Traditional `S` Graphics

Paul Murrell

February 12, 2003

This document is *not* a description of high-level `S` plotting functions. It is a description of the underlying graphics system and is intended to provide information for modifying the output of high-level plotting functions, adding additional graphics to the output from high-level plotting functions, and producing complete plots using low-level plotting functions.

There are three things to know in order to construct a statistical graphic (plot):

1. How to create and control different graphics regions and coordinate systems.
2. How to control which graphics region and coordinate system graphics output goes into.
3. How to produce graphics output (lines, points, text, ...) including how to control its appearance (colour, line type, line width, ...).

High-level `S` plotting functions do all of this for you, but in order to produce customised plots you need to know a bit about what is going on in the background.

1 Creating and Controlling Graphics Regions and Coordinate Systems

A page of graphics output in `S` is split up into a number of different regions, each with one or more coordinate systems:

- Outer margins
- Figure regions
- Figure margins
- Plot regions

Figures 1 and 2 show how these regions are arranged. Figure 3 shows that there can be multiple figure regions on a page. These regions and their coordinate systems are created when a "new plot" is created.

There are three types of `S` graphics functions: the `par()` function, plot functions, and annotation functions. The `par()` function can be used to create and control all graphics regions in fine detail. Plot functions create entire plots so set up figure and plot regions. Annotation functions do not create graphics regions, but add graphics output to existing regions.

The standard usage of `S` graphics is to use `par()` to create a plot, and annotation functions to add further detail.

R Difference: In `S`, a default set of regions and coordinate systems is created if you use an annotation function on a fresh device. In `R`, this sort of action produces an error; i.e., you have to use a plot function first on a fresh device.

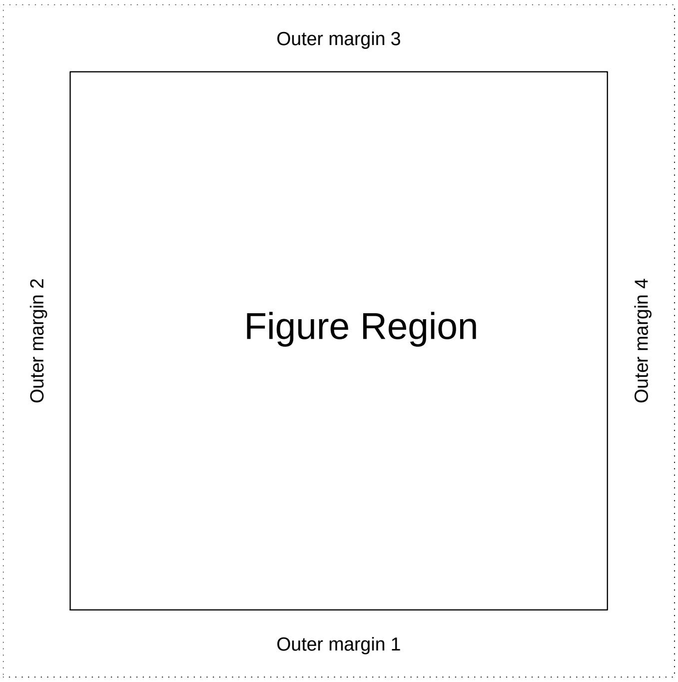

Figure 1: The outer margin and figure region in traditional S graphics.

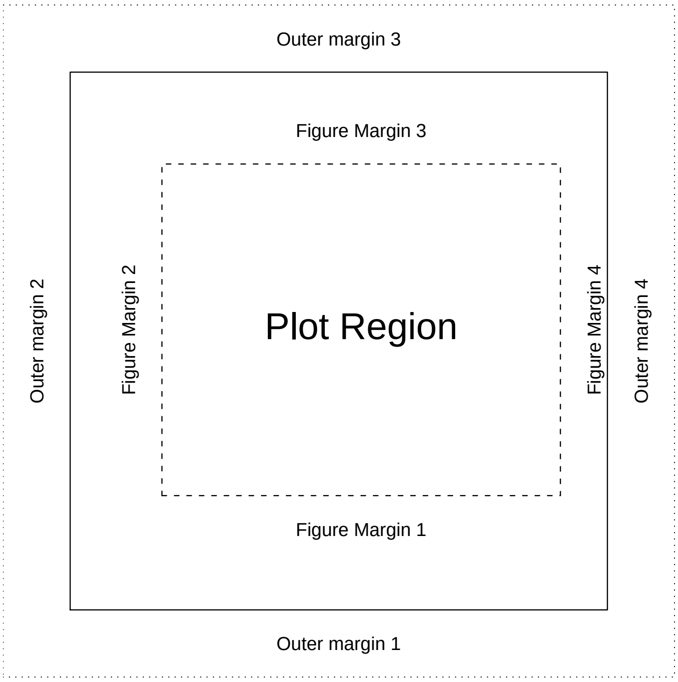

Figure 2: The figure margin and plot region in traditional S graphics.

1.1 Arranging Graphics Regions

The arrangement of the graphics regions is controlled via the `par()` function.

