5

Scatter Plots and Extensions

The scatter plot is possibly the single most important statistical graphic. In this chapter we discuss the xyplot() function, which can be used to produce several variants of scatter plots, and splom(), which produces scatter-plot matrices. We also include a brief discussion of parallel coordinates plots, as produced by parallel(), which are related to scatter-plot matrices in terms of the kinds of data they are used to visualize, although not so much in the actual visual encoding.

A *scatter plot* graphs two variables directly against each other in a Cartesian coordinate system. It is a simple graphic in the sense that the data are directly encoded without being summarized in any way; often the aspects that the user needs to worry about most are graphical ones such as whether to join the points by a line, what colors to use, and so on. Depending on the purpose, scatter plots can also be enhanced in several ways. In this chapter, we go over some of the variants supported by panel.xyplot(), which is the default panel function for both xyplot() and splom() (under the alias panel.splom()).

5.1 The standard scatter plot

We continue with the quakes example from Chapter 3. We saw in Figure 3.16 that the depths of the epicenters more or less fall into two clusters. The latitude and longitude are also recorded for each event, and together with depth could provide a three-dimensional view of how the epicenters are spatially distributed. Of course, scatter plots can only show us two dimensions at a time. As a first attempt, we could divide up the events into two groups by depth and plot latitude against longitude for each. Figure 5.1 is created using the by now familiar formula interface.

```
> xyplot(lat ~ long | cut(depth, 2), data = quakes)
```

The cut() function is used here to convert depth into a discrete factor by dividing its range into two equal parts. There does indeed seem to be some

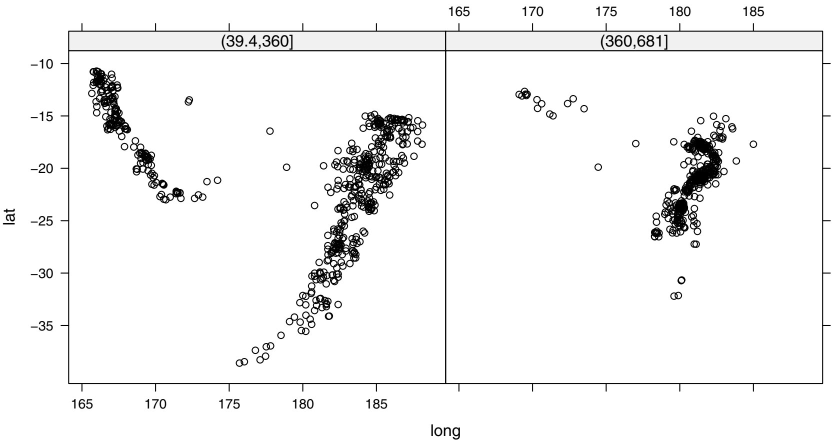

Figure 5.1. Scatter plots of latitude against longitude of earthquake epicenters, conditioned on depth discretized into two groups. The distribution of locations in the latitude–longitude space is clearly different in the two panels.

differentiation in the two parts; the cluster of locations towards the upper-left corner all but disappears in the second panel. The other cluster also appears to shrink, but it is not immediately clear if there is a spatial shift as well.

For our second attempt, we make several changes. We discretize the `depth` values into three groups instead of two, hoping to discern some finer patterns. We use a variant of the default strip function so that the name of the conditioning variable is included in the strips. We change the plotting symbol to dots rather than circles. Because the two axes have the same units (degrees), we constrain the scales to be isometric by specifying `aspect = "iso"`, which forces the aspect ratio to be such that the relationship between the physical and native coordinate systems (the number of data units per cm) is the same on both axes (of course, this does not account for the locations falling on a sphere and not a plane). Finally, and perhaps most important, we add a common reference grid to all three panels. Figure 5.2 is produced by

```
> xyplot(lat ~ long | cut(depth, 3), data = quakes,
         aspect = "iso", pch = ".", cex = 2, type = c("p", "g"),
         xlab = "Longitude", ylab = "Latitude",
         strip = strip.custom(strip.names = TRUE, var.name = "Depth"))
```

Thanks to the reference grid, careful inspection now confirms a subtle but systematic spatial pattern; for example, consider the neighbourhood of the $(185, -20)$ grid location in the three panels. Grids and other common (not data driven) reference objects are often invaluable in multipanel displays.

As we have seen in other contexts, superposition offers more direct between group comparison when it is feasible. In Figure 5.3 we show a grouped display with a slight variation; we discretize `depth` into three groups as before, but

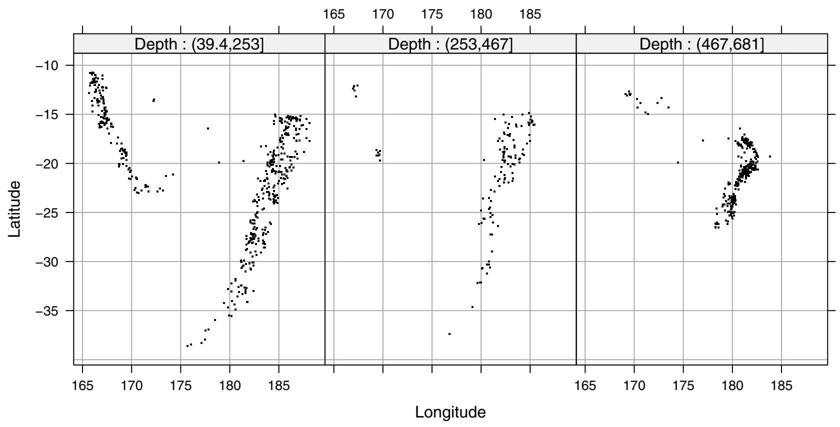

Figure 5.2. A slight variant of the previous plot. Depth is now discretized into three groups, a smaller plotting character reduces overlap, and a reference grid makes it easier to see trends across panels. In addition, the aspect ratio is such that the scales are now "isometric" (i.e., the number of data units per cm is the same on both axes). This aspect ratio is retained even when an on screen rendering is resized.

use equispaced quantiles as breakpoints, ensuring that all three groups have roughly the same number of points.

```
> xyplot(lat ~ long, data = quakes, aspect = "iso",
         groups = cut(depth, breaks = quantile(depth, ppoints(4, 1))),
         auto.key = list(columns = 3, title = "Depth"),
         xlab = "Longitude", ylab = "Latitude")
```

Although these examples all consistently hint at a certain spatial pattern, they all discretize the continuous `depth` variable. An obvious extension to this idea is to encode `depth` by a continuous gradient of some sort; color and symbol size are the most common choices. The human eye does not make very good quantitative judgments from such encodings, but relative ordering is conveyed reasonably well. In Figure 5.4, we use shades of grey to encode depth. There is no built-in support to achieve this in `xyplot()`, and we need to first create a suitable vector of colors to go with each observation. To this end, we use `cut()` again to convert `depth` into an integer code that is used to index a vector of colors.

```
> depth.col <- grey.colors(100)[cut(quakes$depth, 100, label = FALSE)]
```

We also reorder the rows to ensure that shallower points are plotted after deeper points; this requires us to reorder the color vector as well to keep the association between rows and colors valid. Figure 5.4 is produced by

```
> depth.ord <- rev(order(quakes$depth))
> xyplot(lat ~ long, data = quakes[depth.ord, ],
```

