# 13

# Advanced Panel Functions

R is a complete programming language that allows, and indeed encourages, its users to go beyond the canned uses built into the system. The transition from user to programmer can be intimidating for the beginner to contemplate, but is almost inevitable after a point. In the context of lattice, this transition is most often necessitated by a desire to customize the display in small ways, perhaps just to add a common reference line to all panels. Such customizations are fairly basic in any serious use of lattice, and we have seen a number of examples throughout this book. In this chapter, which is meant for the more advanced user, we take a more formal look at panel functions, give pointers to the tools that might help in writing new ones, and finally discuss some nontrivial examples.

## 13.1 Preliminaries

Panel functions are like any other R function, except that they are expected to produce some graphical output when they are executed. They typically do so by calling a series of simpler panel functions, which might be viewed as building blocks of the complete display. Often, one also needs to manipulate the data available to the panel function before encoding them in the display. In this section, we describe the simple low-level panel functions available for use as building blocks, as well as some other related utilities. We demonstrate their use in creating data-driven displays in the subsequent sections.

### 13.1.1 Building blocks for panel functions

As we noted in Chapter 12, lattice is implemented using the low-level tools in the grid package. This has two important implications in the context of panel functions. First, lattice panel functions can make full use of grid primitives such as `grid.points()` and `grid.text()` and all their features. Second, lattice panel functions cannot make use of traditional graphics primitives[1] such as `points()` and `text()`. The plotting actions performed by a lattice panel function can consist entirely of grid function calls; in fact, grid primitives are more flexible than their traditional counterparts. A full discussion of grid is beyond the scope of this book, but a detailed exposition can be found in Murrell (2005) and the online documentation accompanying the grid package.

For those already familiar with traditional graphics, one practical drawback of grid is that it has an incompatible interface; that is, to reimplement a `text()` call in grid, one cannot simply change the name of the function to `grid.text()`; one also needs to modify the argument list. This can be a nuisance particularly when writing code that is intended for use both in R and S-PLUS (the latter does not have an implementation of grid). To make life easier, lattice provides analogues of several traditional graphics primitives; these are implemented using grid, but are intended to be drop-in replacements for the corresponding traditional graphics functions. For example,

`panel.points()` draws points (or lines, depending on the `type` argument) with an argument list similar to that of `points()`,
`panel.lines()` is analogous to `lines()`, and draws lines joining specified data points,
`panel.text()` is like `text()` and adds simple text or LaTeX-like expressions,
`panel.rect()` draws rectangles like `rect()`,
`panel.polygon()` draws polygons like `polygon()`,
`panel.segments()` draws line segments like `segments()`, and
`panel.arrows()` draws arrows like `arrows()`, with a slightly more general interface.

Needless to say, these functions are less flexible than the underlying grid functions, particularly in the choice of coordinate system. The lattice package also provides several "utility" panel functions that are not quite as generic, but are primarily intended for inclusion in other panel functions rather than for use by themselves. Among these are

`panel.fill()`, which fills the panel with a given color,
`panel.grid()`, which draws a reference grid,
`panel.abline()`, which draws reference lines of various kinds,
`panel.curve()`, which draws a curve defined by a mathematical expression, like `curve()`,
`panel.mathdensity()`, which draws a probability density function,
`panel.rug()`, which draws "rugs" like `rug()`,
`panel.loess()`, which adds a LOESS smooth of the supplied data,
`panel.lmline()`, which adds a regression line fit to the data,
`panel.qqmathline()`, which adds a line through two quantiles of the data and a theoretical distribution, and is primarily useful with `qqmath()`,

[1] This is not entirely true. See Figure 14.5.

`panel.violin()`, which draws violin plots, a useful alternative to box-and-whisker plots, and

`panel.average()`, which draws lines after aggregating and summarizing one variable by the unique levels of another.

Finally, each high-level function has its own panel function that can be reused in other contexts; these include `panel.bwplot()` and `panel.xyplot()`, among others. The `panel.superpose()` function is particularly useful for superposed displays. It conveniently handles separation of graphical parameters and allows another function to be specified as the `panel.groups` argument; this function is used as the panel function for each group and is supplied the appropriate graphical parameters.

We make no attempt to describe each of these functions in detail, as that would make this book longer than it already is. Instead, we refer the reader to their respective help pages.

