Chapter 4

Graphics

Both S-PLUS and R provide comprehensive graphics facilities for static two-dimensional plots, from simple facilities for producing common diagnostic plots by `plot(`*object*`)` to fine control over publication-quality graphs. In consequence, the number of graphics parameters is huge. In this chapter, we build up the complexity gradually. Most readers will not need the material in Section 4.4, and indeed the material there is not used elsewhere in this book. However, we *have* needed to make use of it, especially in matching existing graphical styles.

Some graphical ideas are best explored in their statistical context, so that, for example, histograms are covered in Chapter 5, survival curves in Chapter 13, biplots in Chapter 11 and time-series graphics in Chapter 14. Table 4.1 gives an overview of the high-level graphics commands with page references.

There are many books on graphical design. Cleveland (1993) discusses most of the methods of this chapter and the detailed design choices (such as the aspect ratios of plots and the presence of grids) that can affect the perception of graphical displays. As these are to some extent a matter of personal preference and this is also a guide to S, we have kept to the default choices. Spence (2001) and Wilkinson (1999) and the classics of Tufte (1983, 1990, 1997) discuss the visual exploration of data.

Trellis graphics (Becker *et al.*, 1996) are a later addition to S with a somewhat different style and philosophy to the basic plotting functions. We describe the basic functions first, then the Trellis functions in Section 4.5. R has a variant on Trellis in its `lattice` package. The Windows version of S-PLUS has a very different (and less powerful) style of object-oriented editable graphics which we do not cover. One feature we do find useful is the ability to interactively change the viewpoint in perspective plots (see page 422). The `rgl` package[1] for R under Windows provides similar facilities.

There are quite a few small differences in the R graphics model, and the description here tries to be completely accurate only for S-PLUS 6.

[1]Available at `http://www.stats.uwo.ca/faculty/murdoch/software/`.

Table 4.1: High-level plotting functions. Page references are given to the most complete description in the text. Those marked by † have alternatives in Trellis.

Function	Page	Description
`abline`	74	Add lines to the current plot in slope-intercept form.
`axis`	80	Add an axis to the plot.
`barplot` †	72	Bar graphs.
`biplot`	312	Represent rows and columns of a data matrix.
`brush spin`	75	Dynamic graphics. not R.
`contour` †	76	Contour plot. The Trellis equivalent is `contourplot` .
`dotchart` †		Produce a dot chart.
`eqscplot`	75	Plot with geometrically equal scales (our library).
`faces`		Chernoff's faces plot of multivariate data.
`frame`	78	Advance to next figure region.
`hist`	112	Histograms. We prefer our function `truehist` .
`hist2d`	130	Two-dimensional histogram calculations.
`identify locator`	80	Interact with an existing plot.
`image` † `image.legend`	76	High-density image plot functions. The Trellis version is `levelplot` .
`interaction.plot`		Interaction plot for a two-factor experiment.
`legend`	81	Add a legend to the current plot.
`matplot`	88	Multiple plots specified by the columns of a matrix.
`mtext`	81	Add text in the margins.
`pairs` †	75	All pairwise plots between multiple variables. The Trellis version is `splom` .
`par`	83	Set or ask about graphics parameters.
`persp` † `perspp` `persp.setup`	76	Three-dimensional perspective plot functions. Similar Trellis functions are called `wireframe` and `cloud`.
`pie` †		Produce a pie chart.
`plot`		Generic plotting function.
`polygon`		Add polygon(s) to the present plot, possibly filled.
`points lines`	73	Add points or lines to the current plot.
`qqplot qqnorm`	108	Quantile-quantile and normal Q-Q plots.
`rect`		(R only) Add rectangles, possibly filled.
`scatter.smooth`	230	Scatterplot with a smooth curve.
`segments arrows`	88	Draw line segments or arrows on the current plot.
`stars`		Star plots of multivariate data.
`symbols`		Draw variable-sized symbols on a plot.
`text`	73	Add text symbols to the current plot.
`title`	79	Add title(s).

Table 4.2: Some of the graphical devices available.

S-PLUS :		
	`motif`	UNIX: X11–windows systems.
	`graphsheet`	Windows, screen, printer, bitmaps.
	`win.printer`	Windows, a wrapper for a `graphsheet` .
	`postscript`	PostScript printers.
	`hplj`	UNIX: Hewlett-Packard LaserJet printers.
	`hpgl`	Hewlett-Packard HP-GL plotters.
	`pdf.graph`	Adobe's PDF format.
	`wmf.graph`	Windows metafiles.
	`java.graph`	Java device.
R :		
	`X11`	UNIX: X11–windows systems.
	`windows`	Windows, screen, printer, metafiles.
	`macintosh`	classic MacOS screen device.
	`postscript`	PostScript printers.
	`pdf`	PDF files.
	`xfig`	files for `XFig` .
	`png`	PNG bitmap graphics.
	`jpeg`	JPEG bitmap graphics.
	`bitmap`	several bitmap formats *via* GhostScript.

4.1 Graphics Devices

Before any plotting commands can be used, a graphics device must be opened to receive graphical output. Most commonly this is a window on the screen of a workstation or a printer. A list of supported devices on the current hardware with some indication of their capabilities is available from the on-line help system by `?Devices` . (Note the capital letter.)

A graphics device is opened by giving the command in Table 4.2, possibly with parameters giving the size and position of the window; for example, using S-PLUS on UNIX,

```
motif("-geometry 600x400-0+0")
```

opens a small graphics window initially positioned in the top right-hand corner of the screen. All current S environments will automatically open a graphics device if one is needed, but we often choose to open the device ourselves and so take advantage of the ability to customize it.

To make a device request permission before each new plot to clear the screen use either `par(ask = T)` (which affects just the current device) or `dev.ask(ask = T)` (not R: applies to every device). R

All open graphics devices may be closed using `graphics.off()` ; quitting the S session does this automatically.

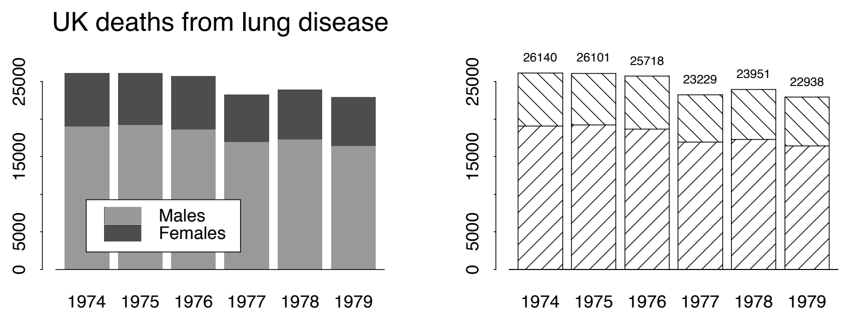


Figure 4.1: Two different styles of bar chart showing the annual UK deaths from certain lung diseases. In each case the lower block is for males, the upper block for females.

It is possible to have several graphical devices open at once. By default the most recently opened one is used, but `dev.set` can be used to change the current device (by number). The function `dev.list` lists currently active devices, and `dev.off` closes the current device, or one specified by number. There are also commands `dev.cur`, `dev.next` and `dev.prev` which return the number of the current, next or previous device on the list. The `dev.copy` function copies the current plot to the specified device (by default the next device on the list).

Note that for some devices little or no output will appear on a file until `dev.off` or `graphics.off` is called.

Many of the graphics devices on windowing systems have menus of choices, for example, to make hardcopies and to alter the colour scheme in use. The S-PLUS `motif` device has a `Copy` option on its `Graph` menu that allows a (smaller) copy of the current plot to be copied to a new window, perhaps for comparison with later plots. (The copy window can be dismissed by the Delete item on its Graph menu.)

There are some special considerations for users of `graphsheet` devices on S-PLUS for Windows: see page 451.

4.2 Basic Plotting Functions

The function `plot` is a generic function that, when applied to many types of S objects, will give one or more plots. Many of the plots appropriate to univariate data such as boxplots and histograms are considered in Chapter 5.

Bar charts

The function to display barcharts is `barplot`. This has many options (described in the on-line help), but some simple uses are shown in Figure 4.1. (Many of the details are covered in Section 4.3.)

```
# R: data(mdeaths); data(fdeaths); library(ts)
lung.deaths <- aggregate(ts.union(mdeaths, fdeaths), 1)
```

```
barplot(t(lung.deaths), names = dimnames(lung.deaths)[[1]],
        main = "UK deaths from lung disease")
legend(locator(1), c("Males", "Females"), fill = c(2, 3))
loc <- barplot(t(lung.deaths), names = dimnames(lung.deaths)[[1]],
               angle = c(45, 135), density = 10, col = 1)
total <- rowSums(lung.deaths)
text(loc, total + par("cxy")[2], total, cex = 0.7) #R: xpd = T
```

Line and scatterplots

The default plot function takes arguments `x` and `y`, vectors of the same length, or a matrix with two columns, or a list (or data frame) with components `x` and `y` and produces a simple scatterplot. The axes, scales, titles and plotting symbols are all chosen automatically, but can be overridden with additional graphical parameters that can be included as named arguments in the call. The most commonly used ones are:

`type = "c"`	Type of plot desired. Values for `c` are: `p` for points only (the default), `l` for lines only, `b` for both points and lines (the lines miss the points), `s, S` for step functions (`s` specifies the level of the step at the left end, `S` at the right end), `o` for overlaid points and lines, `h` for high-density vertical line plotting, and `n` for no plotting (but axes are still found and set).
`axes = L`	If `F` all axes are suppressed (default `T`, axes are automatically constructed).

`xlab = "string"`
`ylab = "string"`
	Give labels for the `x`- and/or `y`-axes (default: the names, including suffices, of the `x`- and `y`-coordinate vectors).

`sub = "string"`
`main = "string"`
	`sub` specifies a title to appear under the `x`-axis label and `main` a title for the top of the plot in larger letters (default: both empty).

`xlim = c(lo ,hi)`
`ylim = c(lo, hi)`
	Approximate minimum and maximum values for `x`- and/or `y`-axis settings. These values are normally automatically rounded to make them 'pretty' for axis labelling.

The functions `points`, `lines`, `text` and `abline` can be used to add to a plot, possibly one created with `type = "n"`. Brief summaries are:

`points(x,y,...)`	Add points to an existing plot (possibly using a different plotting character). The plotting character is set by `pch=` and the size of the character by `cex=` or `mkh=`.
`lines(x,y,...)`	Add lines to an existing plot. The line type is set by `lty=` and width by `lwd=`. The `type` options may be used.
`text(x,y,labels,...)`	Add text to a plot at points given by `x,y`. `labels` is an integer or character vector; `labels[i]` is plotted at point `(x[i],y[i])`. The default is `seq(along=x)`.

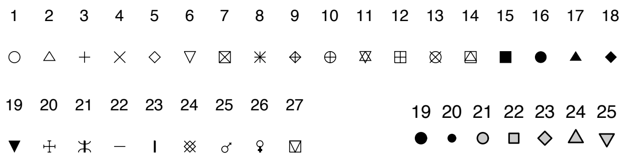


Figure 4.2: Plotting symbols or marks, specified by `pch = n`. Those on the left of the second row are only available on `graphsheet` devices, and those on the right of the second row only in R (where the fill colour for 21 to 25 has been taken as light grey).