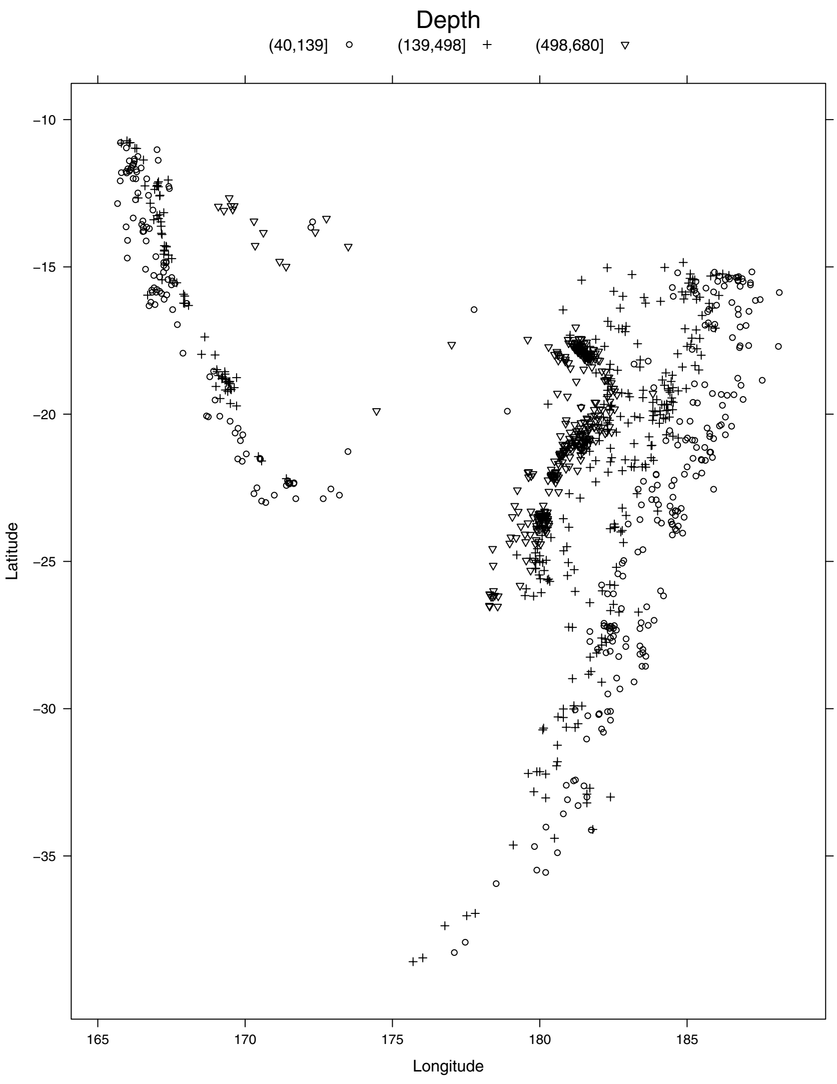

Figure 5.3. Scatter plots of latitude against longitude of earthquake epicenters. Depth, discretized into three slightly different groups, is now indicated using different plotting symbols within a single panel.

```
       aspect = "iso", type = c("p", "g"), col = "black",
       pch = 21, fill = depth.col[depth.ord], cex = 2,
       xlab = "Longitude", ylab = "Latitude")
```

This simple approach works in this case; however, it does not generalize to multipanel displays. Attempting to add a conditioning variable will lead to the same color vector being used in each panel, thus losing the correspondence between colors and rows in `quakes`.

5.2 Advanced indexing using `subscripts`

Fortunately, this is a common enough situation that a standard solution exists. It does, however, require the use of a simple panel function, and the reader is encouraged to revisit Section 2.5.3 before proceeding.

Our goal in this section is to create a multipanel version of Figure 5.4. A natural choice for a conditioning variable is `mag`, which gives the magnitude of each earthquake on the Richter scale, as we may be interested in knowing if the location of a quake has any relation to its magnitude. As `mag` is a continuous variable, we need to discretize it, just as we did with `depth`. However, this time, instead of `cut()`, we use `equal.count()` to create a *shingle*.

```
> quakes$Magnitude <- equal.count(quakes$mag, 4)
> summary(quakes$Magnitude)
Intervals:
   min  max count
1 3.95 4.55   484
2 4.25 4.75   492
3 4.45 4.95   425
4 4.65 6.45   415

Overlap between adjacent intervals:
[1] 293 306 217
```

As mentioned in Chapter 2, shingles are generalizations of factors for continuous variables, with possibly overlapping levels, allowing a particular observation to belong to more than one level. The `equal.count()` function creates shingles with overlapping levels that each have roughly the same number of observations (hence the name `equal.count`). The newly created `Magnitude` variable can now be used as a conditioning variable. By default, the intervals defining levels of a shingle relative to its full range are indicated by a shaded rectangle in the strip. To produce Figure 5.5, we use a call similar to the last one (this time creating a data frame with the desired row order beforehand), but with an explicit panel function.

```
> quakes$color <- depth.col
> quakes.ordered <- quakes[depth.ord, ]
> xyplot(lat ~ long | Magnitude, data = quakes.ordered, col = "black",
         aspect = "iso", fill.color = quakes.ordered$color, cex = 2,
```

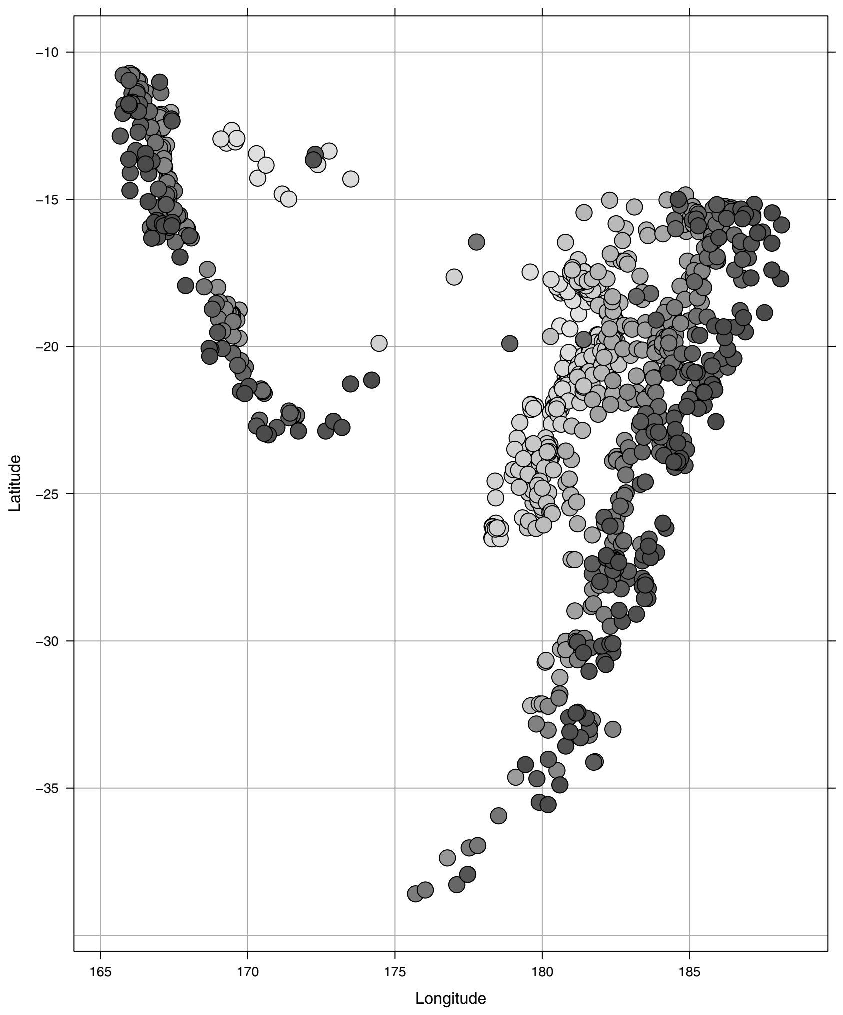

Figure 5.4. Latitude and longitude of earthquake epicenters, with the continuous depth variable encoded by fill color. A legend that describes the association between grey levels and the depths they represent would be a useful addition, but this is slightly more difficult. We show an example of such a legend in Figure 5.6.

```
        panel = function(x, y, fill.color, ..., subscripts) {
            fill <- fill.color[subscripts]
            panel.grid(h = -1, v = -1)
            panel.xyplot(x, y, pch = 21, fill = fill, ...)
        },
        xlab = "Longitude", ylab = "Latitude")
```

Before looking at the panel function, note the argument `fill.color` which contains the vector of colors corresponding to rows of the full data frame. As explained in Section 2.5.3, `xyplot()` will pass this argument on to the panel function as it does not recognize it itself. Thus, every time the panel function gets executed, it has access to the full vector of colors.