The outer margins are by default of zero dimension. They can be specified in terms of inches or as a multiple of a line of margin text. For example, `par(omi=rep(1, 4))` sets all outer margins to be 1 inch. `par(oma=1:4)` sets the outer margins to be one, two, three, and four lines of text respectively.

The figure margins are also specified in terms of inches or as a multiple of a line of margin text. These are set using the `par` arguments `mai` and `mar`, respectively.

The number of figures on a page is controlled using the `par` arguments `mfrow` and `mfcol`. For example, the arrangement in Figure 3 (3 rows and 2 columns) is specified as either `par(mfrow=c(3, 2))` or `par(mfcol=c(3, 2))`[1].

With the above arrangements, the figure region is determined automatically by the size of the outer margins and the number of figures. The figure region can be set manually by specifying it with one of the `par` arguments `fig` or `fin`. The former defines the figure region in terms of proportions of the region within the outer margins. For example, the following code allows for 1 inch outer margins and creates a figure region which occupies the left half of the space within those margins (Figure 4 shows a diagram of this arrangement):

```
> par(omi = rep(1, 4), fig = c(0, 0.5, 0, 1))
```

The `fin` argument specifies the figure region in terms of inches. The specified figure region is centred within the space left by the outer margins.

A useful trick to know in combination with these figure region specifications is the command `par(new=T)`. This command means that the next high-level plotting function will *not* clear the page. This can be used to produce arbitrary arrangements of figures. The following code illustrates the idea (see Figure 5):

> **R Difference:** In `R`, you need to do `par(new=TRUE)` between calls to `par(fig)`, but in S-Plus (6.1 at least), you do not need the `par(new=T)`. Also note the use of `TRUE` in `R` versus `T` in S-Plus.

> **R Difference:** `R` also has the `layout()` function for making complex arrangements of figure regions.

```
> par(fig = c(0.1, 0.6, 0.1, 0.6))
> plot(c(0, 1), c(0, 1), type = "n", xlab = "", ylab = "", axes = FALSE)
> box()
> text(0.5, 0.5, "Figure 1")
> par(new = T)
> par(fig = c(0.4, 0.9, 0.4, 0.8))
> plot(c(0, 1), c(0, 1), type = "n", xlab = "", ylab = "", axes = FALSE)
> box()
> text(0.5, 0.5, "Figure 2")
```

The plot region defaults to the figure region minus the figure margins. This can be overridden by specifying one of the `par` arguments `pin`, `plt`, or `pty`. The first of these specifies the plot region in inches and the second specifies the plot region in terms of proportions of the figure region. These act very much like the figure region counterparts. The third specifies the plot region in terms of its shape. For example, `par(pty="s")` specifies that the plot region should be "square" (i.e., the biggest square region within the space available).

[1] The difference between these two specifications is whether the figure regions are numbered counting first across the rows, or down the columns.

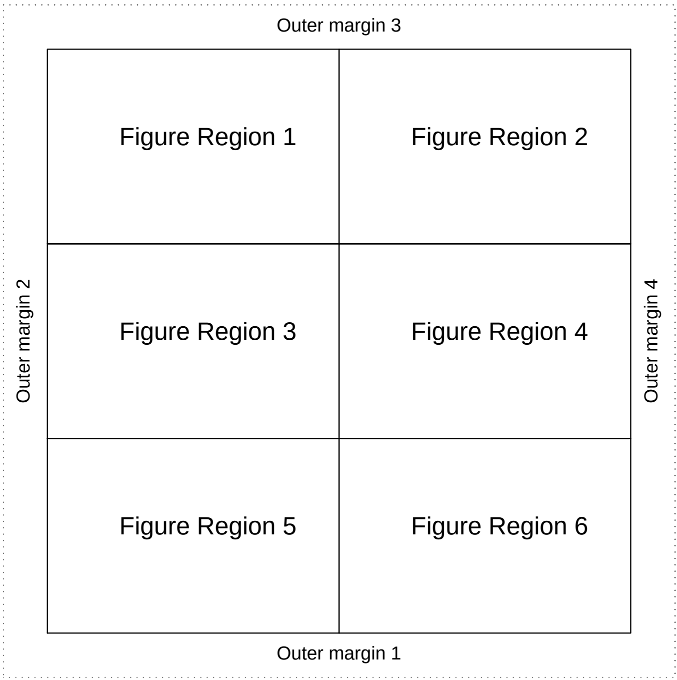

Figure 3: Multiple figure regions in traditional `S` graphics.