`abline(a, b, ...)`
`abline(h = c, ...)`
`abline(v = c, ...)`
`abline(`*lmobject*`, ...)`
	Draw a line in intercept and slope form, (a,b), across an existing plot. `h = c` may be used to specify y-coordinates for the heights of horizontal lines to go across a plot, and `v = c` similarly for the x-coordinates for vertical lines. The coefficients of a suitable *lmobject* are used.

These are the most commonly used graphics functions; we have shown examples of their use in Chapter 1, and show many more later. (There are also functions `arrows` and `symbols` that we do not use in this book.) The plotting characters available for `plot` and `points` can be characters of the form `pch = "o"` or numbered from 0 to 27, which uses the marks shown in Figure 4.2.

Size of text and symbols

Confusingly, the size of plotting characters is selected in one of two very different ways. For plotting characters (by `pch = "o"`) or text (by `text`), the parameter `cex` (for 'character expansion') is used. This defaults to the global setting (which S+ defaults to 1), and rescales the character by that factor. In S-PLUS for a mark set by `pch = n`, the size is controlled by the `mkh` parameter which gives the height of the symbol *in inches*. (This will be clear for printers; for screen devices the default device region is about 8 in $\times$ 6 in and this is not changed by resizing the window.) However, if `mkh = 0` (the default, and always in R) the size is then controlled by `cex` and the default size of each symbol is approximately that of `O`. Care is needed in changing `cex` on a call to `plot`, as this may[2] also change the size of the axis labels. It is better to use, for example,

```
plot(x, y, type = "n")                        # axes only
points(x, y, pch = 4, mkh = 0, cex = 0.7)  # add the points
```

If `cex` is used to change the size of all the text on a plot, it will normally be desirable to set `mex` to the same value to change the interline spacing. An alternative to specifying `cex` is `csi`, which specifies the absolute size of a character (in inches). (There is no `msi`.)

The default text size can be changed for some graphics devices, for example, by argument `pointsize` for the `postscript`, `win.printer`, `windows` and `macintosh` devices.

[2]In R these are controlled by `cex.axis`; there are also `cex.main`, `cex.sub` and `cex.lab`, the last for the axis titles.

Equally scaled plots

There are many plots, for example, in multivariate analysis, that represent distances in the plane and for which it is essential to have a scaling of the axes that is geometrically accurate. This can be done in many ways, but most easily by our function `eqscplot` which behaves as the default plot function but shrinks the scale on one axis until geometrical accuracy is attained.

Warning: when screen devices (except a `graphsheet`) are resized the S-PLUS process is not informed, so `eqscplot` can only work for the original window shape.

R has an argument `asp` that can be given to many high-level plotting functions and fixes scales so that x units are `asp` times as large as y units, even across window resizing. R

Multivariate plots

The plots we have seen so far deal with one or two variables. To view more we have several possibilities. A *scatterplot matrix* or *pairs* plot shows a matrix of scatterplots for each pair of variables, as we saw in Figure 1.2, which was produced by `splom(~ hills)`. Enhanced versions of such plots are a *forte* of Trellis graphics, so we do not discuss how to make them in the base graphics system.

Dynamic graphics

S-PLUS has limited facilities for dynamic plots; R has none. Both can work with XGobi and GGobi (see page 302) to add dynamic brushing, selecting and rotating.

The S-PLUS function `brush` allows interaction with the (lower half) of a scatterplot matrix. An example is shown in Figure 1.3 on page 9. As it is much easier to understand these by using them, we suggest you try

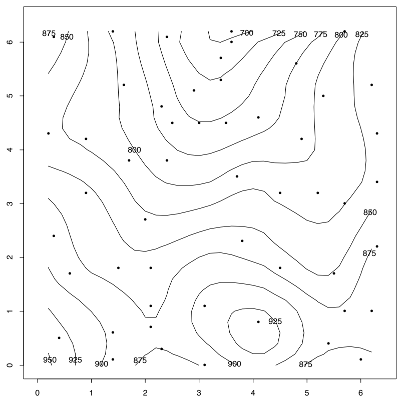
```
brush(hills)
```

and experiment.

Points can be highlighted (marked with a symbol) by moving the brush (a rectangular window) over them with button 1 held down. When a point is highlighted, it is shown highlighted in all the displays. Highlighting is removed by brushing with button 2 held down. It is also possible to add or remove points by clicking with button 1 in the scrolling list of row names.

One of four possible (device-dependent) marking symbols can be selected by clicking button 1 on the appropriate one in the display box on the right. The marking is by default persistent, but this can be changed to 'transient' in which only points under the brush are labelled (and button 1 is held down). It is also possible to select marking by row label as well as symbol.

The brush size can be altered under UNIX by picking up a corner of the brush in the `brush size` box with the mouse button 1 and dragging to the required size. Under Windows, move the brush to the background of the main `brush` window, hold down the left mouse button and drag the brush to the required size.


(a) by `contour`

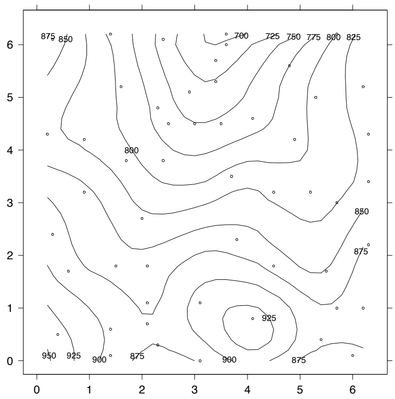

(b) by `contourplot`

Figure 4.3: Contour plots of `loess` smoothing of the `topo` dataset. Note the differences in the axes and the way points are depicted.

The plot produced by `brush` will also show a three-dimensional plot (unless `spin = F`), and this can be produced on its own by `spin`. Under UNIX clicking with mouse button 1 will select three of the variables for the x-, y- and z-axes. The plot can be spun in several directions and resized by clicking in the appropriate box. The `speed` box contains a vertical line or slider indicating the current setting.

Plots from `brush` and `spin` can only be terminated by clicking with the mouse button 1 on the `quit` box or button.

Obviously `brush` and `spin` are available only on suitable screen devices, including `motif` and `graphsheet`. Hardcopy is possible only by directly printing the window used, not by copying the plot to a printer graphics device.

Plots of surfaces

The functions `contour`, `persp` and `image` allow the display of a function defined on a two-dimensional regular grid. Their Trellis equivalents give more elegant output, so we do not discuss them in detail. The function `contour` allows more control than `contourplot`.[3] We anticipate an example from Chapter 15 of plotting a smooth topographic surface for Figure 4.3, and contrast it with `contourplot`.

```
# R: library(modreg)
topo.loess <- loess(z ~ x * y, topo, degree = 2, span = 0.25)
topo.mar <- list(x = seq(0, 6.5, 0.2), y = seq(0, 6.5, 0.2))
topo.lo <- predict(topo.loess, expand.grid(topo.mar))
par(pty = "s")       # square plot
contour(topo.mar$x, topo.mar$y, topo.lo, xlab = "", ylab = "",
   levels = seq(700,1000,25), cex = 0.7)
```

[3]At the time of writing `contourplot` was not available in the R package `lattice`.

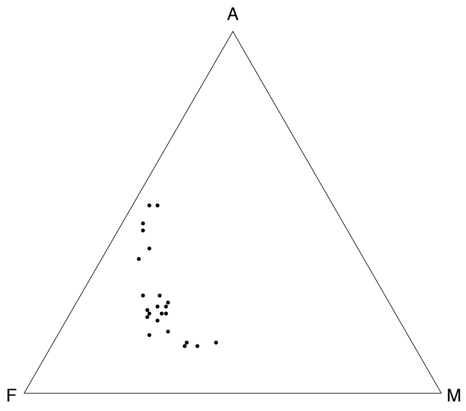


Figure 4.4: A ternary plot of the compositions of 23 rocks from Aitchison (1986).

```
points(topo$x, topo$y)
par(pty =  "m")

contourplot(z ~ x * y, mat2tr(topo.lo), aspect = 1,
   at = seq(700, 1000, 25), xlab = "", ylab = "",
   panel = function(x, y, subscripts, ...) {
      panel.contourplot(x, y, subscripts, ...)
      panel.xyplot(topo$x,topo$y, cex = 0.5)
   }
)
```

This generates values of the surface on a regular 33×33 grid generated by `expand.grid`. Our `MASS` library provides the functions `con2tr` and `mat2tr` to convert objects designed for input to `contour` and matrices as produced by `predict.loess` into data frames suitable for the Trellis 3D plotting routines.

The S-PLUS for Windows GUI and R under Windows have ways to visualize such surfaces interactively; see pages 69 and 422.

Making new types of plots

The basic components described so far can be used to create new types of plot as the need arises.

Ternary plots are used for compositional data (Aitchison, 1986) where there are three components whose proportions add to one. These are represented by a point in an equilateral triangle, where the distances to the sides add to a constant. These are implemented in a function `ternary` which is given on the help page of the `MASS` dataset `Skye`; see Figure 4.4.

4.3 Enhancing Plots

In this section we cover a number of ways that are commonly used to enhance plots, without reaching the level of detail of Section 4.4.

Some plots (such as Figure 1.1) are square, whereas others are rectangular. The shape is selected by the graphics parameter `pty`. Setting `par(pty = "s")` selects a square plotting region, whereas `par(pty = "m")` selects a maximally sized (and therefore usually non-square) region.

Multiple figures on one plot

We have already seen several examples of plotting two or more figures on a single device surface, apart from scatterplot matrices. The graphics parameters `mfrow` and `mfcol` subdivide the plotting region into an array of figure regions. They differ in the order in which the regions are filled. Thus

```
par(mfrow = c(2, 3))
par(mfcol = c(2, 3))
```

both select a 2×3 array of figures, but with the first they are filled along rows, and with the second down columns. A new figure region is selected for each new plot, and figure regions can be skipped by using `frame`.

All but two[4] of the multi-figure plots in this book were produced with `mfrow`. Most of the side-by-side plots were produced with `par(mfrow = c(2, 2))`, but using only the first two figure regions.

The `split.screen` function provides an alternative and more flexible way of generating multiple displays on a graphics device. An initial call such as

```
split.screen(figs = c(3, 2))
```

subdivides the current device surface into a 3×2 array of *screens*. The screens created in this example are numbered 1 to 6, *by rows*, and the original device surface is known as screen 0. The current screen is then screen 1 in the upper left corner, and plotting output will fill the screen as it would a figure. Unlike multi-figure displays, the next plot will use the same screen unless another is specified using the `screen` function. For example, the command

```
screen(3)
```

causes screen 3 to become the next current screen.