The problem of course is that the `x` and `y` values in the panel function only represent the subset of rows in that panel and not the full data. To use the colors correctly, we need to extract the colors associated with this subset from the full color vector `fill.color`. This is where the `subscripts` argument comes in. Along with other arguments, `xyplot()` can provide the panel function with an argument called `subscripts` containing a vector of integer indices that give the row numbers of the corresponding primary variables (`x` and `y` in this case). In other words, the correct color vector to go with `x` and `y` in a panel is `fill.color[subscripts]`; this fact is used in the panel function above to obtain the correct colors.

While we are discussing `subscripts`, we should note that the `groups` argument, already used in many examples, is essentially no different from the `fill.colors` argument used above; it simply gets passed on to the panel function in its entirety. The only thing special about `groups`, other than the fact that certain panel functions treat it specially, is that it gets evaluated in `data`. This is not true for other arguments, which is why we had to specify `fill.colors` explicitly as `quakes.ordered$color`. In fact, we can simply replace all references to `fill.color` by `groups` and obtain the same results, as in the following call that produces Figure 5.6,[1] with a slightly different color calculation that uses the convenient `level.colors()` function.

```
> depth.breaks <- do.breaks(range(quakes.ordered$depth), 50)
> quakes.ordered$color <-
      level.colors(quakes.ordered$depth, at = depth.breaks,
                   col.regions = grey.colors)
> xyplot(lat ~ long | Magnitude, data = quakes.ordered,
        aspect = "iso", groups = color, cex = 2, col = "black",
        panel = function(x, y, groups, ..., subscripts) {
            fill <- groups[subscripts]
            panel.grid(h = -1, v = -1)
            panel.xyplot(x, y, pch = 21, fill = fill, ...)
        },
        legend =
```

[1] Of course, the name `groups` is misleading in this example, and in any case this will not work when there are two or more variables to pass. See Figure 9.2 for such an example.

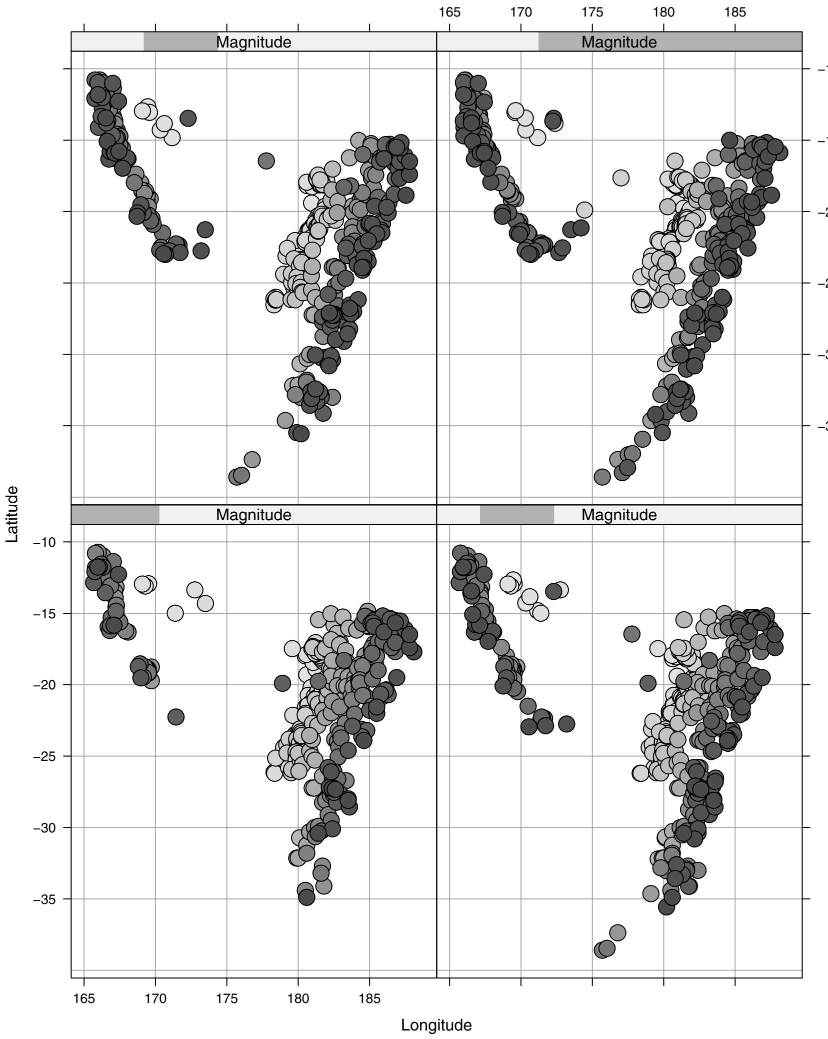

Figure 5.5. A multipanel version of Figure 5.4, conditioning on overlapping subsets of magnitudes.

type	Effect	Panel function
`"p"`	Plot points	
`"l"`	Join points by lines	
`"b"`	Both points and lines	
`"o"`	Points and lines overlaid	
`"S"`, `"s"`	Plot as step function	
`"h"`	Drop lines to origin ("histogram-like")	
`"a"`	Join by lines after averaging	`panel.average()`
`"r"`	Plot regression line	`panel.lmline()`
`"smooth"`	Plot LOESS smooth	`panel.loess()`
`"g"`	Plot a reference grid	`panel.grid()`

Table 5.1. The effect of various values of the `type` argument in `panel.xyplot()`. For some values, the effect will also depend on the value of the `horizontal` argument, as seen in Figure 5.7. Effects can be (and usually are) combined by specifying `type` as a vector. The actual rendering for some of these effects is performed by other specialized panel functions, and having access to them through the `type` argument is simply a convenience. The `type` argument also works for grouped displays transparently; when the `groups` argument is specified, `panel.xyplot` automatically calls another specialized panel function, `panel.superpose()`, to handle the necessary details.

```
    list(right =
         list(fun = draw.colorkey,
              args = list(key = list(col = grey.colors,
                                     at = depth.breaks),
                          draw = FALSE))),
    xlab = "Longitude", ylab = "Latitude")
```

Here, to make things interesting, we have also added a color key linking the colors to the depth values. This is somewhat nontrivial because `xyplot()` does not explicitly support such legends. What we have done, in fact, is to use a very general feature of lattice where an arbitrary legend can be specified in terms of a function that creates it. In this case, the relevant function is `draw.colorkey()`, which is called with arguments `key` and `draw` as specified in the call above. To learn more about this feature, consult Chapter 9 and the online documentation.

5.3 Variants using the `type` argument

As we have already seen, the `type` argument can be used to add a reference grid to each panel. It can also be used for a variety of other enhancements. Although it is typically supplied directly to `xyplot()`, it is actually an argument of the default panel function `panel.xyplot()`. Valid values of `type` and their effects are summarized in Table 5.1 and Figure 5.7. Its most common use is as `type = "l"` to plot lines instead of points (e.g., for time-series data). It is often supplied as a vector, in which case the effects of the individual components