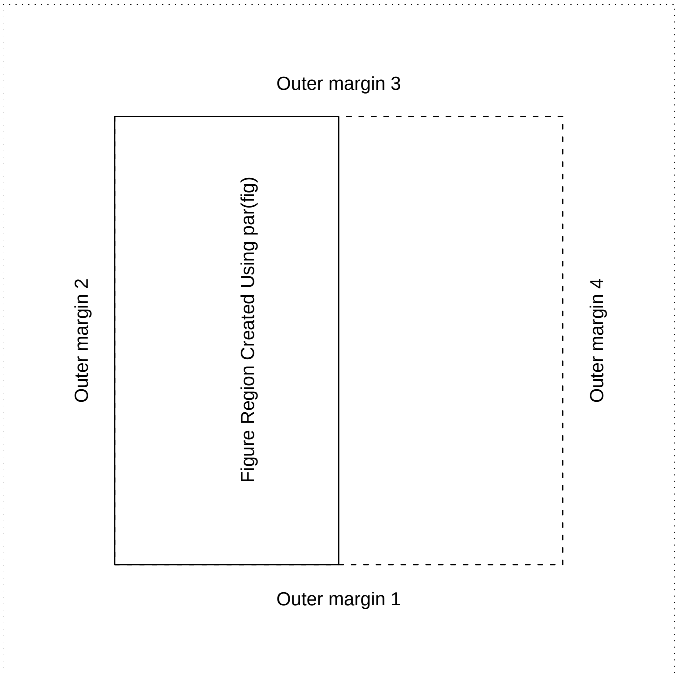

Figure 4: A figure region created using `par(fig)`.

1.2 Defining Coordinate Systems for Graphics Regions

Each region has one or more coordinate systems associated with it. The plot region is probably the easiest to understand.

The coordinate system in the plot region is referred to as "user coordinates". This corresponds to the range of values on the axes of the plot (see Figure 6). This coordinate system is controlled via the `par` parameters `xlim` and `ylim`. Typically, these are specified in a call to a high-level plotting function, for example,

```
> plot(x, y, xlim = c(0, 10), ylim = c(50, 100))
```

A small, but not insignificant, complication is introduced by the `par` argument `xaxs`. By default, the x- and y-axis ranges that you specify in `xlim` and `ylim` are not taken literally; they are extended to produce the final coordinate system. If you want your x- and y-ranges to be taken literally, then you should specify something like:

```
> plot(x, y, xlim = c(0, 10), ylim = c(50, 100), xaxs = "i", yaxs = "i")
```

Figure 7 demonstrates the different coordinate system specifications.

The figure margins contain the next most commonly-used coordinate systems. The coordinate systems in these margins are a combination of x- or y-ranges (like user coordinates) and lines of text. Figure 8 shows two of the possible figure margin coordinate systems.

There is a further set of figure margin coordinate systems available in which the x- and y-ranges are replaced with a normalised dimension. In other words, it is possible to specify locations along the axes as a proportion of the total axis length.

All of these figure margin coordinate systems are created implicitly from the arrangement of the figure margins and the setting of the user coordinate system.

The outer margins have similar sets of coordinate systems. Things are slightly complicated when there are multiple plots; the user-coordinate-based systems vary depending on the "current plot", but the normalised system always refers to the extent of the complete outer margin. Figure 9 shows the difference between these outer margin coordinate systems.

2 Directing Graphics Output into Different Graphics Regions and Coordinate Systems

In traditional `S` graphics, the graphics function you use determines which graphics region your output will go into. For example, the `text()` function produces text graphics in the plot region. Where there is more than one coordinate system within the region, function arguments are provided to determine which coordinate system is used. For example, the `mtext` function, which produces text graphics in the figure margins, will use user coordinates if the `at` argument is specified and normalised coordinates if the `adj` argument is specified.

Table 1 lists which functions can be used to produce output in each graphics region.

3 Producing Graphics Output

The most useful graphics functions for statistical graphics are the `axis()` and `points()` functions. These draw x-/y-axes and data symbols, respectively. There are also several functions for drawing even simpler graphics components, such as lines, text, polygons, etc ... (see Table 1).

3.1 Controlling the Appearance of Graphics Output

A large set of graphics parameters are provided to control the colour, line type, line width, and so on for graphics output. These can be specified via `par()` or "in-line" in the specific graphics function, e.g., `lines(1:2, 1:2, col=1)`. Table 2 lists common graphics parameters.

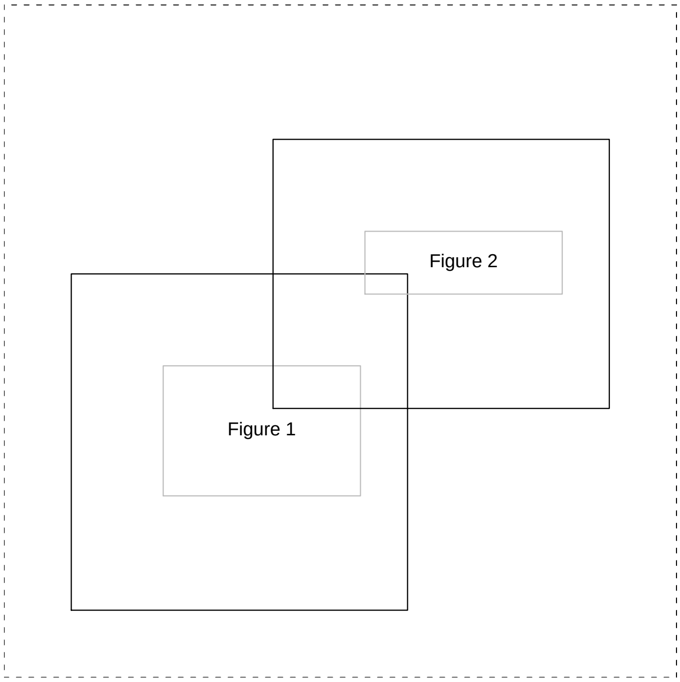

Figure 5: Arbitrary arrangements of figure regions using `par(fig)` and `par(new=T)`.