On screen devices the function `prompt.screen` may be used to define a screen layout interactively. The command

```
split.screen(prompt.screen())
```

allows the user to define a screen layout by clicking mouse button 1 on diagonally opposite corners. In our experience this requires a steady hand, although there is a `delta` argument to `prompt.screen` that can be used to help in aligning screen edges. Alternatively, if the `figs` argument to `split.screen` is specified as an $N \times 4$ matrix, this divides the plot into N screens (possibly overlapping) whose corners are specified by giving (xl, xu, yl, yl) as the row of the matrix (where the whole region is $(0, 1, 0, 1)$).

[4]Figures 6.2 and 6.3, where the `fig` parameter was used.

The `split.screen` function may be used to subdivide the current screen recursively, thus leading to irregular arrangements. In this case the screen numbering sequence continues from where it had reached.

Split-screen mode is terminated by a call to `close.screen(all = T)`; individual screens can be shut by `close.screen(n)`.

The function `subplot`[5] provides a third way to subdivide the device surface. This has call `subplot(fun, ...)` which adds the graphics output of `fun` to an existing plot. The size and position can be determined in many ways (see the on-line help); if all but the first argument is missing a call to `locator` is used to ask the user to click on any two opposite corners of the plot region.[6]

Use of the `fig` parameter to `par` provides an even more flexible way to subdivide a plot; see Section 4.4 and Figure 6.2 on page 153.

With multiple figures it is normally necessary to reduce the size of the text. If either the number of rows or columns set by `mfrow` or `mfcol` is three or more, the text size is halved by setting `cex = 0.5` (and `mex = 0.5`; see Section 4.4). This may produce characters that are too small and some resetting may be appropriate. (On the other hand, for a 2×2 layout the characters will usually be too large.) For all other methods of subdividing the plot surface the user will have to make an appropriate adjustment to `cex` and `mex` or to the default text size (for example by changing `pointsize` on the `postscript` and other devices).

Adding information

The basic plots produced by `plot` often need additional information added to give context, particularly if they are not going to be used with a caption. We have already seen the use of `xlab`, `ylab`, `main` and `sub` with scatterplots. These arguments can all be used with the function `title` to add titles to existing plots. The first argument is `main`, so

```
title("A Useful Plot?")
```

adds a main title to the current plot.

Further points and lines are added by the `points` and `lines` functions. We have seen how plot symbols can be selected with `pch=`. The line type is selected by `lty=`. This is device-specific, but usually includes solid lines (1) and a variety of dotted, dashed and dash-dot lines. Line width is selected by `lwd=`, with standard width being 1, and the effect being device-dependent.

Using colour

The colour model of S-PLUS graphics is quite complex. Colours are referred to as numbers, and set by the parameter `col`. Sometimes only one colour is allowed (e.g., `points`) and sometimes `col` can be a vector giving a colour for each plot item (e.g., `text`). There will always be at least two colours, 0 (the background, useful for erasing by over-plotting) and 1. However, how many colours there are and what they appear as is set by the device. Furthermore, there are separate

[5]Not in R, which has another approach called `layout`.

[6]Not the figure region; Figure 4.5 on page 81 shows the distinction.

colour groups, and what they are is device-specific. For example, motif devices have separate colour spaces for lines (including symbols), text, polygons (including histograms, bar charts and pie charts) and images, and graphsheet devices have two spaces, one for lines and text, the other for polygons and images. Thus the colours can appear completely differently when a graph is copied from device to device, in particular on screen and on a hardcopy. It is usually a good idea to design a colour scheme for each device.

It is necessary to read the device help page thoroughly (and for postscript, also that for ps.options.send).

R has a different and more coherent colour model involving named colours, user-settable palettes and even transparency. See the help topics palette and colors for more details.

Identifying points interactively

The function identify has a similar calling sequence to text. The first two arguments give the x- and y-coordinates of points on a plot and the third argument gives a vector of labels for each point. (The first two arguments may be replaced by a single list argument with two of its components named x and y, or by a two-column matrix.) The labels may be a character string vector or a numeric vector (which is coerced to character). Then clicking with mouse button 1 near a point on the plot causes its label to be plotted; labelling all points or clicking anywhere in the plot with button 2 terminates the process.[7] (The precise position of the click determines the label position, in particular to left or right of the point.) We saw an example in Figure 1.4 on page 10. The function returns a vector of index numbers of the points that were labelled.

In Chapter 1 we used the locator function to add new points to a plot. This function is most often used in the form locator(1) to return the (x, y) coordinates of a single button click to place a label or legend, but can also be used to return the coordinates of a series of points, terminated by clicking with mouse button 2.

Adding further axes and grids

It is sometimes useful to add further axis scales to a plot, as in Figure 8.1 on page 212 which has scales for both kilograms and pounds. This is done by the function axis. There we used

```
attach(wtloss)
oldpar <- par() # R: oldpar <- par(no.readonly = TRUE)
# alter margin 4; others are default
par(mar = c(5.1, 4.1, 4.1, 4.1))
plot(Days, Weight, type = "p", ylab = "Weight (kg)")
Wt.lbs <- pretty(range(Weight*2.205))
axis(side = 4, at = Wt.lbs/2.205, lab = Wt.lbs, srt = 90)
mtext("Weight (lb)", side = 4, line = 3)
detach()
```

[7]With R on a Macintosh (which only has one mouse button) click outside the plot window to terminate locator or identify.

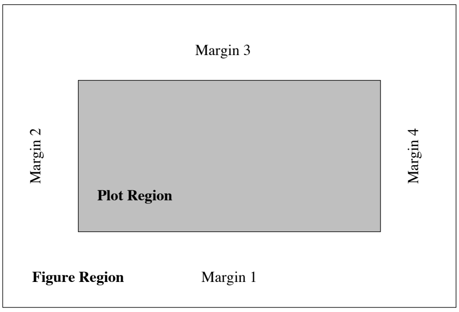


Figure 4.5: Anatomy of a graphics figure.

```
    par(oldpar)
```

This adds an axis on side 4 (labelled clockwise from the bottom; see Figure 4.5) with labels rotated by $90°$ (`srt = 90`, not needed in R where unlike S-PLUS the rotation is controlled by the setting of `las`) and then uses `mtext` to add a label 'underneath' that axis. Other parameters are explained in Section 4.4. Please read the on-line documentation very carefully to determine which graphics parameters are used in which circumstances. R

Grids can be added by using `axis` with long tick marks, setting parameter `tck = 1` (yes, obviously). For example, a dotted grid is created by

```
    axis(1, tck = 1, lty = 2); axis(2, tck = 1, lty = 2)
```

and the location of the grid lines can be specified using `at=`.

Adding legends

Legends are added by the function `legend`. Since it can label many types of variation such as line type and width, plot symbol, colour, and fill type, its description is very complex. All calls are of the form

```
    legend(x, y, legend, ...)
```

where `x` and `y` give either the upper left corner of the legend box or both upper left and lower right corners. These are often most conveniently specified on-screen by using `locator(1)` or `locator(2)`. Argument `legend` is a character vector giving the labels for each variation. Most of the remaining arguments are vectors of the same length as `legend` giving the appropriate coding for each variation, by `lty=`, `lwd=`, `col=`, `fill=`, `angle=` and `density=`. Argument `pch` is a single character string concatenating the symbols; for numerical `pch` in S-PLUS use the vector argument `marks`.

By default the legend is contained in a box; the drawing of this box can be suppressed by argument `bty = "n"`.

The Trellis function `key` provides a more flexible approach to constructing legends, and can be used with basic plots. (See page 104 for further details.)

Non-English labels

Non-native English speakers will often want to include characters from their other languages in labels. For Western European languages written in ISO-latin1 encoding this will normally work; it does for all the R devices and for `motif` and `graphsheet` devices in S-PLUS. To use such characters with the `postscript` device under S-PLUS, set

```
ps.options(setfont = ps.setfont.latin1)
```

If you are unable to enter the characters from the keyboard, octal escapes of the form `"\341"` (which encodes á) can be used.

R's `postscript` device allows arbitrary encodings via its `encoding` parameter, and S-PLUS's `ps.setfont.latin1` could be modified to use a different encoding such as ISO-latin2.

Mathematics in labels

Users frequently wish to include the odd subscript, superscript and mathematical symbol in labels. There is no general solution, but for the S-PLUS `postscript` driver[8] Alan Zaslavsky's package `postscriptfonts` adds these features. We can label Figure 7.3 (on page 209) by λ (from font 13, the PostScript symbol font).

```
library(postscriptfonts)
x <- 0:100
plik <- function(lambda)
  sum(dpois(x, lambda) * 2 * ( (lambda - x) +
       x * log(pmax(1, x)/lambda)))
lambda <- c(1e-8, 0.05, seq(0.1, 5, 0.1))
plot(lambda, sapply(lambda, plik), type = "l",
     ylim = c(0, 1.4), ylab = "", xlab = "")
abline(h = 1, lty = 3)
mixed.mtext(texts = "l", side = 1, line = 3, font = 13) # xlab
mixed.mtext(texts = "E~f13~d~.l~f1~.(deviance)", adj = 0.5,
            side = 2, line = 3, font = 13)             # ylab
```

R has rather general facilities to label with mathematics: see `?plotmath` and Murrell and Ihaka (2000). Here we could use (on most devices, including on-screen)

```
plot(lambda, sapply(lambda, plik), type = "l", ylim = c(0, 1.4),
     xlab = expression(lambda),
     ylab = expression(paste(E[lambda], "(deviance)")))
```

4.4 Fine Control of Graphics

The graphics process is controlled by *graphics parameters*, which are set for each graphics device. Each time a new device is opened these parameters for that

[8] Under UNIX or Windows.

device are reset to their default values. Graphics parameters may be set, or their current values queried, using the `par` function. If the arguments to `par` are of the `name = value` form the graphics parameter `name` is set to `value`, if possible, and other graphics parameters may be reset to ensure consistency. The value returned is a list giving the previous parameter settings. Instead of supplying the arguments as `name = value` pairs, `par` may also be given a single list argument with named components.

If the arguments to `par` are quoted character strings, `"name"`, the current value of graphics parameter `name` is returned. If more than one quoted string is supplied the value is a list of the requested parameter values, with named components. The call `par()` with no arguments returns a list of all the graphics parameters.

Some of the many graphics parameters are given in Tables 4.3 and 4.4 (on pages 84 and 87). Those in Table 4.4 can also be supplied as arguments to high-level plot functions, when they apply just to the figure produced by that call. (The layout parameters are ignored by the high-level plot functions.)

The figure region and layout parameters

When a device is opened it makes available a rectangular surface, the *device region*, on which one or more plots may appear. Each plot occupies a rectangular section of the device surface called a *figure*. A figure consists of a rectangular *plot region* surrounded by a *margin* on each side. The margins or sides are numbered one to four, clockwise starting from the bottom. The plot region and margins together make up the *figure region*, as in Figure 4.5 on page 81. The device surface, figure region and plot region have their vertical sides parallel and hence their horizontal sides also parallel.

The size and position of figure and plot regions on a device surface are controlled by *layout parameters*, most of which are listed in Table 4.3. Lengths may be set in either absolute or relative units. Absolute lengths are in *inches*, whereas relative lengths are in *text lines* (so relative to the current font size).