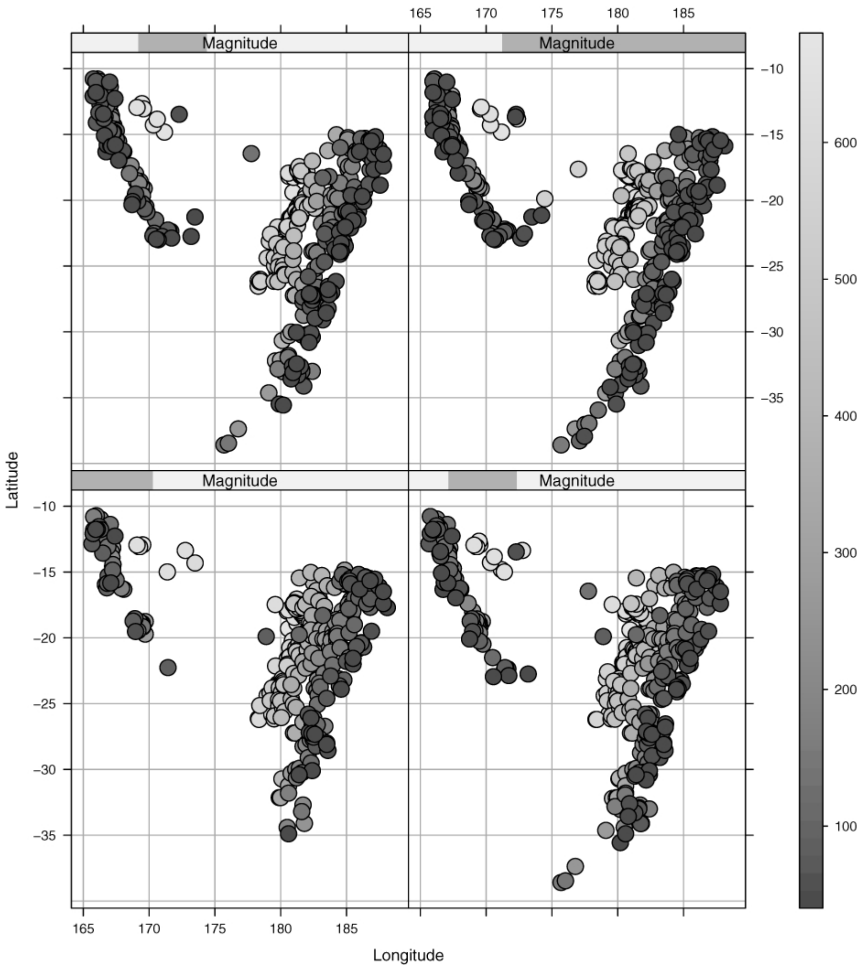

Figure 5.6. Variant of Figure 5.5, with a key describing the encoding of depth by fill color.

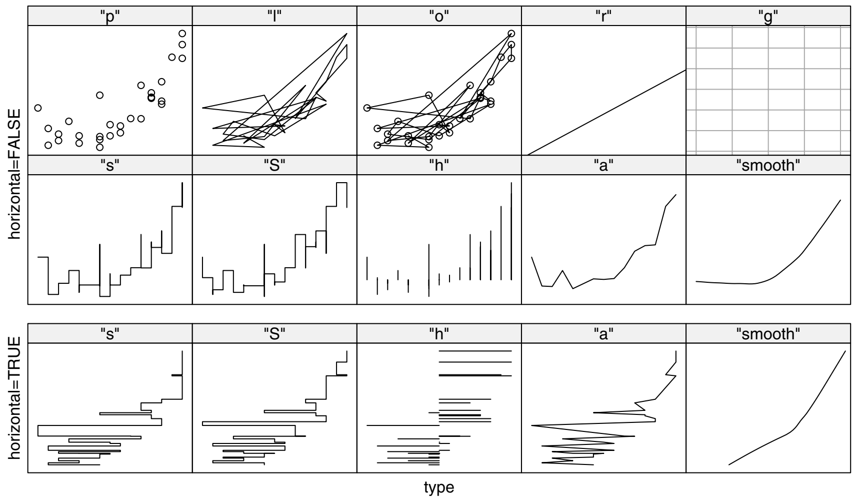

Figure 5.7. The effect of various values of `type` when specified as an argument to `xyplot()`, as well as `dotplot()`, `stripplot()`, and `splom()`. In each of these cases, `type` is eventually passed on to `panel.xyplot()` which does the actual plotting. Some of the types (e.g., `"s"`, `"S"`, and `"a"`) sort the data first. The step types `"s"` and `"S"` differ from each other by whether the first move is vertical or horizontal. The behavior for some types depends on the value of `horizontal`; this is more relevant for `dotplot()` and `stripplot()` where `horizontal` is set to `TRUE` automatically when the `y` variable is a factor. An example can be seen in Figure 4.2. The `"a"` type can be useful in creating interaction plots in conjunction with a `groups` argument.

are combined (except in certain grouped displays; see Figure 5.12). Some of the values (e.g., `"r"`, `"g"`, and `"smooth"`) simply cause other predefined panel functions to be called, and are provided as a convenience. As an example, consider another dataset on earthquakes, this one available in the MEMSS package, consisting of seismometer measurements of 23 large earthquakes in North America (Joyner and Boore, 1981).

```
> data(Earthquake, package = "MEMSS")
```

Ignoring the fact that multiple measurements are recorded from each earthquake, we wish to explore how the maximum horizontal acceleration at a measuring center (`accel`) depends on its distance from the epicenter (`distance`). It is fairly common to include a reference grid and a LOESS smooth (Cleveland and Devlin, 1988; Cleveland and Grosse, 1991) in such scatter plots. Without using the `type` argument, we could call

```
> xyplot(accel ~ distance, data = Earthquake,
         panel = function(...) {
             panel.grid(h = -1, v = -1)
             panel.xyplot(...)
```

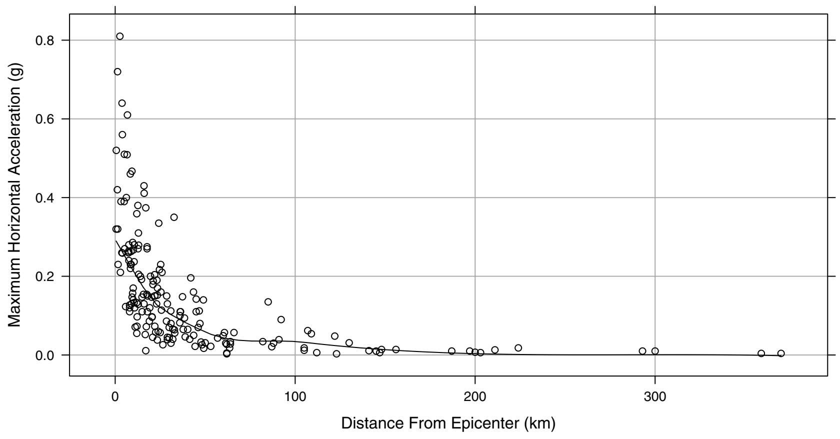

Figure 5.8. Scatter plot of acceleration versus distance in the `Earthquake` data, with a reference grid and a LOESS smooth. The asymmetry in the distribution of points on both axes, with only a few large values, suggests that a transformation is required.

```
            panel.loess(...)
        },
        xlab = "Distance From Epicenter (km)",
        ylab = "Maximum Horizontal Acceleration (g)")
```

This produces Figure 5.8. It is clear that transforming the data should improve the plot, and because both quantities are positive, we try plotting them on a logarithmic scale next in Figure 5.9. This time, however, we use the `type` argument instead of a custom panel function to get the equivalent result.

```
> xyplot(accel ~ distance, data = Earthquake,
         type = c("g", "p", "smooth"),
         scales = list(log = 2),
         xlab = "Distance From Epicenter (km)",
         ylab = "Maximum Horizontal Acceleration (g)")
```

This approach allows for concise and more readable code. It also avoids the concept of a panel function, which can be daunting for R beginners, while exposing some of its power. Of course, the disadvantage is that one is limited to the functionality built in to `panel.xyplot()`. Figure 5.10, produced by the following call, splits the data into three panels depending on the magnitude of the quakes, adds a common reference regression line to each panel, and uses an alternative smoothing method from the locfit package (Loader, 1999).

```
> library("locfit")
> Earthquake$Magnitude <-
      equal.count(Earthquake$Richter, 3, overlap = 0.1)
```