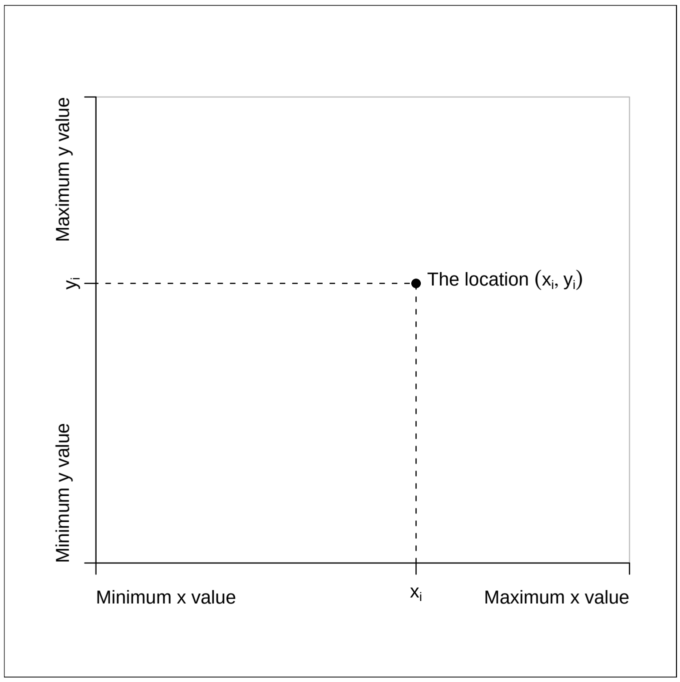

Figure 6: The user coordinate system in the plot region.

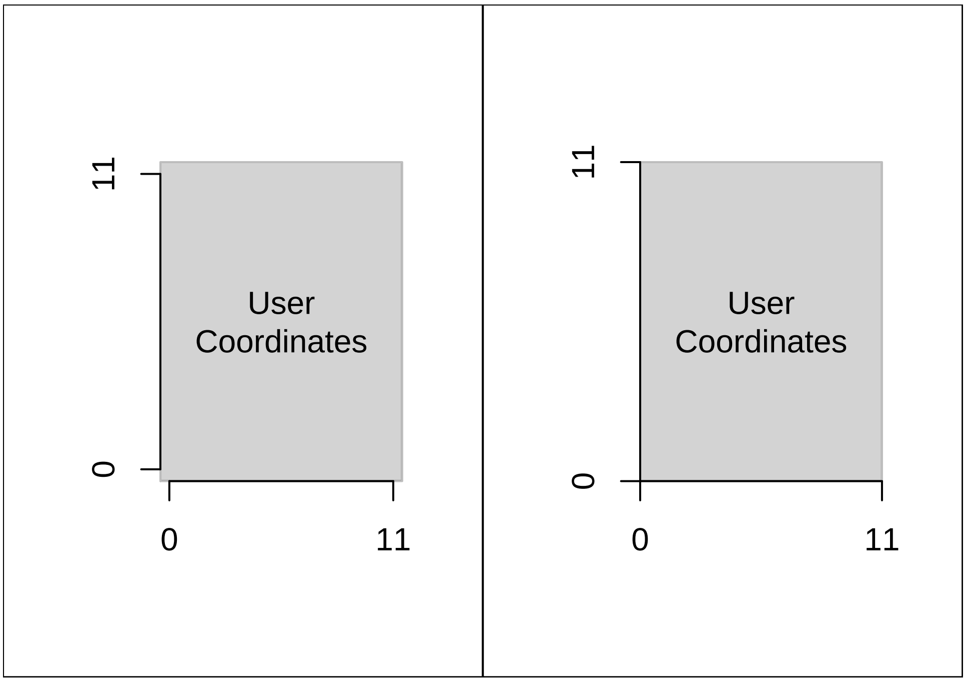

Figure 7: Two ways to specify the user coordinate system: the left-hand plot just uses `xlim=c(0, 11)` and `ylim=c(0, 11)`; the right-hand plot also specifies `xaxs="i"` and `yaxis="i"`.