Margin sizes are set using `mar` for text lines or `mai` for inches. These are four-component vectors giving the sizes of the lower, left, upper and right margins in the appropriate units. Changing one causes a consistent change in the other; changing `mex` will change `mai` but not `mar`.

Positions may be specified in relative units using the unit square as a coordinate system for which some enclosing region, such as the device surface or the figure region, is the unit square. The `fig` parameter is a vector of length four specifying the current figure as a fraction of the device surface. The first two components give the lower and upper x-limits and the second two give the y-limits. Thus to put a point plot in the left-hand side of the display and a Q-Q plot on the right-hand side we could use:

```
postscript(file = "twoplot.ps")    # open a postscript device
par(fig = c(0, 2/3, 0, 1))         # set a figure on the left
plot(x, y)                         # point plot
par(fig = c(2/3, 1, 0, 1))         # set a figure on the right
```

Table 4.3: Some graphics layout parameters with example settings.

`din, fin, pin`	Absolute device size, figure size and plot region size in inches. `fin = c(6, 4)`
`fig`	Define the figure region as a fraction of the device region. `fig = c(0, 0.5, 0,1)`
`font`	Small positive integer determining a text font for characters and hence an interline spacing. For S-PLUS's `postscript` device one of the standard PostScript fonts given by `ps.options("fonts")`. In R font 1 is plain, font 2 italic, font 3 bold, font 4 bold italic and font 5 is the symbol font. `font = 3`
`mai, mar`	The four margin sizes, in inches (`mai`), or in text line units (`mar`, that is, *relative* to the current font size). Note that `mar` need not be an integer. `mar = c(3, 3, 1, 1) + 0.1`
`mex`	Number of text lines per interline spacing. `mex = 0.7`
`mfg`	Define a position within a specified multi-figure display. `mfg = c(2, 2, 3, 2)`
`mfrow, mfcol`	Define a multi-figure display. `mfrow = c(2, 2)`
`new`	Logical value indicating whether the current figure has been used. `new = T`
`oma, omi, omd`	Define outer margins in text lines or inches, or by defining the size of the array of figures as a fraction of the device region. `oma = c(0, 0, 4, 0)`
`plt`	Define the plot region as a fraction of the figure region. `plt = c(0.1, 0.9, 0.1, 0.9)`
`pty`	Plot type, or shape of plotting region, `"s"` or `"m"`
`uin`	(not R) Return inches per user coordinate for x and y.
`usr`	Limits for the plot region in user coordinates. `usr = c(0.5, 1.5, 0.75, 10.25)`

```
qqnorm(resid(obj))                 # diagnostic plot
dev.off()
```

The left-hand figure occupies $2/3$ of the device surface and the right-hand figure $1/3$. For regular arrays of figures it is simpler to use `mfrow` or `split.screen`.

Positions in the plot region may also be specified in absolute *user coordinates*. Initially user coordinates and relative coordinates coincide, but any high-level plotting function changes the user coordinates so that the x- and y-coordinates range from their minimum to maximum values as given by the plot axes. The graphics parameter `usr` is a vector of length four giving the lower and upper x- and y-limits for the user coordinate system. Initially its setting is `usr = c(0,1,0,1)`. Consider another simple example:

```
> motif()                  # open a device
> par("usr")               # usr coordinates
[1] 0 1 0 1
```

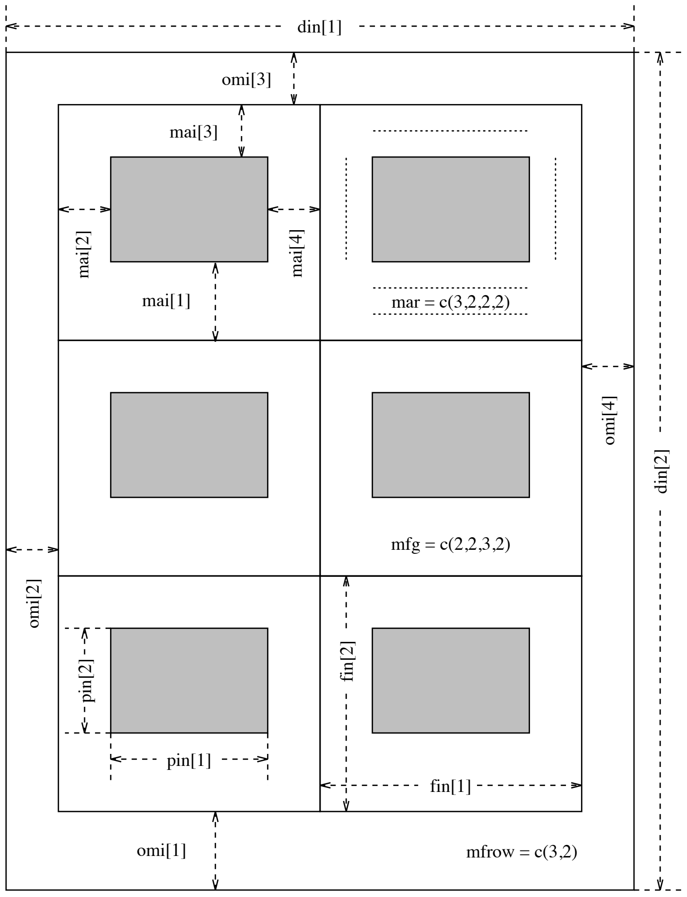


Figure 4.6: An outline of a 3×2 multi-figure display with outer margins showing some graphics parameters. The current figure is at position $(2, 2)$ and the display is being filled by rows. In this figure "`fin[1]`" is used as a shorthand for `par("fin")[1]`, and so on.

```
> x <- 1:20
> y <- x + rnorm(x)        # generate some data
> plot(x, y)               # produce a scatterplot
> par("usr")               # user coordinates now match the plot
[1]  0.2400 20.7600  1.2146 21.9235
```

Any attempt to plot outside the user coordinate limits causes a warning message unless the general graphics parameter `xpd` is set to `T`.

Figure 4.6 shows some of the layout parameters for a multi-figure layout. Such an array of figures may occupy the entire device surface, or it may have *outer margins*, which are useful for annotations that refer to the entire array. Outer margins are set with the parameter `oma` (in text lines) or `omi` (in inches). Alternatively `omd` may be used to set the region containing the array of figures in a similar way to which `fig` is used to set one figure. This implicitly determines the outer margins as the complementary region. In contrast to what happens with the margin parameters `mar` and `mai`, a change to `mex` will leave the outer margin size, `omi`, constant but adjust the number of text lines, `oma`.

Text may be put in the outer margins by using `mtext` with parameter `outer = T`.

Common axes for figures

There are at least two ways to ensure that several plots share a common axis or axes.

1. Use the same `xlim` or `ylim` (or both) setting on each plot and ensure that the parameters governing the way axes are formed, such as `lab`, `las`, `xaxs` and allies, do not change.
2. Set up the desired axis system with the first plot and then use `par` to set the low-level parameter `xaxs = "d"`, `yaxs = "d"` or both as appropriate. This ensures that the axis or axes are not changed by further high-level plot commands on the same device.

An example: A Q-Q normal plot with envelope

In Chapter 5 we recommend assessing distributional form by quantile-quantile plots. A simple way to do this is to plot the sorted values against quantile approximations to the expected normal order statistics and draw a line through the 25 and 75 percentiles to guide the eye, performed for the variable `Infant.Mortality` of the Swiss provinces data (on fertility and socio-economic factors on Swiss provinces in about 1888) by

```
## in R just use data(swiss)
swiss <- data.frame(Fertility = swiss.fertility, swiss.x)
attach(swiss)
qqnorm(Infant.Mortality)
qqline(Infant.Mortality)
```

The reader should check the result and compare it with the style of Figure 4.7.

Another suggestion to assess departures is to compare the sample Q-Q plot with the envelope obtained from a number of other Q-Q plots from generated normal samples. This is discussed in (Atkinson, 1985, §4.2) and is based on an idea of Ripley (see Ripley, 1981, Chapter 8). The idea is simple. We generate a number of other samples of the same size from a normal distribution and scale all samples to mean 0 and variance 1 to remove dependence on location and scale parameters. Each sample is then sorted. For each order statistic the maximum and minimum values for the generated samples form the upper and lower envelopes. The envelopes are plotted on the Q-Q plot of the scaled original sample and form a guide to what constitutes serious deviations from the expected behaviour under normality. Following Atkinson our calculation uses 19 generated normal samples.

We begin by calculating the envelope and the x-points for the Q-Q plot.

Table 4.4: Some of the more commonly used general and high-level graphics parameters with example settings.

Text:	
`adj`	Text justification. 0 = left justify, 1 = right justify, 0.5 = centre.
`cex`	Character expansion. `cex = 2`
`csi`	Height of font (inches). `csi = 0.11`
`font`	Font number: device-dependent.
`srt`	String rotation in degrees. `srt = 90`
`cin cxy`	Character width and height in inches and `usr` coordinates (for information, not settable).
Symbols:	
`col`	Colour for symbol, line or region. `col = 2`
`lty`	Line type: solid, dashed, dotted, etc. `lty = 2`
`lwd`	Line width, usually as a multiple of default width. `lwd = 2`
`mkh`	Mark height (inches). Ignored in R. `mkh = 0.05`
`pch`	Plotting character or mark. `pch = "*"` or `pch = 4` for marks. (See page 74.)
Axes:	
`bty`	Box type, as `"o"`, `"l"`, `"7"`, `"c"`, `"n"`.
`exp`	(not R) Notation for exponential labels. `exp = 1`
`lab`	Tick marks and labels. `lab = c(3, 7, 4)`
`las`	Label orientation. 0 = parallel to axis, 1 = horizontal, 2 = vertical.
`log`	Control log axis scales. `log = "y"`
`mgp`	Axis location. `mgp = c(3, 1, 0)`
`tck`	Tick mark length as signed fraction of the plot region dimension. `tck = -0.01`
`xaxp yaxp`	Tick mark limits and frequency. `xaxp = c(2, 10, 4)`
`xaxs yaxs`	Style of axis limits. `xaxs = "i"`
`xaxt yaxt`	Axis type. `"n"` (null), `"s"` (standard), `"t"` (time) or `"l"` (log).
High Level:	
`ann`	(R only) Should titles and axis labels be plotted?
`ask`	Prompt before going on to next plot? `ask = F`
`axes`	Print axes? `axes = F`
`main`	Main title. `main = "Figure 1"`
`sub`	Subtitle. `sub = "23-Jun-2002"`
`type`	Type of plot. `type = "n"`
`xlab ylab`	Axis labels. `ylab = "Speed in km/sec"`
`xlim ylim`	Axis limits. `xlim = c(0, 25)`
`xpd`	May points or lines go outside the plot region? `xpd = T`