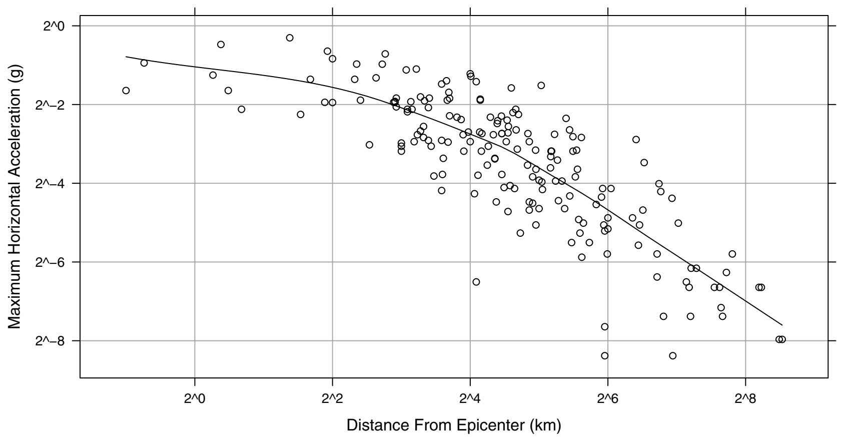

Figure 5.9. Scatter plot of acceleration versus distance on a logarithmic scale. The relationship between the variables is much more obvious in this plot. The axis labeling could be improved; this issue is taken up in Chapter 8.

```
> coef <- coef(lm(log2(accel) ~ log2(distance), data = Earthquake))
> xyplot(accel ~ distance | Magnitude, data = Earthquake,
         scales = list(log = 2), col.line = "grey", lwd = 2,
         panel = function(...) {
             panel.abline(reg = coef)
             panel.locfit(...)
         },
         xlab = "Distance From Epicenter (km)",
         ylab = "Maximum Horizontal Acceleration (g)")
```

This simple yet useful plot would not have been possible without a custom panel function.

5.3.1 Superposition and `type`

The `type` argument is useful in grouped displays as well. By default, it is interpreted just as described earlier; each component of `type` is used for each level of `groups`, with different graphical parameters. However, this is not always the desired behavior. Consider the `SeatacWeather` dataset in the latticeExtra package, which records daily temperature and rainfall amounts at the Seattle–Tacoma airport in the U.S. state of Washington over the first three months of 2007.

```
> data(SeatacWeather, package = "latticeExtra")
```

Suppose that we wish to plot the daily minimum and maximum temperatures as well as the daily rainfall in a single plot, with one panel for each month.

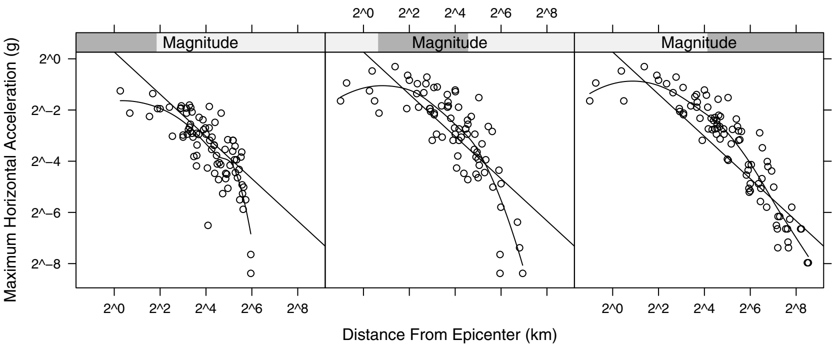

Figure 5.10. Scatter plots of acceleration by distance conditioned on earthquake magnitude. The common reference line makes it easier to see the shift across panels. The smooths are computed and plotted by the `panel.locfit()` function in the locfit package.

Getting all the variables into a single panel is simple if we use the extended formula interface described in Chapter 10; Figure 5.11 is produced by

```
> xyplot(min.temp + max.temp + precip ~ day | month,
         ylab = "Temperature and Rainfall",
         data = SeatacWeather, type = "l", lty = 1, col = "black")
```

The `lty` and `col` arguments are explicitly specified to prevent `panel.xyplot()` from using different ones for the three groups, which does not really help in this example. There are two problems with this plot. First, the rainfall measurements are in a completely different scale. Second, even though this is not obvious from Figure 5.11, most of the rainfall measurements are 0, which is special in this context, and joining the daily rainfall values by lines does not reflect this point. The first problem can only be solved by rescaling the rainfall values for the purpose of plotting (this brings up the issue of axis labeling, which we deal with later). As for the second problem, `type = "h"` seems to be the right solution. Thus, we would like to use `type = "l"` as before for the first two groups (`min.temp` and `max.temp`), and `type = "h"` for the third (`precip`). This can be achieved using the `distribute.type` argument[2] which, when `TRUE`, changes the interpretation of `type` by using the first component for the first level of `groups`, the second component for the second level, and so on. Figure 5.12 is produced by

```
> maxp <- max(SeatacWeather$precip, na.rm = TRUE)
> xyplot(min.temp + max.temp + I(80 * precip / maxp) ~ day | month,
         data = SeatacWeather, lty = 1, col = "black",
```

[2] As with `type`, this can be supplied directly to `xyplot()`, which will pass it to `panel.superpose()` through `panel.xyplot()`. See `?panel.superpose` for further details.

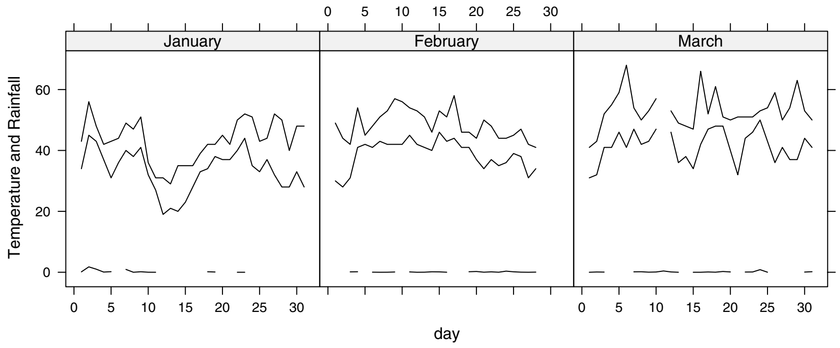

Figure 5.11. Daily meteorological data recorded at the Seattle–Tacoma airport. This figure represents an unsuccessful first attempt to incorporate both rainfall and temperature measurements in a single graphic. The problems arise because the units of rainfall and temperature are different, and the ranges of their numeric values are also different. In addition, `type = "l"` is not quite the right choice for rainfall.

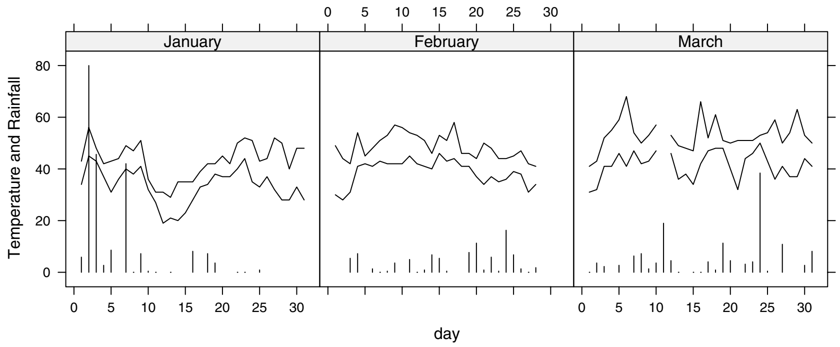

Figure 5.12. Daily rainfall and temperature in Seattle. The rainfall values have been rescaled to make their numeric range comparable to that of the temperature values. The `distribute.type` argument is used to change the interpretation of `type`.

```
ylab = "Temperature and Rainfall",
type = c("l", "l", "h"), distribute.type = TRUE)
```

This still leaves the issue of axis labeling, as Figure 5.12 gives us no information about what the precipitation amounts actually are. A quick-and-dirty solution is to create a fake axis inside using a panel function; the middle panel representing February conveniently has some space on the right that can be used for this purpose. Figure 5.13 is produced by adding a suitable panel function to the previous call.[3]

[3] `panel.number()` is a convenient accessor function described in Chapter 12.

