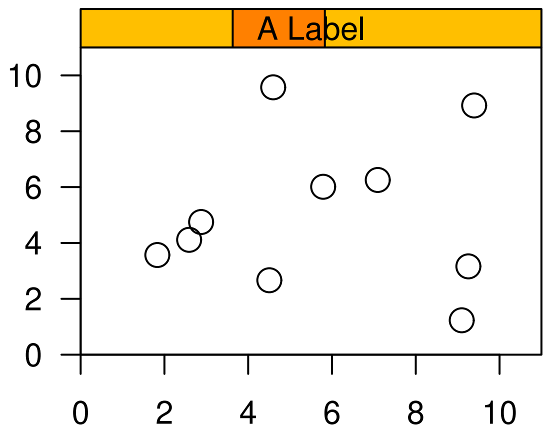
This produces a basic plot, with axes and data points. Notice again that drawing is relative to a viewport. We also see a third Grid coordinate system called `"native"` coordinates. The viewport has x- and y-scales defined and the `grid.points` function uses these '`native`' scales to locate the data points within the viewport. Similarly, `grid.xaxis` and `grid.yaxis` use the scales to locate their tick marks.

Note that your plot will not look exactly like the one shown because of the use of `runif` to generate random data.

3.3 A Taster

So far, we have dealt with drawing within a single viewport. Now try the following code:

```
vp1 <- viewport(x=0.5, y=0.5, w=0.6, h=0.6,
                     layout=grid.layout(2, 1,
                                        heights=unit(c(1, 1),
                                                     c("lines", "null"))))
push.viewport(vp1)
vp2 <- viewport(layout.pos.row=1)
push.viewport(vp2)
grid.rect(x=0.5, y=0.5, h=1, w=1,
          just="centre",
          gp=gpar(border=NULL, fill="#FFBF00"))
grid.rect(x=0.33, y=0, h=1, w=0.2,
          just=c("left", "bottom"),
          gp=gpar(border=NULL, fill="#FF8000"))
grid.text(x=0.5, y=0.5, "A Label")
pop.viewport()
vp3 <- viewport(layout.pos.row=2, xscale=c(0, 11), yscale=c(0, 11))
push.viewport(vp3)
grid.rect()
grid.points(runif(10, 1, 10), runif(10, 1, 10))
grid.xaxis()
grid.yaxis()
pop.viewport(2)
```



Here we have a top-level viewport (`vp1`) with two other viewports nested within it (`vp2` and `vp3`). Drawing occurs within the lower-level viewports and these viewports are positioned within the top-level viewport.

The positioning of the lower-level viewports can be controlled by `x`, `y`, `width`, `height`, and `just` as in previous examples, but in this case, the positioning is controlled by a *layout* (which is a way of cutting up a viewport into subregions). The top-level viewport defines a layout with 2 rows and 1 column, where the top row is 1 `"lines"` unit high and the second row just expands to fill the remaining space. The first low-level viewport occupies the first row of the layout (`layout.pos.row=1`) and the second low-level viewport occupies the second row of the layout (`layout.pos.row=2`).

It is possible to nest viewports to more than two levels, there are other coordinate systems not described here, and there are many different features of layouts to explore. Some idea of what can be done is given by the output of the `grid.show.viewport` and `grid.show.layout` functions. Try typing `example(Grid)` to see examples of this stuff.