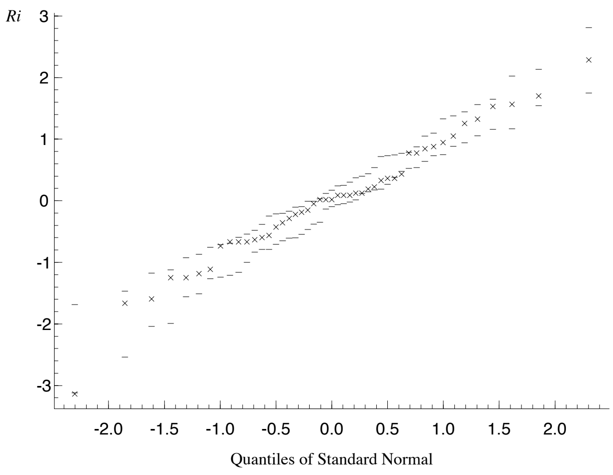


Figure 4.7: The Swiss fertility data. A Q-Q normal plot with envelope for infant mortality.

```
samp <- cbind(Infant.Mortality, matrix(rnorm(47*19), 47, 19))
samp <- apply(scale(samp), 2, sort)
rs <- samp[, 1]
xs <- qqnorm(rs, plot = F)$x
env <- t(apply(samp[, -1], 1, range))
```

As an exercise in building a plot with specific requirements we now present the envelope and Q-Q plot in a style very similar to Atkinson's. To ensure that the Q-Q plot has a y-axis large enough to take the envelope we could calculate the y-limits as before, or alternatively use a matrix plot with `type = "n"` for the envelope at this stage. The axes and their labels are also suppressed for the present:

```
matplot(xs, cbind(rs, env), type = "pnn",
        pch = 4, mkh = 0.06, axes = F, xlab = "", ylab = "")
```

The argument setting `type = "pnn"` specifies that the first column (`rs`) is to produce a point plot and the remaining two (`env`) no plot at all, but the axes will allow for them. Setting `pch = 4` specifies a 'cross' style plotting symbol (see Figure 4.2) similar to Atkinson's, and `mkh = 0.06` establishes a suitable size for the plotting symbol.

Atkinson uses small horizontal bars to represent the envelope. We can now calculate a half length for these bars so that they do not overlap and do not extend beyond the plot region. Then we can add the envelope bars using `segments`:

```
xyul <- par("usr")
smidge <- min(diff(c(xyul[1], xs, xyul[2])))/2
segments(xs - smidge, env[, 1], xs + smidge, env[, 1])
segments(xs - smidge, env[, 2], xs + smidge, env[, 2])
```

Atkinson's axis style differs from the default S style in several ways. There are many more tick intervals; the ticks are inside the plot region rather than outside;

there are more labelled ticks; and the labelled ticks are longer than the unlabelled. From experience ticks along the x-axis at 0.1 intervals with labelled ticks at 0.5 intervals seems about right, but this is usually too close on the y-axis. The axes require four calls to the `axis` function:

```
xul <- trunc(10*xyul[1:2])/10
axis(1, at=seq(xul[1], xul[2], by = 0.1), labels = F, tck = 0.01)
xi <- trunc(xyul[1:2])
axis(1, at = seq(xi[1], xi[2], by = 0.5), tck = 0.02)
yul <- trunc(5*xyul[3:4])/5
axis(2, at = seq(yul[1], yul[2], by = 0.2), labels = F, tck= 0.01)
yi <- trunc(xyul[3:4])
axis(2, at = yi[1]:yi[2], tck = 0.02)
```

Finally we add the L-box, put the x-axis title at the centre and the y-axis title at the top:

```
box(bty = "l")           # lower case "L"
# S: ps.options()$fonts
mtext("Quantiles of Standard Normal", side=1, line=2.5, font=3)
# S: mtext("Ri", side = 2, line = 2, at = yul[2], font = 10)
# R: mtext(expression(R[i]), side = 2, line = 2, at = yul[2])
```

where in S-PLUS fonts 3 and 10 are Times-Roman and Times-Italic on the device used (`postscript` under UNIX), found from the list given by `ps.options()`.

The final plot is shown in Figure 4.7 on page 88.

4.5 Trellis Graphics

Trellis graphics were developed to provide a consistent graphical 'style' and to extend conditioning plots; the style is a development of that used in Cleveland (1993).

Trellis is very prescriptive, and changing the display style is not always an easy matter.

It may be helpful to understand that Trellis is written entirely in the S language, as calls to the basic plotting routines. Two consequences are that it can be slow and memory-intensive, and that it takes over many of the graphics parameters for its own purposes. (Global settings of graphics parameters are usually not used, the outer margin parameters `omi` being a notable exception.) Computation of a Trellis plot is done in two passes: once when a Trellis object is produced, and once when that object is printed (producing the actual plot).

The `trellis` library contains a large number of examples: use

```
?trellis.examples
```

to obtain an up-to-date list. These are all functions that can be called to plot the example, and listed to see how the effect was achieved.

R has a similar system in its package `lattice`; however, that is built on a different underlying graphics model called `grid` and mixes (even) less well with traditional S graphics. This runs most of the examples shown here, but the output will not be identical.

Trellis graphical devices

The `trellis.device` graphical device is provided by the `trellis` library. It is perhaps more accurate to call it a meta-device, for it uses one of the underlying graphical devices,[9] but customizes the parameters of the device to use the Trellis style, and in particular its colour schemes.

Trellis graphics is intended to be used on a `trellis.device` device, and may give incorrect results on other devices.

Trellis devices by default use colour for screen windows and greylevels for printer devices. The settings for a particular device can be seen by running the command `show.settings()`. These settings are not the same for all colour screens, nor for all printer devices. Trellis colour schemes have a mid-grey background on colour screens (but not colour printers). If a Trellis plot is used without a graphics device already in use, a suitable Trellis device is started.

Trellis model formulae

Trellis graphics functions make use of the language for model formulae described in Section 3.7. The Trellis code for handling model formulae to produce a data matrix from a data frame (specified by the `data` argument) allows the argument `subset` to select a subset of the rows of the data frame, as one of the first three forms of indexing vector described on page 27. (Character vector indices are not allowed.)

There are a number of inconsistencies in the use of the formula language. There is no `na.action` argument, and missing values are handled inconsistently; generally rows with `NA`s are omitted, but `splom` fails if there are missing values. Surprisingly, `splom` uses a formula, but does not accept a `data` argument.

Trellis uses an extension to the model formula language, the operator '`|`' which can be read as 'given'. Thus if `a` is a factor, `lhs ~ rhs | a` will produce a plot for each level of `a` of the subset of the data for which `a` has that level (so estimating the conditional distribution given `a`). Conditioning on two or more factors gives a plot for each combination of the factors, and is specified by an interaction, for example, `| a*b`. For the extension of conditioning to continuous variates via what are known as *shingles*, see page 101.

Trellis plot objects can be kept, and `update` can be used to change them, for example, to add a title or change the axis labels, before re-plotting by printing (often automatically as the result of the call to `update`).

Basic Trellis plots

As Table 4.5 shows, the basic styles of plot exist in Trellis, but with different names and different default styles. Their usage is best seen by considering how to produce some figures in the Trellis style.

Figure 1.2 (page 9) was produced by `splom(~ hills)`. Trellis plots of scatterplot matrices read from bottom to top (as do all multi-panel Trellis displays,

[9]Currently `motif`, `postscript`, `graphsheet`, `win.printer`, `pdf.graph`, `wmf.graph` and `java.graph` where these are available.

Table 4.5: Trellis plotting functions. Page references are given to the most complete description in the text.

Function	Page	Description
`xyplot`	94	Scatterplots.
`bwplot`	92	Boxplots.
`stripplot`	98	Display univariate data against a numerical variable.
`dotplot`		ditto in another style,
`histogram`		'Histogram', actually a frequency plot.
`densityplot`		Kernel density estimates.
`barchart`		Horizontal bar charts.
`piechart`		Pie chart.
`splom`	90	Scatterplot matrices.
`contourplot`	76	Contour plot of a surface on a regular grid.
`levelplot`	94	Pseudo-colour plot of a surface on a regular grid.
`wireframe`	94	Perspective plot of a surface evaluated on a regular grid.
`cloud`	104	A perspective plot of a cloud of points.
`key`	104	Add a legend.
`color.key`	94	Add a color key (as used by `levelplot`).
`trellis.par.get`	93	Save Trellis parameters.
`trellis.par.set`	93	Reset Trellis parameters.
`equal.count`	102	Compute a shingle.

like graphs rather than matrices, despite the meaning of the name `splom`). By default the panels in a `splom` plot are square.

Figure 4.8 is a Trellis version of Figure 1.4. Note that the y-axis numbering is horizontal by default (equivalent to the option `par(las = 1)`), and that points are plotted by open circles rather than filled circles or stars. It is not possible to add to a Trellis plot,[10] so the Trellis call has to include all the desired elements. This is done by writing a *panel function*, in this case

```
# R: library(lqs)
xyplot(time ~ dist, data = hills,
   panel = function(x, y, ...) {
      panel.xyplot(x, y, ...)
      panel.lmline(x, y, type = "l")
      panel.abline(lqs(y ~ x), lty = 3)
      identify(x, y, row.names(hills))
   }
```

[10] As the user coordinate system is not retained; but the plot call can be updated by `update` and re-plotted.

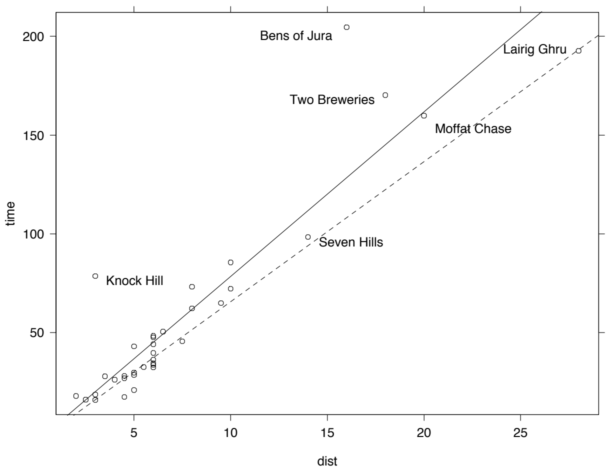


Figure 4.8: A Trellis version of Figure 1.4 (page 10).

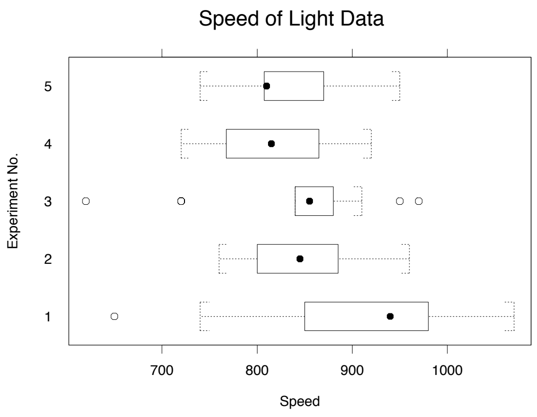


Figure 4.9: A Trellis version of Figure 1.5 (page 11).

```
)
```

Figure 4.9 is a Trellis version of Figure 1.5. Boxplots are known as box-and-whisker plots, and are displayed horizontally. This figure was produced by

```
bwplot(Expt ~ Speed, data = michelson, ylab = "Experiment No.")
title("Speed of Light Data")
```

Note the counter-intuitive way the formula is used. This plot corresponds to a one-way layout splitting `Speed` by experiment, so it is tempting to use `Speed` as the response. It may help to remember that the formula is of the `y ~ x` form for the x- and y-axes of the plot. (The same ordering is used for all the univariate plot functions.)