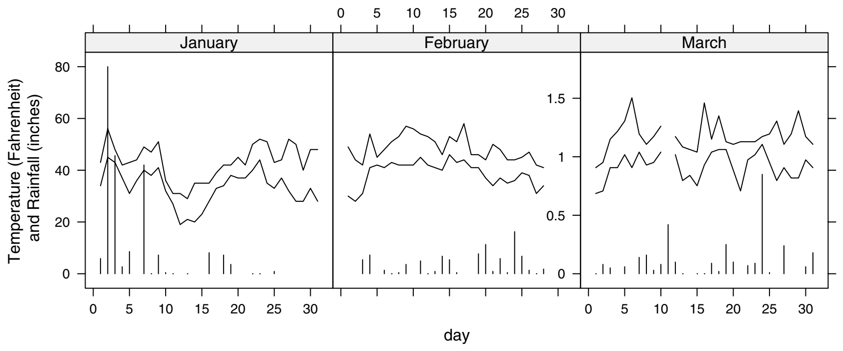

Figure 5.13. A variant of Figure 5.12 that includes a crude axis representing rainfall amounts.

```
> update(trellis.last.object(),
         ylab = "Temperature (Fahrenheit) \n and Rainfall (inches)",
         panel = function(...) {
             panel.xyplot(...)
             if (panel.number() == 2) {
                 at <- pretty(c(0, maxp))
                 panel.axis("right", half = FALSE,
                            at = at * 80 / maxp, labels = at)
             }
         })
```

The techniques outlined in Chapter 8 can be adapted to obtain a more systematic solution, perhaps by having the temperature axis on the left and the rainfall axis on the right. It should be noted, however, that using a common axis to represent multiple units is generally a bad idea, and should be avoided unless there is strong justification.

5.4 Scatter-plot variants for large data

Naïve scatter plots can easily become useless as the number of plotted points increases, causing overplotting. A simple but often effective remedy is to use partially transparent points (as in Figure 3.16); regions with extensive overplotting end up being darker than sparser regions. There are three problems with this solution: not all graphics devices in R support partial transparency, output files in vector formats such as PDF can still end up being large to the point of being impractical, and the solution is not scalable in the sense that with a large enough number of points, overplotting is likely to obscure patterns even with partially transparent points.

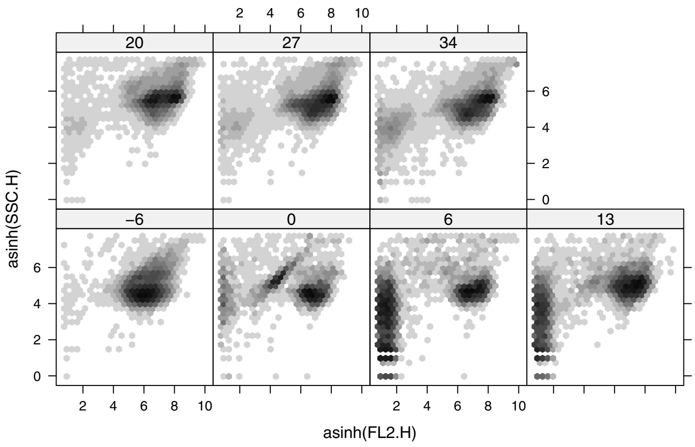

Figure 5.14. A large dataset visualized using hexagonal binning. Each panel visualizes the bivariate distribution of two measurements on cells in blood samples obtained from a blood and marrow transplant patient, taken before and after the transplant. The panels for days 6 and 13 show a large population not seen in the other days.

There are a number of approaches that attempt to deal with this problem, but none are implemented in the default panel function `panel.xyplot()`. In other words, any solution needs to be implemented separately as a custom panel function. One popular approach is to use hexagonal binning (Carr et al., 1987), where the x–y plane is tiled using hexagons which are then colored (or otherwise decorated) to indicate the number of points that fall inside. A panel function implementing this approach is available in the hexbin package (Carr et al., 2006), and can be used to visualize the `gvhd10` data encountered in Chapter 3 as follows.

```
> library("hexbin")
> data(gvhd10, package = "latticeExtra")
> xyplot(asinh(SSC.H) ~ asinh(FL2.H) | Days, gvhd10, aspect = 1,
         panel = panel.hexbinplot, .aspect.ratio = 1, trans = sqrt)
```

The result is shown in Figure 5.14. The `asinh()` transformation is largely similar to `log()`, but can handle negative numbers as well. The call is somewhat unwieldy, and can be misleading in the sense that grey levels do not necessarily represent the same number of points in a bin in each panel. A high-level function called `hexbinplot()`, defined in the hexbin package, provides a better interface that addresses this problem and also supports the automatic creation of meaningful legends. An example is given in Figure 14.4.

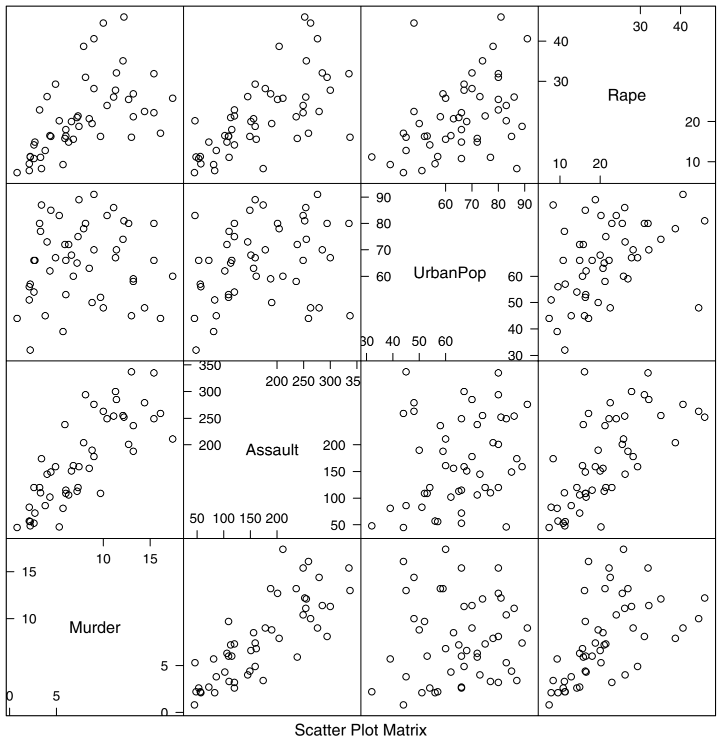

Figure 5.15. A scatter-plot matrix of the `USArrests` data. The `UrbanPop` variable records the percentage of urban population. The remaining variables record the number of arrests per 100,000 population for various violent crimes.

5.5 Scatter-plot matrix

Scatter-plot matrices, produced by `splom()`, are exactly what the name suggests; they are a matrix of pairwise scatter plots given two or more variables. Conditioning is possible, but it is more common to call `splom()` with a data frame as its first argument. Figure 5.15 is a scatter-plot matrix of the `USArrests` dataset, which contains statistics on violent crime rates in the 50 U.S. states in 1973. It is produced by

```
> splom(USArrests)
```

For conditioning with a formula, the primary variables are specified as `~x`, where `x` is a data frame. Figure 5.16 is produced by

```
> splom(~USArrests[c(3, 1, 2, 4)] | state.region,
        pscales = 0, type = c("g", "p", "smooth"))
```

The individual scatter plots are drawn by `panel.splom()`, which is an alias of `panel.xyplot()` and thus honors the same arguments; in particular, it