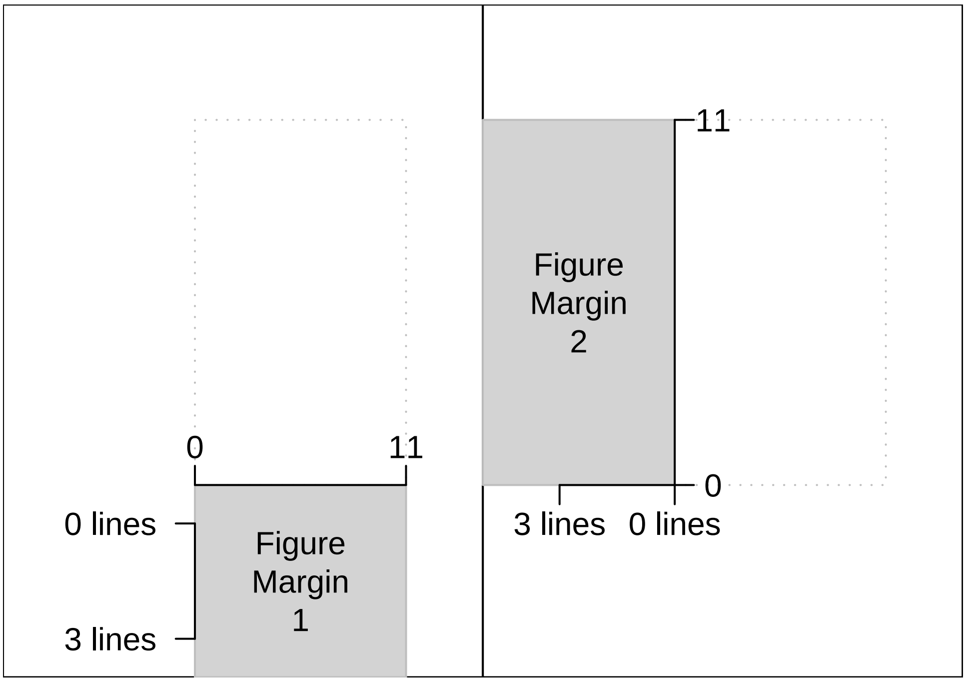

Figure 8: The typical coordinate systems for figure margin 1 (left-hand plot) and figure margin 2 (right-hand plot).

plot region	figure margins	outer margins
`text()`	`mtext()`	`mtext()`
`points()`	`axis()`	
`lines()`		
`arrows()`		
`polygon()`		
`segments()`		
`box()`		
`abline()`		

Table 1: `S` functions for different graphics regions.

R Difference: In `R`, the colours and line types can be specified as strings, such as `"red"` and `"dotted"`, as well as integers.

`col`	colour of lines, text, ...
`lwd`	line width
`lty`	line type
`font`	font face (plain, bold, italic)
`pch`	type of plotting symbol
`srt`	string rotation

Table 2: `S` functions for different graphics regions.

4 Adding to Existing Plots

High-level `S` plotting functions like `plot`, `barplot`, and `boxplot` set up the graphics regions and coordinate systems, and produce a lot of graphics output, but often we want to add a few extra annotations.

In simple cases, this is just a matter of using the functions in Table 1, but there are a number of complications.

4.1 Obscure User Coordinates

The `barplot` function does not display the x-scale that it sets up. This must be obtained by the function's return value. For example, the following code labels each of the bars with the amount being plotted (Figure 10):

```
> midpts <- barplot(1:10, density = -1)
> text(midpts, 0.1, 1:10, srt = 90, adj = 0)
```

The `pie` function provides even less information about its user coordinate system, but the user coordinates established for a plot can always be queried using `par("usr")`.

Another useful thing to be able to do is determine the relationship between physical and user coordinates. `par("uin")` returns the number of inches in one unit of user coordinates for both x- and y-dimensions. This can be used in calculations to position and size graphics in the plot region in terms of physical units.

4.2 Plots which Revert Graphics Region Settings

Some high-level plotting functions (e.g., `coplot()`) produce complex arrangements of multiple sub-plots and "undo" the graphics regions that they set up after they have produced their graphics