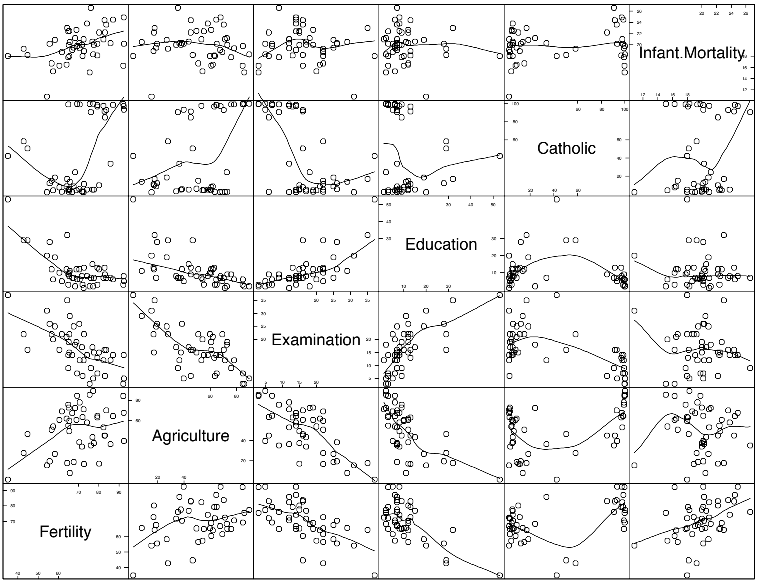


Figure 4.10: A Trellis scatterplot matrix display of the Swiss provinces data.

Figure 4.10 is an enhanced scatterplot matrix, again using a panel function to add to the basic display. Now we see the power of panel functions, as the basic plot commands can easily be applied to multi-panel displays. The `aspect = "fill"` command allows the array of plots to fill the space; by default the panels are square as in Figure 1.2.

```
splom(~ swiss, aspect = "fill",
   panel = function(x, y, ...) {
      panel.xyplot(x, y, ...); panel.loess(x, y, ...)
   }
)
```

Most Trellis graphics functions have a `groups` parameter, which we can illustrate on the `stormer` data used in Section 8.4 (see Figure 4.11).

```
sps <- trellis.par.get("superpose.symbol")
sps$pch <- 1:7
trellis.par.set("superpose.symbol", sps)
xyplot(Time ~ Viscosity, data = stormer, groups = Wt,
    panel = panel.superpose, type = "b",
    key = list(columns = 3,
        text = list(paste(c("Weight:   ", "", ""),
                          unique(stormer$Wt), "gms")),
        points = Rows(sps, 1:3)
        )
)
```

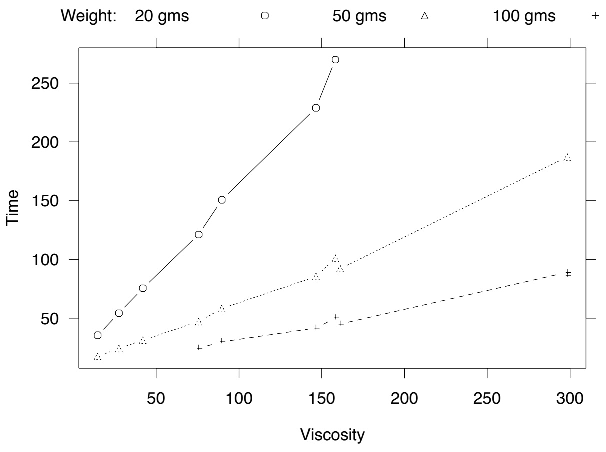


Figure 4.11: A Trellis plot of the `stormer` data.

Here we have changed the default plotting symbols (which differ by device) to the first seven `pch` characters shown in Figure 4.2 on page 74. (We could just use the argument `pch = 1:7` to `xyplot`, but then specifying the key becomes much more complicated.)

Figure 4.12 shows further Trellis plots of the smooth surface shown in Figure 4.3. Once again panel functions are needed to add the points. The `aspect = 1` parameter ensures a square plot. The `drape = T` parameter to `wireframe` is optional, producing the superimposed greylevel (or pseudo-colour) plot.

```
topo.plt <- expand.grid(topo.mar)
topo.plt$pred <- as.vector(predict(topo.loess, topo.plt))
levelplot(pred ~ x * y, topo.plt, aspect = 1,
   at = seq(690, 960, 10), xlab = "", ylab = "",
   panel = function(x, y, subscripts, ...) {
      panel.levelplot(x, y, subscripts, ...)
      panel.xyplot(topo$x,topo$y, cex = 0.5, col = 1)
   }
)
wireframe(pred ~ x * y, topo.plt, aspect = c(1, 0.5),
   drape = T, screen = list(z = -150, x = -60),
   colorkey = list(space="right", height=0.6))
```

(The arguments given by `colorkey` refer to the `color.key` function.) There is no simple way to add the points to the perspective display.

Trellises of plots

In multivariate analysis we necessarily look at several variables at once, and we explore here several ways to do so. We can produce a scatterplot matrix of the first three principal components of the `crabs` data (see page 302) by

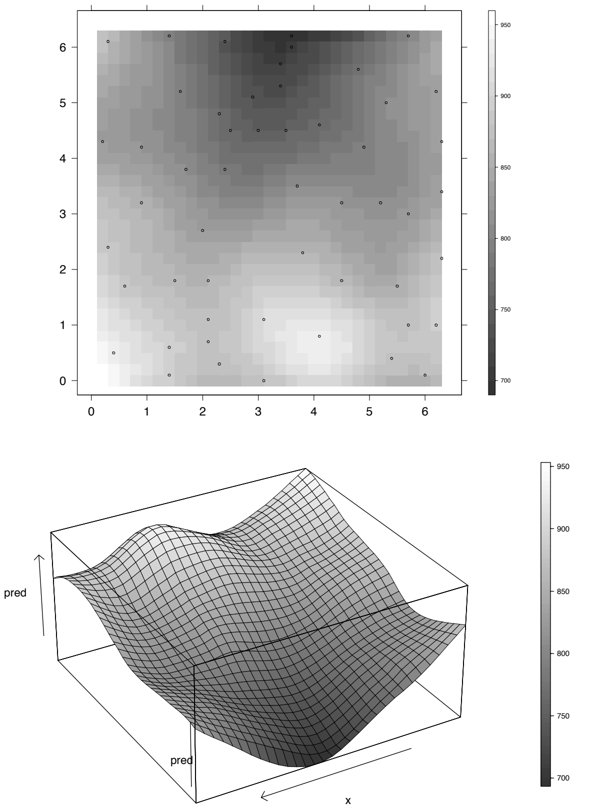


Figure 4.12: Trellis `levelplot` and `wireframe` plots of a `loess` smoothing of the `topo` dataset.

```
lcrabs.pc <- predict(princomp(log(crabs[,4:8])))
crabs.grp <- c("B", "b", "O", "o")[rep(1:4, each = 50)]
splom(~ lcrabs.pc[, 1:3], groups = crabs.grp,
    panel = panel.superpose,
    key = list(text = list(c("Blue male", "Blue female",
                             "Orange Male", "Orange female")),
        points = Rows(trellis.par.get("superpose.symbol"), 1:4),
        columns = 4)
   )
```

A 'black and white' version of this plot is shown in Figure 4.13. On a 'colour' device the groups are distinguished by colour and are all plotted with the same symbol (`o`).

However, it might be clearer to display these results as a trellis of `splom` plots, by

```
sex <- crabs$sex; levels(sex) <- c("Female", "Male")
sp <- crabs$sp; levels(sp) <- c("Blue", "Orange")
```

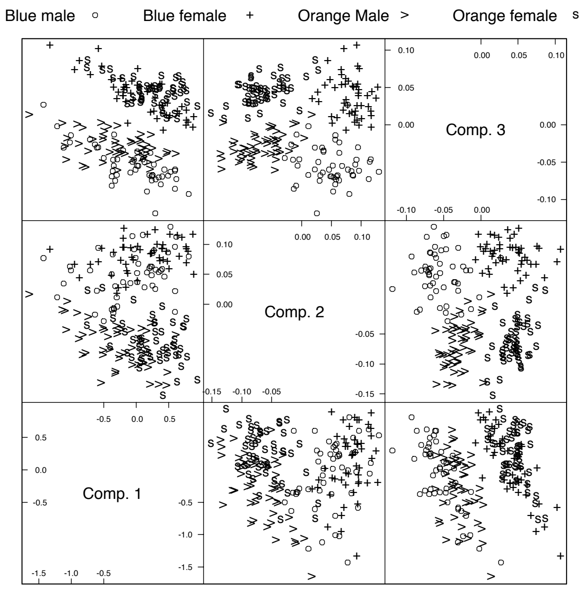


Figure 4.13: A scatterplot matrix of the first three principal components of the `crabs` data.

```
splom(~ lcrabs.pc[, 1:3] | sp*sex, cex = 0.5, pscales = 0)
```

as shown in Figure 4.14. Notice how this is the easiest method to code. It is at the core of the paradigm of Trellis, which is to display many plots of subsets of the data in some meaningful layout.

Now consider data from a multi-factor study, Quine's data on school absences discussed in Sections 6.6 and 7.4. It will help to set up more informative factor labels, as the factor names are not given (by default) in trellises of plots.

```
Quine <- quine
levels(Quine$Eth) <- c("Aboriginal", "Non-aboriginal")
levels(Quine$Sex) <- c("Female", "Male")
levels(Quine$Age) <- c("primary", "first form",
                       "second form", "third form")
levels(Quine$Lrn) <- c("Average learner", "Slow learner")
bwplot(Age ~ Days | Sex*Lrn*Eth, data = Quine)
```

This gives an array of eight boxplots, which by default takes up two pages. On a screen device there will be no pause between the pages unless the argument `ask = T` is set for `par`. It is more convenient to see all the panels on one page, which we can do by asking for a different layout (Figure 4.15). We also suppress the colouring of the strip labels by using `style = 1`; there are currently six preset styles.

```
bwplot(Age ~ Days | Sex*Lrn*Eth, data = Quine, layout = c(4, 2),
       strip = function(...) strip.default(..., style = 1))
```

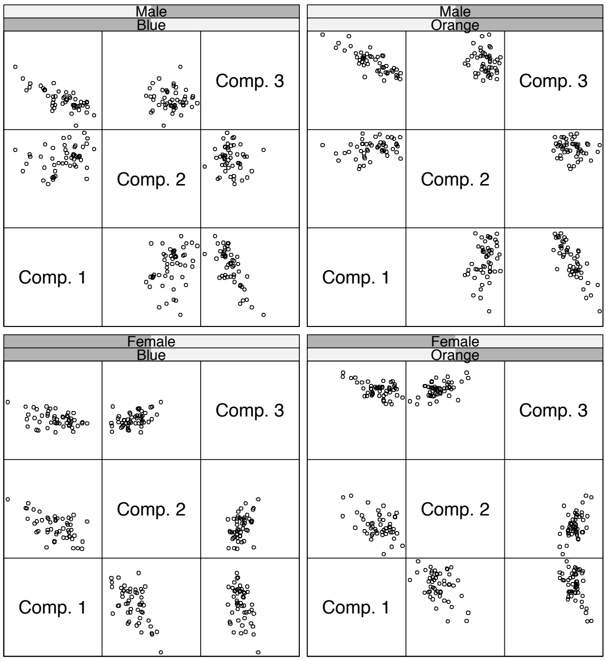


Figure 4.14: A multi-panel version of Figure 4.13.