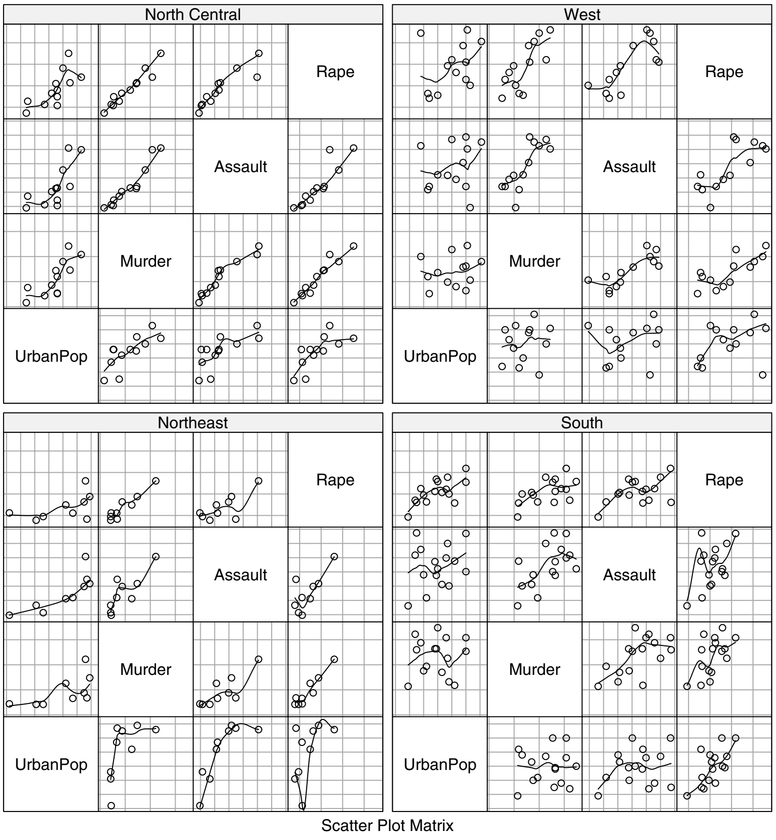

Figure 5.16. Scatter-plot matrices of the `USArrests` data, conditioned on geographical region. The columns have been reordered to make `UrbanPop` the first variable. Reference grids and LOESS smooths have been added as well.

interprets the `type` argument in the same manner. The `pscales` argument is used to suppress the axis labeling. Note that `USArrests` and `state.region` are separate datasets, and can be used together only because they record their data in the same order (alphabetically by state name). The subplots for different levels of `state.region` are slightly separated by default; the amount of separation can be customized using the `between` argument.

The concept of the panel function is somewhat confusing for `splom()`. By analogy with other high-level functions, the panel function should be the one that handles an entire packet (in this case, a conditional data frame subset) and is responsible for the individual scatter plots as well as their layout, including the names of the columns and the axis labeling along the diagonal. In practice, this is instead referred to as the *superpanel function*, and the panel function is the one that renders the individual scatter plots. The superpanel function is specified as the superpanel argument, which defaults to panel.pairs() and is seldom overridden. panel.pairs() allows different panel functions to be used for entries above and below the diagonal, and also allows a user-supplied function for the diagonal blocks. The help page for panel.pairs() describes these and other features in detail. In particular, the pscales and varnames arguments can be used to customize the contents of the diagonal panels relatively easily.

The next example illustrates the use of pscales and varnames. The mtcars dataset (Henderson and Velleman, 1981) records various characteristics of a sample of 32 automobiles (1973–1974 models), extracted from the 1974 *Motor Trend* magazine. Figure 5.17 is a scatter-plot matrix of a subset of the variables recorded, with the number of cylinders as a grouping variable. The varnames argument is used to specify more informative labels for the variables.

```
> splom(~data.frame(mpg, disp, hp, drat, wt, qsec),
        data = mtcars, groups = cyl, pscales = 0,
        varnames = c("Miles\nper\ngallon", "Displacement\n(cu. in.)",
                     "Gross\nhorsepower", "Rear\naxle\nratio",
                     "Weight", "1/4 mile\ntime"),
        auto.key = list(columns = 3, title = "Number of Cylinders"))
```

Note the use of a data argument, where the data frame specified inline in the formula is evaluated. Specifying each variable by name is not always convenient, and one might prefer the equivalent specification

```
> splom(~mtcars[c(1, 3:7)], data = mtcars, groups = cyl)
```

In this case, although groups is evaluated in data, mtcars[c(1, 3:7)] is not. If, as here, there are no conditioning variables, yet another alternative that avoids data altogether is

```
> splom(mtcars[c(1, 3:7)], groups = mtcars$cyl)
```

The appropriate choice in a given situation is largely a matter of taste.

5.5.1 Interacting with scatter-plot matrices

Scatter-plot matrices are useful for continuous multivariate data because they show all the data in a single plot, but they only show pairwise associations and are not particularly helpful in detecting higher-dimensional relationships. However, the layout of the scatter-plot matrix makes it an ideal platform for interactive exploration. In particular, the processes of "linking" and "brushing", where interactively selecting a subset of points in one scatter plot highlights the corresponding points in all the other scatter plots, can be extremely effective in finding hidden relationships. Such interaction with the output produced by splom() is possible, although the capabilities are greatly limited by the underlying graphics system. An example can be found in Chapter 12, which discusses the facilities available for interacting with lattice displays.

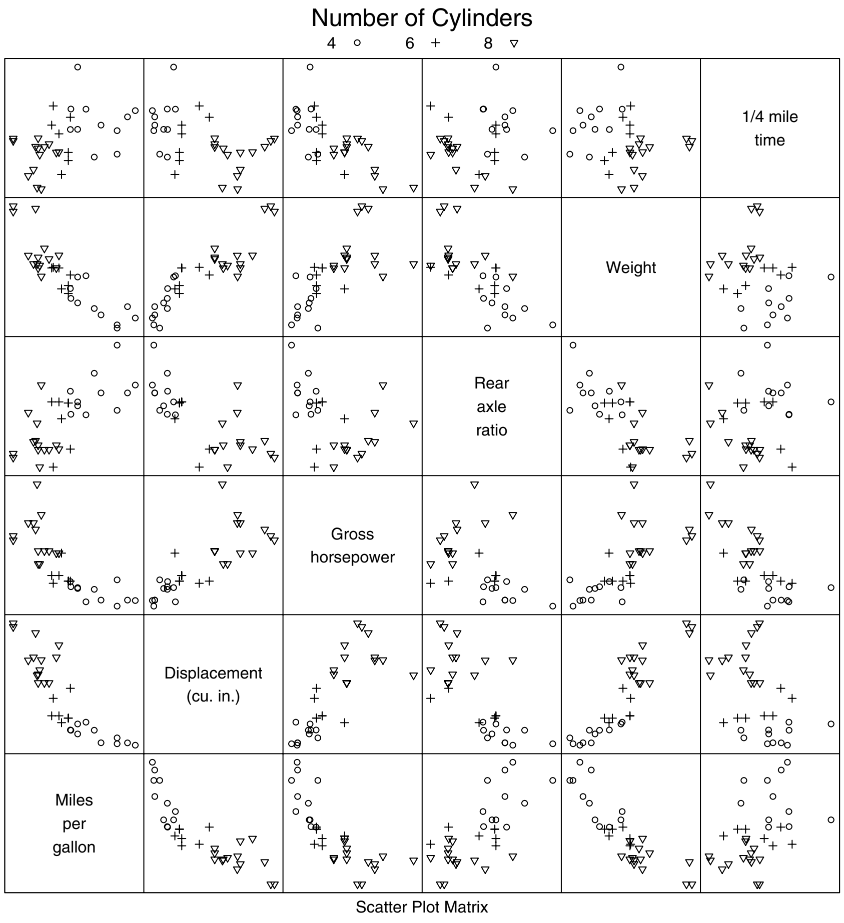

Figure 5.17. A scatter-plot matrix of a subset of the `mtcars` dataset, using the number of cylinders for grouping. As is often the case, using colors (rather than plotting characters) to distinguish between group levels is much more effective. For comparison, a color version of this plot is also available (see color plates).