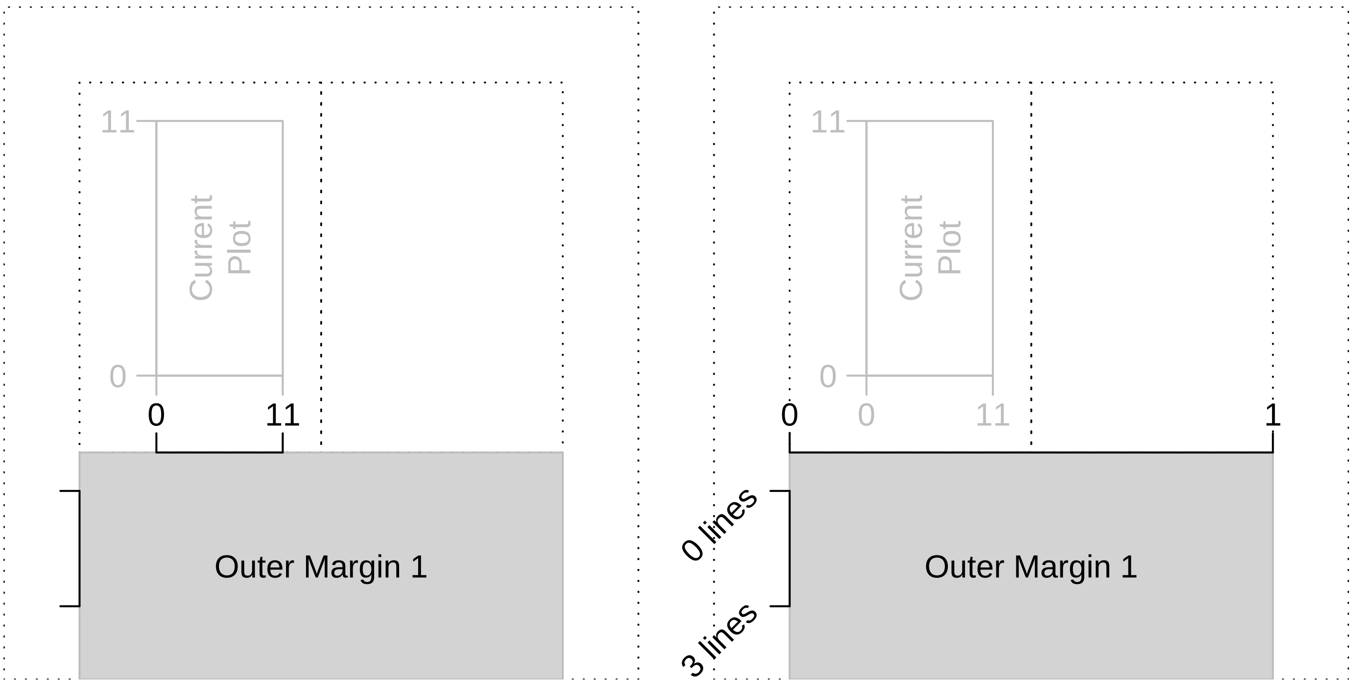

Figure 9: The two outer margin coordinate systems available with multiple plots: one based on user coordinates for the "current plot" (left-hand figure) and one based on normalised coordinates (right-hand figure).

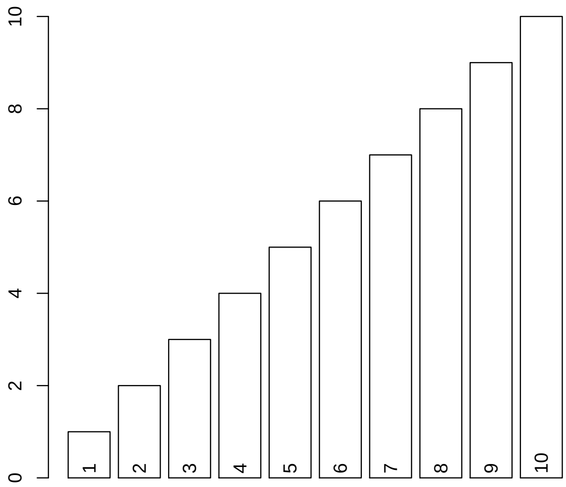

Figure 10: Annotating a barplot.

output. This makes it impossible to annotate elements of the complex plot once it is complete.

In this case, the typical solution is to allow the user to specify a "panel function". This is a function that gets executed during the drawing of each sub-plot, and thereby has access to the graphics regions and coordinate systems for that sub-plot. The following code adds grid lines to each panel (see Figure 11).

```
> E.intervals <- co.intervals(ethanol$E, 16, 0.25)
> coplot(NOx ~ C | E, given.values = E.intervals, data = ethanol,
+     panel = function(x, y, col, pch) {
+         points(x, y, cex = 1.5)
+         axis(1, tck = 1, lty = 2, labels = F)
+         axis(2, tck = 1, lty = 2, labels = F)
+     })
```

This approach is the common method for annotating Trellis plots.

5 Plots from First Principles

Sometimes it is not possible to achieve an effect by starting with the graphics output from a high-level plot. In such cases, the `plot()` function can be used simply to establish the graphics regions and coordinate systems, then all of the graphics output can be produced from "first principles", using only annotation functions.

As a trivial example, here is the `S` code for producing the same result as `plot(1:10)` (Figure 12) from first principles:

```
> par(omi = rep(0, 4), mar = c(5.1, 4.1, 4.1, 2.1), mfrow = c(1,
+     1))
> plot(0, type = "n", xlim = c(0, 10), ylim = c(0, 10), axes = F,
+     xlab = "", ylab = "")
> par(col = 1, lty = 1, lwd = 1, cex = 1, srt = 0)
> box()
> axis(1)
> axis(2)
> points(1:10)
> mtext("1:10", side = 2, line = 3)
```

Now consider a more complex example, where we want to create a barplot with a custom legend. Figure 13 shows what we want to end up with.

First of all, we reserve most of the page for the barplot, but leave room for the legend.

```
> par(fig = c(0, 0.8, 0, 1), mar = c(4, 4, 4, 2))
```

Draw a "stacked bar" barplot, coding different "groups" using different densities of shading lines.

```
> barplot(matrix(sample(1:4, 16, replace = T), ncol = 4), angle = 45,
+     density = 1:4 * 10, col = 1)
```

Now, we stay on the same page and set up a region and coordinate system for the legend.

```
> par(new = T)
> par(fig = c(0.8, 1, 0, 1), mar = c(4, 0, 4, 2))
> plot(0, xlim = c(0, 1), ylim = c(0, 5), axes = F, xlab = "",
+     ylab = "", type = "n")
```

We want the sample squares in the legend to be 0.5 inches square, so we calculate those dimensions in terms of user coordinates.