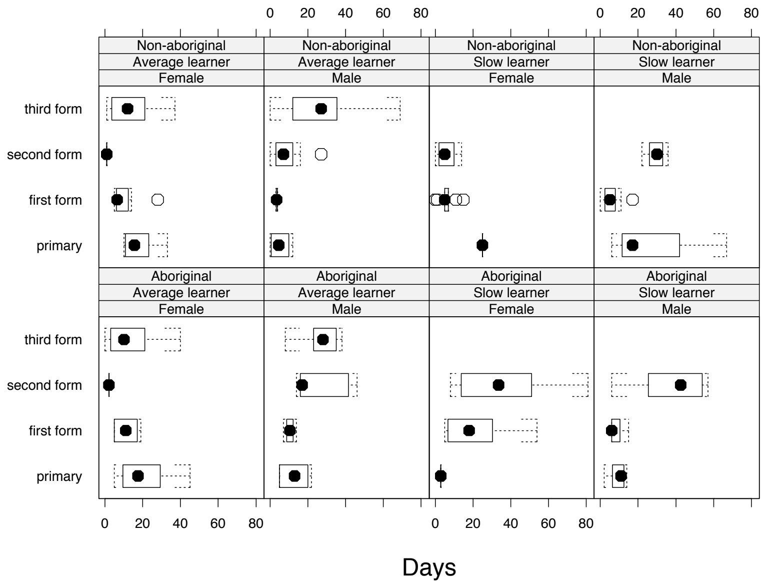


Figure 4.15: A multi-panel boxplot of Quine's school attendance data.

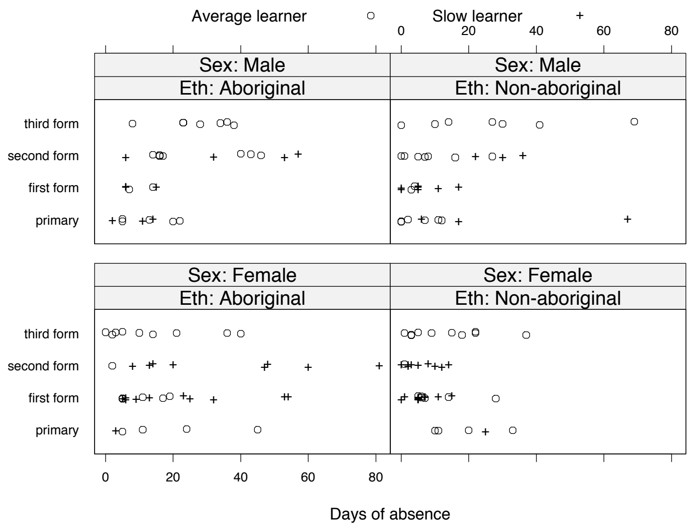


Figure 4.16: A `stripplot` of Quine's school attendance data.

A `stripplot` allows us to look at the actual data. We jitter the points slightly to avoid overplotting.

```
stripplot(Age ~ Days | Sex*Lrn*Eth, data = Quine,
          jitter = T, layout = c(4, 2))

stripplot(Age ~ Days | Eth*Sex, data = Quine,
    groups = Lrn, jitter = T,
    panel = function(x, y, subscripts, jitter.data = F, ...) {
        if(jitter.data)  y <- jitter(y)
        panel.superpose(x, y, subscripts, ...)
    },
    xlab = "Days of absence",
    between = list(y = 1), par.strip.text = list(cex = 0.7),
    key = list(columns = 2, text = list(levels(Quine$Lrn)),
        points = Rows(trellis.par.get("superpose.symbol"), 1:2)
        ),
    strip = function(...)
         strip.default(..., strip.names = c(T, T), style = 1)
)
```

The second form of plot, shown in Figure 4.16, uses different symbols to distinguish one of the factors. We include the factor name in the strip labels, using a custom `strip` function.

The Trellis function `dotplot` is very similar to `stripplot`; its panel function includes horizontal lines at each level. Function `stripplot` uses the styles of `xyplot` whereas `dotplot` has its own set of defaults; for example, the default plotting symbol is a filled rather than open circle.

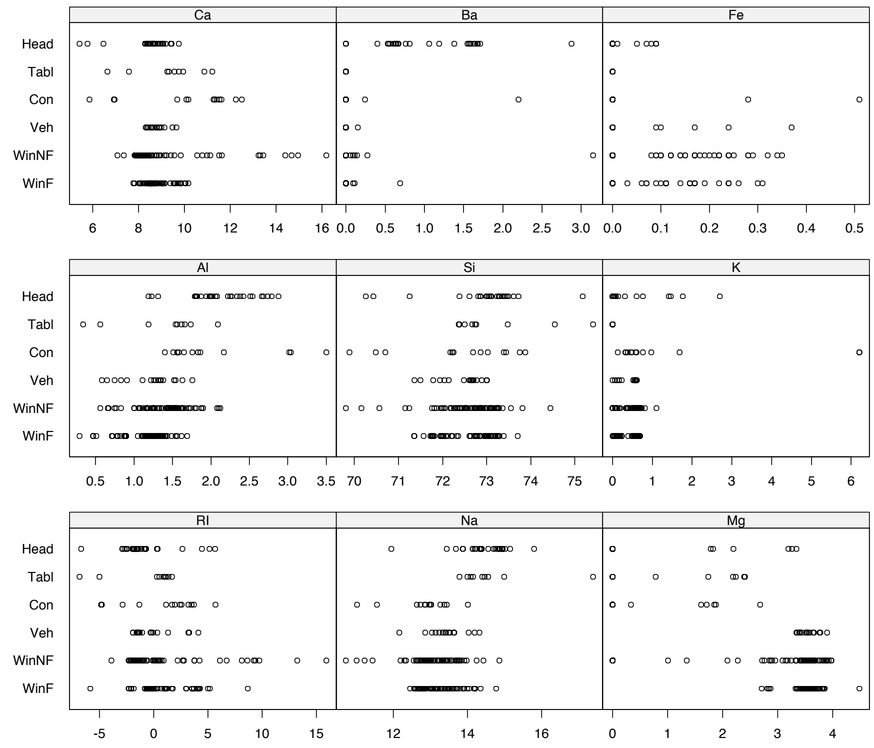


Figure 4.17: Plot by `stripplot` of the forensic glass dataset `fgl`.

As a third example, consider our dataset `fgl`. This has 10 measurements on 214 fragments of glass from forensic testing, the measurements being of the refractive index and composition (percent weight of oxides of Na, Mg, Al, Si, K, Ca, Ba and Fe). The fragments have been classified by six sources. We can look at the types for each measurement by

```
fgl0 <- fgl[ , -10] # omit type.
fgl.df <- data.frame(type = rep(fgl$type, 9),
   y = as.vector(as.matrix(fgl0)),
   meas = factor(rep(1:9, each = 214), labels = names(fgl0)))
stripplot(type ~ y | meas, data = fgl.df,
   scales = list(x = "free"), xlab = "", cex = 0.5,
   strip = function(...) strip.default(style = 1, ...))
```

Layout of a trellis

A trellis of plots is generated as a sequence of plots that are then arranged in rows, columns and pages. The sequence is determined by the order in which the conditioning factors are given: the first varying fastest. The order of the levels of the factor is that of its `levels` attribute.

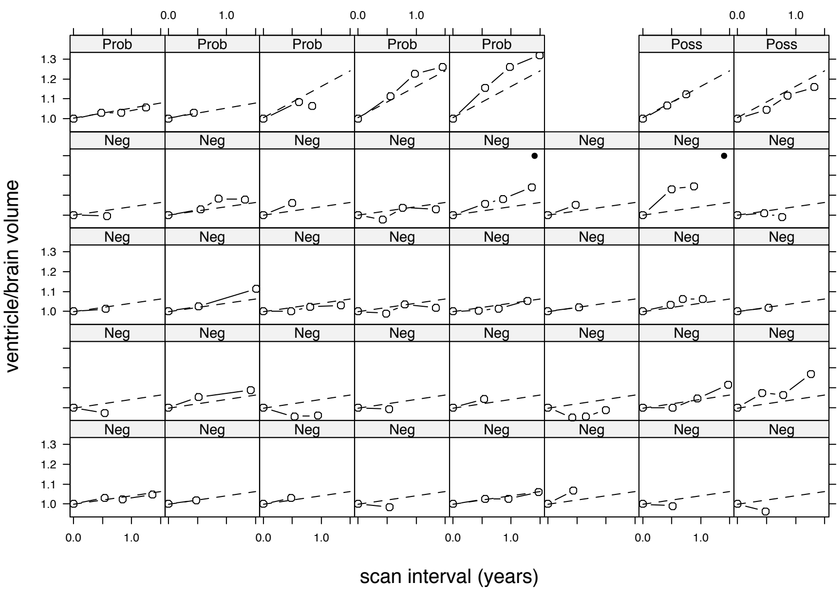


Figure 4.18: The presentation of results from a study of 39 subjects. In a real application this could be larger and so less dominated by the labels. Colour could be used to distringuish groups, too.

How the sequence of plots is displayed on the page(s) is controlled by an algorithm that tries to optimize the use of the space available, but it can be controlled by the `layout` parameters. A specification `layout = c(`c, r, p`)` asks for c columns, r rows and p pages. (Note the unusual ordering.) Using $c = 0$ allows the algorithm to choose the number of columns; p is used to produce only the first p pages of a many-page trellis.

If the number of levels of a factor is large and not easily divisible (for example, seven), we may find a better layout by leaving some of the cells of the trellis empty using the `skip` argument. Figure 4.18 shows another use of `layout` and `skip`.

The `between` parameter can be used to specify gaps in the trellis layout, as in Figure 4.16. It is a list with `x` and `y` components, numeric vectors that specify the gaps in units of character height. The `page` parameter can be used to invoke a function (with argument n, the page number) to label each page. The default page function does nothing.

Subscripts and groups

The `subscripts` argument of the panel function is supplied if the Trellis function is called with argument `subscripts = T`.[11] Then its value is a numeric vector of indices of cases (normally rows of `data`) that have been passed to that panel.

[11] And it seems sometimes even if it is not.