### 13.1.2 Accessor functions

In principle, panel functions require no information beyond the data that are to be graphically encoded in that panel. In particular, it should not need to know where in the physical layout the current panel is located, nor should it worry about whether the current axis limits are appropriate for the data being encoded; it is expected that an appropriate data rectangle (viewport in grid jargon) with a suitable coordinate system has already been set up, and an appropriate clipping policy put in place, before the panel function is called. In practice, however, knowledge of these details can be important. Rather than supply such information through additional arguments, lattice provides a system of accessor functions that report the current state of the affairs when called from inside the panel function (or the strip or axis functions).

`current.panel.limits()` reports the limits of the current panel (viewport), typically in the native coordinate system, but possibly in any of the other systems supported by grid.

`packet.number()` returns an integer indicating which packet is being drawn. Packets are counted according to packet order, which is determined by varying the first conditioning variable the fastest, then the second, and so on.

`panel.number()` returns an integer counting which panel is being drawn, starting from one for the first panel. This is usually the same as the packet number, but not necessarily so.

`trellis.currentLayout()` returns a matrix with the same dimensions as the current layout of panels. The elements of the matrix indicate which packet (or panel) belongs in which position. For empty positions, the corresponding entry is 0.

`current.row()`, `current.column()` return the row or column position of the current panel in the layout.

`which.packet()` returns an integer vector as long as the number of conditioning variables, with each element an integer giving the current level of the corresponding variable.

These functions can be used while a *"trellis"* object is being plotted, as well as afterwards, while interacting with the display using the interface described in Chapter 12. As before, we refer the reader to the online documentation for more details.

### 13.1.3 Arguments

Panel functions are somewhat unusual in that they are rarely called by the user directly; they are instead called during the process of displaying a *"trellis"* object. This means, among other things, that the arguments available to a panel function are fully determined only in that context (recall that arguments supplied to a high-level function and not recognized by it are passed on to the panel function). To write a generally useful panel function, the author must take this fact into account. The arguments available will also depend on the relevant high-level function; for example, a panel function for `xyplot()` will expect arguments named `x` and `y` containing data, whereas one for `densityplot()` will only expect `x`. Usually, the most effective way to find out what arguments will be available (and how they should be interpreted) is to consult the help page of the default panel function, for example, `panel.densityplot()` for `densityplot()`. Of course, the most reliable way is to have the arguments listed explicitly; for example, using the panel function

```
> panel.showArgs <- function(...) str(list(...))
```

which is a function that simply writes out a compact summary of all its arguments. Not all potential arguments available to a panel function are necessarily supplied to it; only the ones that match the formal argument list of the panel function do, unless `...` is one of the formal arguments. It is generally good practice to have a `...` argument in panel functions and pass it on to further plotting functions, as this provides a simple mechanism to propagate graphical parameters.

One special argument in lattice panel functions is `subscripts`. If a panel function has a formal argument named `subscripts`, it will be called with `subscripts` containing the integer indices representing the rows in the original `data` (before any effect of `subset`) that define the packet used in that panel. Examples demonstrating the use of `subscripts` can be found in Section 5.2 and Chapter 12.

## 13.2 A toy example: Hypotrochoids and hypocycloids

Hypotrochoids are geometric curves traced out by a point within a circle that is rolling along "inside" another fixed circle. (Technically, the fixed circle can be smaller, in which case the moving circle is physically outside it. ) They are examples of a more general class of curves called roulettes, which are generated by one object rolling along another. Hypotrochoids can be parameterized by the equations

$$x(t) = (R - r)\cos t + d\cos(R - r)\frac{t}{r}$$
$$y(t) = (R - r)\sin t - d\sin(R - r)\frac{t}{r}$$

where $R$ is the radius of the fixed circle, $r$ the radius of the moving circle, and $d$ is the distance of the point being traced from the center of the latter. We can write a panel function that traces out this curve (with $R$ fixed at 1) as follows.

```
> panel.hypotrochoid <- function(r, d, cycles = 10, density = 30)
  {
      if (missing(r)) r <- runif(1, 0.25, 0.75)
      if (missing(d)) d <- runif(1, 0.25 * r, r)
      t <- 2 * pi * seq(0, cycles, by = 1/density)
      x <- (1 - r) * cos(t) + d * cos((1 - r) * t / r)
      y <- (1 - r) * sin(t) - d * sin((1 - r) * t / r)
      panel.lines(x, y)
  }
```

This function has two interesting features; first, it does *not* have a `...` argument, and second, none of the arguments is essential; even `r` and `d` are chosen randomly if they are missing. We show the implications of this in a moment.