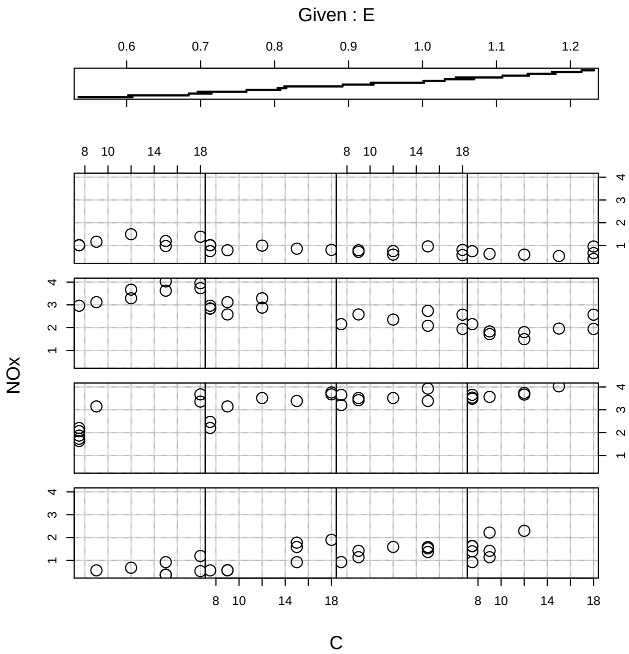

Figure 11: An annotated coplot.

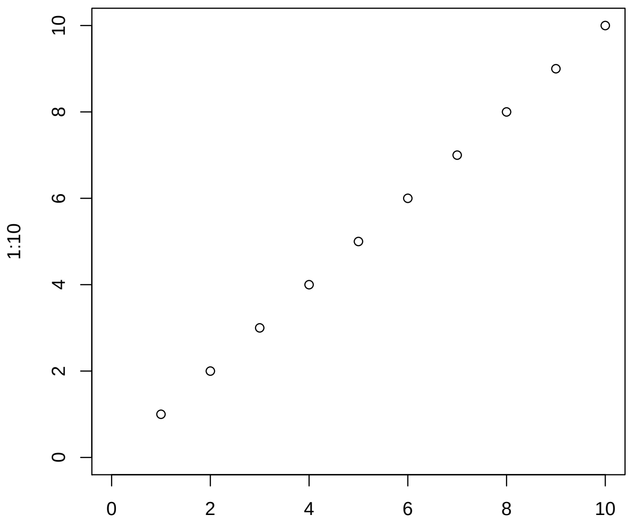

Figure 12: The result from `plot(1:10)`.

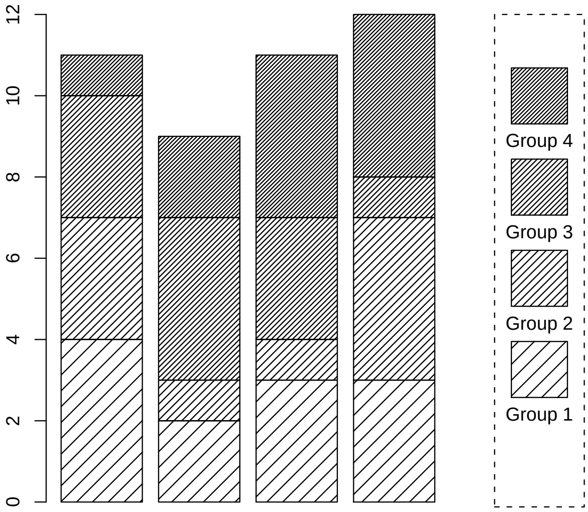

Figure 13: A barplot with a custom legend.

```
> size <- par("cxy")/par("cin") * 0.5
```

Finally, we draw the sample squares and labels using `polygon()` and `text()`, and give the legend a dashed border.

```
> box(lty = 2)
> for (i in 1:4) {
+     polygon(c(0.5 - size[1]/2, 0.5 - size[1]/2, 0.5 + size[1]/2,
+         0.5 + size[1]/2), c(i, i + size[2], i + size[2], i),
+         angle = 45, density = i * 10)
+     text(0.5, i - 0.2, paste("Group", i))
+ }
```

6 Traditional Trellis Graphics

The suite of Trellis plots available in `S` provide two main advantages:

1. plots with a design principles based on human perception experiments (text is horizontal, colours and symbols with easily-distinguishable defaults, "banking" of plots, ...)

2. "multipanel conditioning", where multiple plots of x versus y are produced for different levels of a grouping variable g.

Trellis plots tend to have a more complicated layout than other traditional `S` plots, involving multiple plot regions and "strip" regions (see Figure 14). The following are possible ways to influence the layout of Trellis plots:

- To control how many columns and rows of plots there are use the `layout` argument. For example, the following code specifies that there should be two columns and four rows.