Figure 4.18 shows the use of Trellis to present the results of a real study. There were 39 subjects in three groups (marked on the strips), each being brain-scanned 2–4 times over up to 18 months. The plot shows the data and for each patient a dashed line showing the mean rate of change for the alloted group. Two patients whose panels are marked with a dot were later shown to have been incorrectly allocated to the 'normals' group.

Note how we arrange the layout to separate the groups. We make use of the `subscripts` argument to the panel function to identify the subject; vector `pr3` holds a set of predictions at the origin and after 1.5 years from a linear mixed-effects model.

```
xyplot(ratio ~ scant | subject, data = A5,
       xlab = "scan interval (years)",
       ylab = "ventricle/brain volume",
       subscripts = T, ID = A5$ID,
       strip = function(factor, ...)
          strip.default(..., factor.levels = labs, style = 1),
       layout = c(8, 5, 1),
       skip = c(rep(F, 37), rep(T, 1), rep(F, 1)),
       panel = function(x, y, subscripts, ID) {
           panel.xyplot(x, y, type = "b", cex = 0.5)
           which <- unique(ID[subscripts])
           panel.xyplot(c(0, 1.5), pr3[names(pr3) == which],
                        type = "l", lty = 3)
           if(which == 303 || which == 341) points(1.4, 1.3)
       })
```

Note how other arguments, here `ID`, are passed to the panel function as additional arguments. One special extra argument is `groups` which is interpreted by `panel.superpose`, as in Figure 4.11.

Conditioning plots and shingles

The idea of a trellis of plots conditioning on combinations of one or more factors can be extended to conditioning on real-valued variables, in what are known as conditioning plots or *coplots*. Two variables are plotted against each other in a series of plots with the values of further variable(s) restricted to a series of possibly overlapping ranges. This needs an extension of the concept of a factor known as a *shingle*.[12]

Suppose we wished to examine the relationship between `Fertility` and `Education` in the Swiss fertility data as the variable `Catholic` ranges from predominantly non-Catholic to mainly Catholic provinces. We add a smooth fit to each panel (and `span` controls the smoothness: see page 423).

```
Cath <- equal.count(swiss$Catholic, number = 6, overlap = 0.25)
xyplot(Fertility ~ Education | Cath, data = swiss,
   span = 1, layout = c(6, 1), aspect = 1,
   panel = function(x, y, span) {
```

[12]In American usage this is a rectangular wooden tile laid partially overlapping on roofs or walls.

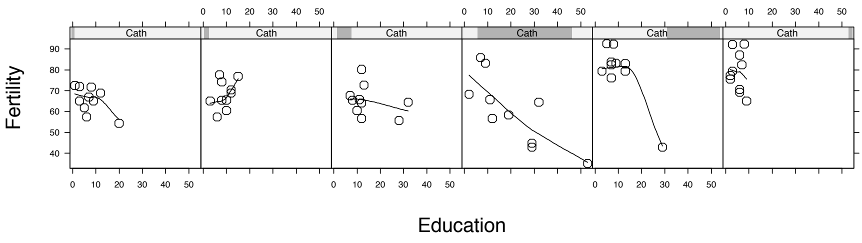


Figure 4.19: A conditioning plot for the Swiss provinces data.

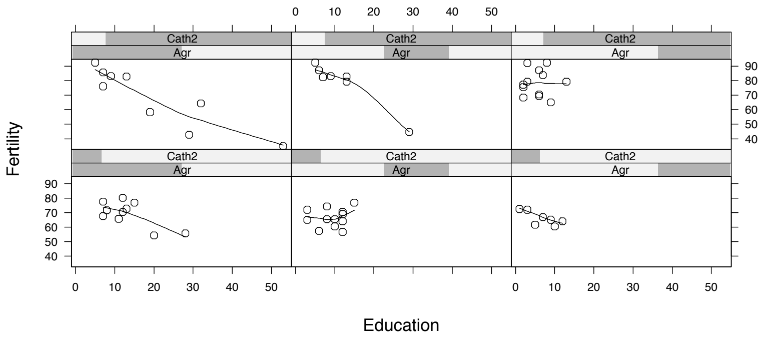


Figure 4.20: Another conditioning plot with two conditioning shingles. The upper row shows the predominantly Catholic provinces.

```
         panel.xyplot(x, y); panel.loess(x, y, span)
      }
   )
```

The result is shown in Figure 4.19, with the strips continuing to show the (now overlapping) coverage for each panel. Fertility generally falls as education rises and rises as the proportion of Catholics in the population rises. Note that the level of education is lower in predominantly Catholic provinces.

The function `equal.count` is used to construct a shingle with suitable ranges for the conditioning intervals.

Conditioning plots may also have more than one conditioning variable. Let us condition on Catholic and agriculture simultaneously. Since the dataset is small it seems prudent to limit the number of panels to six in all.

```
   Cath2 <- equal.count(swiss$Catholic, number = 2, overlap = 0)
   Agr <- equal.count(swiss$Agric, number = 3, overlap = 0.25)
   xyplot(Fertility ~ Education | Agr * Cath2, data = swiss,
      span = 1, aspect = "xy",
      panel = function(x, y, span) {
         panel.xyplot(x, y); panel.loess(x, y, span)
      }
   )
```

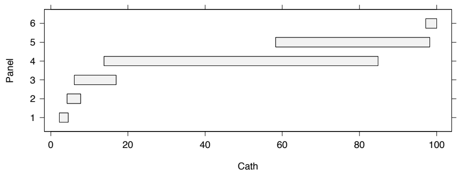


Figure 4.21: A plot of the shingle `Cath`.

The result is shown in Figure 4.20. In general, the fertility rises with the proportion of Catholics and agriculture and falls with education. There is no convincing evidence of substantial interaction.

Shingles have levels, and can be printed and plotted:

```
> Cath
Data:
 [1] 10.0  84.8  93.4  33.8   5.2  90.6  92.9  97.2  97.7  91.4
   ....
Intervals:
  min   max count
  2.2   4.5    10
  4.2   7.7    10
   ....
Overlap between adjacent intervals:
[1] 3 2 3 2 3
> levels(Cath)
  min   max
  2.2   4.5
   ....
> plot(Cath, aspect = 0.3)
```

Multiple displays per page

Recall from page 89 that a Trellis object is plotted by printing it. The method `print.trellis` has optional arguments `position`, `split` and `more`. The argument `more` should be set to `T` for all but the last part of a figure. The position of individual parts on the device surface can be set by either `split` or `position`. A `split` argument is of the form `c(`x, y, nx, ny`)` for four integers. The second pair gives a division into a $nx \times ny$ layout, just like the `mfrow` and `mfcol` arguments to `par`. The first pair gives the rectangle to be used within that layout, with origin at the bottom left.

A `position` argument is of the form `c(`*xmin, ymin, xmax, ymax*`)` giving the corners of the rectangle within which to plot the object. (This is a different order from `split.screen`.) The coordinate system for this rectangle is $[0, 1]$ for both axes, but the limits can be chosen outside this range.

The `print.trellis` works by manipulating the graphics parameter `omi`, so the outer margin settings are preserved. However, none of the basic methods (page 78) of subdividing the device surface will work, and if a trellis print fails omi is not reset. (Using par(omi = rep(0, 4), new = F) will reset the usual defaults.)

Fine control

Detailed control of Trellis plots may be accomplished by a series of arguments described in the help page for trellis.args, with variants for the wireframe and cloud perspective plots under trellis.3d.args.

We have seen some of the uses of panel functions. Some care is needed with computations inside panel functions that use any data (or user-defined objects or functions) other than their arguments. First, the computations will occur inside a deeply nested set of function calls, so care is needed to ensure that the data are visible, often best done by passing the data as extra arguments. Second, those computations will be done at the time the result is printed (that is, plotted) and so the data need to be in the desired state at plot time, not just when the trellis object is created.

If non-default panel functions are used, we may want these to help control the coordinate system of the plots, for example, to use a fitted curve to decide the aspect ratio of the panels. This is the purpose of the prepanel argument, and there are prepanel functions corresponding to the densityplot, lmline, loess, qq, qqmath and qqmathline panel functions. These will ensure that the whole of the fitted curve is visible, and they may affect the choice of aspect ratio.

The parameter aspect controls the aspect ratio of the panels. A numerical value (most usefully one) sets the ratio, "fill" adjusts the aspect ratio to fill the space available and "xy" attempts to bank the fitted curves to $\pm 45°$. (See Figure 4.20.)

The scales argument determines how the x and y axes are drawn. It is a list of components of name = value form, and components x and y may themselves be lists. The default relation = "same" ensures that the axes on each panel are identical. With relation = "sliced" the same numbers of data units are used, but the origin may vary by panel, whereas with relation = "free" the axes are drawn to accommodate just the data for that panel. One can also specify most of the parameters of the axis function, and also log = T to obtain a $\log_{10}$ scale or even log = 2 for a $\log_2$ scale.

The function splom has an argument varnames which sets the names of the variables plotted on the diagonal. The argument pscales determines how the axes are plotted; set pscales = 0 to omit them.

Keys

The function key is a replacement for legend, and can also be used as an argument to Trellis functions. If used in this way, the Trellis routines allocate space for the key, and repeat it on each page if the trellis extends to multiple pages.

The call of key specifies the location of the key by the arguments x, y and corner. By default corner = c(0, 1), when the coordinate (x, y) specifies the upper left corner of the key. Any other coordinate of the key can be specified by setting `corner`, but the size of the key is computed from its contents. (If the argument `plot = F`, the function returns a two-element vector of the computed width and height, which can be used to allocate space.) When `key` is used as an argument to a Trellis function, the position is normally specified not by `x` and `y` but by the argument `space` which defaults to `"top"`.

Most of the remaining arguments to `key` will specify the contents of the key. The (optional) arguments `points`, `lines`, `text` and `rectangles` (for `barchart`) will each specify a column of the key in the order in which they appear. Each argument must be a *list* giving the graphics parameters to be used (and for `text`, the first argument must be the character vector to be plotted). (The function `trellis.par.get` is useful to retrieve the actual settings used for graphics parameters.)

The third group of arguments to `key` fine-tunes its appearance—should it be transparent (`transparent = T`), the presence of a border (specified by giving the border colour as argument `border`), the spacing between columns (`between.columns` in units of character width), the background colour, the font(s) used, the existence of a title and so on. Consult the on-line help for the current details. The argument `columns` specifies the number of columns in the key—we used this in Figures 4.11 and 4.13.

Perspective plots

The argument `aspect` is a vector of two values for the perspective plots, giving the ratio of the y and z sizes to the x size; its effect can be seen in Figure 4.12.

The arguments `distance`, `perspective` and `screen` control the perspective view used. If `perspective = T` (the default), the `distance` argument (default 0.2) controls the extent of the perspective, although not on a physical distance scale as 1 corresponds to viewing from infinity. The `screen` argument (default `list(z = 40, x = -60)`) is a list giving the rotations (in degrees) to be applied to the specified axis in turn. The initial coordinate system has x pointing right, z up and y into the page.

The argument `zoom` (default 1) may be used to scale the final plot, and the argument `par.box` controls how the lines forming the enclosing box are plotted.