5.6 Parallel coordinates plot

Like scatter-plot matrices, parallel coordinates plots (Inselberg, 1985; Wegman, 1990) are hypervariate in nature, that is, they show relationships between an arbitrary number of variables. Their design is related to univariate scatter plots; in fact, they are basically univariate scatter plots of all variables of interest stacked parallel to each other (vertically in the implementation in lattice), with values that correspond to the same observation linked by line segments. In other words, the combination of values defining each observation can be decoded by tracing the corresponding "polyline" through the univariate scatter plots for each variable. Parallel coordinates plots can be created using the `parallel()` function in lattice. The primary variable in `parallel()` is a data frame, as in `splom()`, and the formula is interpreted in the same manner. Figure 5.18 shows a parallel coordinates plot of a subset of the columns in the `mtcars` data, using the number of cylinders as a conditioning variable, and the number of carburetors as a grouping variable. The plot is produced by

```
> parallel(~mtcars[c(1, 3, 4, 5, 6, 7)] | factor(cyl),
           mtcars, groups = carb, layout = c(3, 1),
           auto.key = list(space = "top", columns = 3))
```

It is common to scale each variable individually before plotting it, but this can be suppressed using the `common.scale` argument of `panel.parallel()`.

Static parallel coordinates plots, as implemented in lattice, are not particularly useful. They allow pairwise comparisons only between variables that are adjacent. They do not make high-dimensional relationships easy to see; even bivariate relationships between adjacent variables are not always apparent. One point in their favor is that they often make multidimensional clusters easy to see; for example, we can see differences both between panels and between groups in Figure 5.18. This aspect translates to large datasets (if we are careful), as we show in our next example, which is a parallel coordinates plot of the first five columns of one sample in the `gvhd10` dataset. Figure 5.19 is produced by

```
> parallel(~ asinh(gvhd10[c(3, 2, 4, 1, 5)]), data = gvhd10,
           subset = Days == "13", alpha = 0.01, lty = 1)
```

The resulting plot clearly shows multiple high-dimensional clusters; however, the carefully chosen order of variables plays an important role in enabling this "discovery". As with scatter-plot matrices, their hypervariate nature makes parallel coordinates plots ideal candidates for dynamic linking and brushing. Unfortunately, lattice provides no facilities for such manipulation.

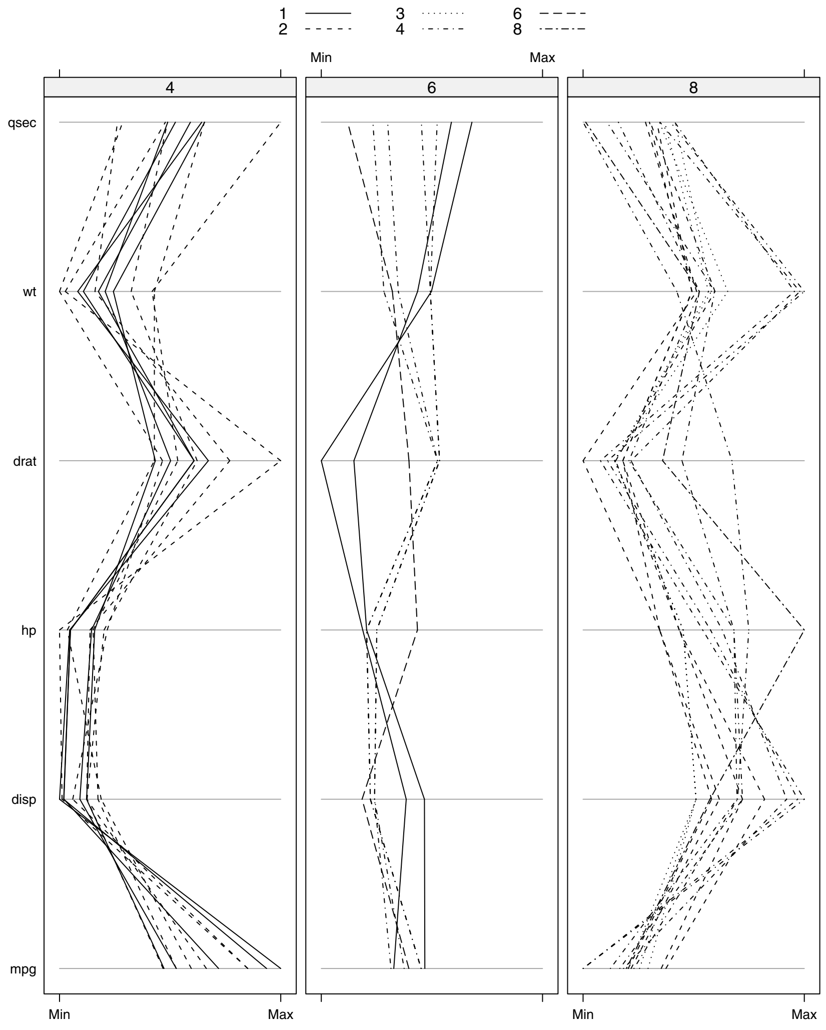

Figure 5.18. A parallel coordinates plot of the `mtcars` data, featuring both conditioning and grouping variables. The groups are not easily distinguishable in this black and white display; color would have been much more effective. Systematic multidimensional differences in the polyline patterns can be seen both between panels and between groups within panels.

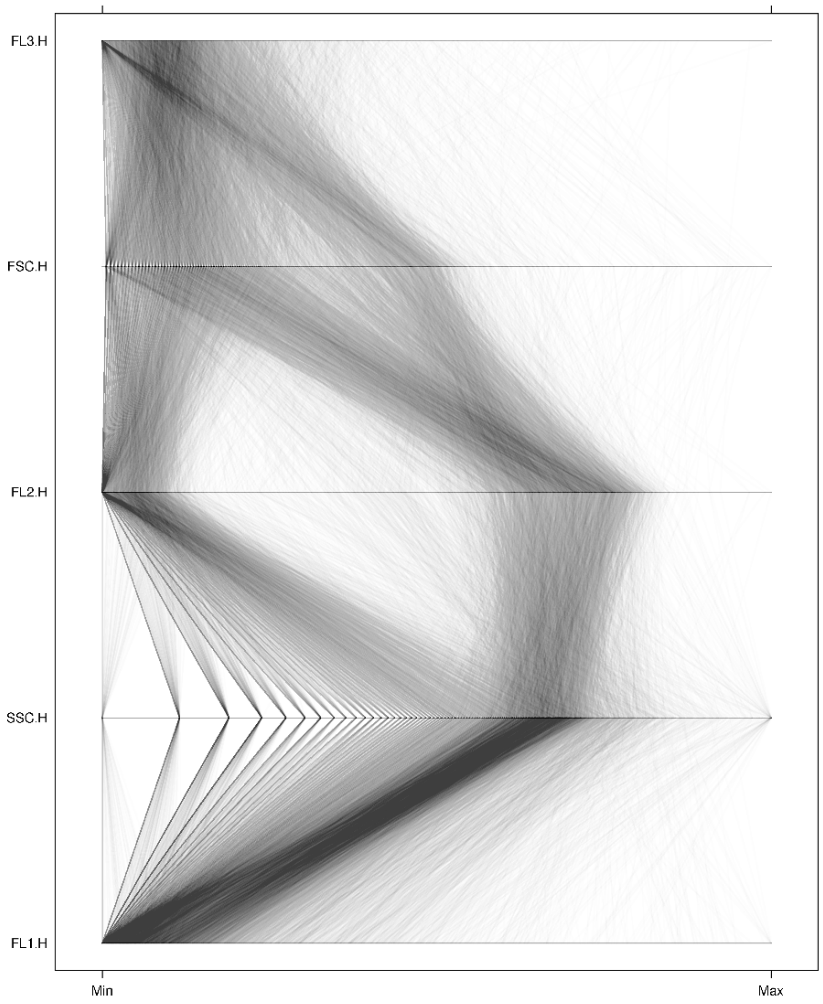

Figure 5.19. Parallel coordinates plot of one sample from the `gvhd10` dataset. The dataset is moderately large, and the display consists of 9540 polylines. The lines are partially transparent, largely alleviating potential problems due to overplotting. This also serves to convey a sense of density, because regions with more line segments overlapping are darker.