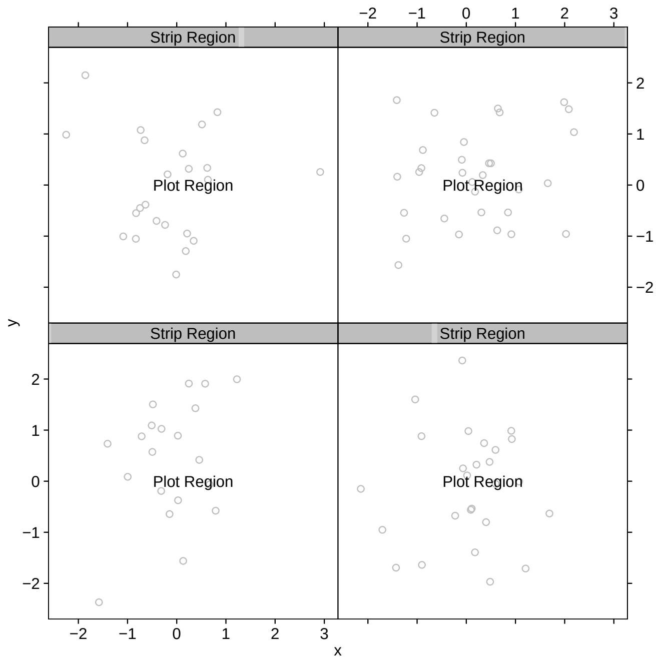

Figure 14: A multipanel Trellis plot.

```
> xyplot(y ~ x | g, layout = c(2, 4))
```

- To control what aspect ratio each plot has, use the `aspect` argument. For example, the following specifies that each panel should be half as high as it is wide:

```
> xyplot(y ~ x | g, aspect = 0.5)
```

- To control the position of the entire plot, save the result of the Trellis function and use the `position` argument to the `print.trellis` function. For example, the following specifies that the entire plot is arranged only in the top half of the page:

```
> myplot <- xyplot(y ~ x | g)
> print.trellis(myplot, position = c(0, 0.5, 1, 1))
```

- To control the scales on the axes of the panels, use the `scales` argument. For example, the following specifies that the range on both x- and y-axes should be -4 to 4:

```
> xyplot(y ~ x | g, scales = list(limits = c(-4, 4)))
```

Because of the complex arrangement of regions in a Trellis plot, the only way to customise the contents of the different regions, even if the Trellis plot consists of a single panel, is via panel functions. There are three important ones:

- The `panel` argument allows you to specify a function to produce graphics output within the plot region of each panel. There are default panel functions, such as `panel.xyplot`, which can be used to produce the default output. The arguments to the panel function differ depending on which trellis function is being used; for trellis function `<name>`, it is useful to look at the default panel function `panel.<name>` to find out more about what gets passed to the panel function. The coordinate system in effect for these panel functions is defined by the scales on the axes of the panels. The following example adds a horizontal line at 0 to a standard `xyplot`:

```
> xyplot(y ~ x | g, panel = function(x, y, ...) {
+     panel.xyplot(x, y, ...)
+     abline(0, 0)
+ })
```

- The `strip` argument allows you to specify what to draw in each of the strip regions. The `strip.default()` function is provided to produce the default output. For drawing in the strip region, a "normalised" coordinate system is in effect (i.e., (0, 0) at bottom-left, and (1, 1) at top-right). The following writes a left-justified label in each strip:

```
> xyplot(y ~ x | g, strip = function(which.given, which.panel,
+     var.name, factor.levels, shingle.intervals, par.strip.text,
+     strip.names, style) {
+     text(0, 0.5, paste("Variable ", which.given, ": Level ",
+         which.panel[which.given], sep = ""), adj = 0)
+ })
```

- The `prepanel` argument allows you to perform calculations to determine x- and y-scales for each panel. For example, the following sets the x- and y-ranges based on the actual ranges of the data, extended by 1 in each direction:

```
> xyplot(y ~ x | g, prepanel = function(x, y) {
+     list(xlim = range(x) + c(-1, 1), ylim = range(y) + c(-1,
+         1))
+ })
```

> **R Difference:** In R, the Trellis functions (provided by the `lattice` package) are implemented on top of the `grid` package so panel functions have to use `grid` functions or a special set of `lattice` equivalents of traditional base functions (e.g., `llines()`, `lpoints()`, ...). Another difference is that, in R, there are more coordinate systems available to the panel functions.

The appearance of graphics output in Trellis can be controlled temporarily (i.e., for the purposes of a single plot) by specifying the usual graphical parameters in-line:

```
> xyplot(y ~ x | g, pch = 16)
```

However, permanent changes to graphical parameters are *not* controlled via `par()`. Instead, Trellis maintains its own (quite large) set of graphical parameter settings. Trellis has different settings for all of the different sorts of plots that it can produce. For example, for standard scatterplots, there are lists of graphical parameters for plotting symbols and lines:

```
> trellis.par.get("plot.symbol")

$cex
[1] 0.8

$col
[1] "#000000"

$font
[1] 1

$pch
[1] 1

> trellis.par.get("plot.line")
```

```
$col
[1] "#000000"

$lty
[1] 1

$lwd
[1] 1
```

These graphical parameters are accessed via the `trellis.par.get()` and `trellis.par.set()` functions:

```
> ps <- trellis.par.get("plot.symbol")
> ps$pch = 16
> trellis.par.set("plot.symbol", ps)
```

NOTE: You should only attempt to use Trellis functions on a Trellis device (created by `trellis.device()`).

R Difference: In R, the `lattice` package provides an alternative interface for managing the appearance of Trellis plots via the `lattice.theme` and `lset` functions.