First however, we consider hypocycloids, which are hypotrochoids with $d = r$, that is, the point being traced lying on the boundary of the moving circle. Hypocycloids are usually defined in terms of $k = 1/r$, and are closed curves when $k$ is rational, with $p$ "corners" if $k$ is expressed as a ratio of two coprime integers $p/q$. We can write a simple wrapper function that draws hypocycloids as

```
> panel.hypocycloid <- function(x, y, cycles = x, density = 30) {
      panel.hypotrochoid(r = x / y, d = x / y,
                         cycles = cycles, density = density)
  }
```

where `x` and `y` represent $q$ and $p$. We also need a prepanel function that defines the rectangle needed to fully contain a circle with unit radius centered at the origin:

```
> prepanel.hypocycloid <- function(x, y) {
      list(xlim = c(-1, 1), ylim = c(-1, 1))
  }
```

We can use the following code, producing Figure 13.1, to create a series of hypocycloids by varying the value of $p$ while keeping $q$ fixed.

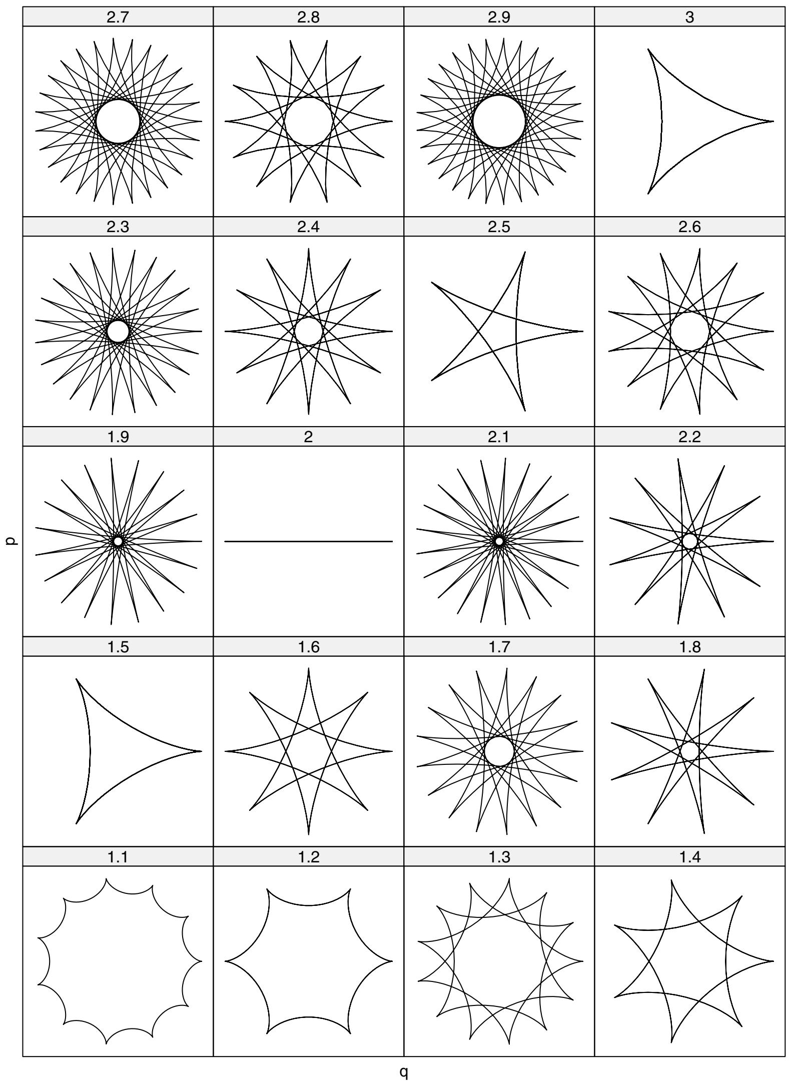

**Figure 13.1.** Hypocycloids with parameter $k$ varying from $11/10, 12/10, \ldots, 30/10$.

```
> grid <- data.frame(p = 11:30, q = 10)
> grid$k <- with(grid, factor(p / q))
> xyplot(p ~ q | k, grid, aspect = 1, scales = list(draw = FALSE),
         prepanel = prepanel.hypocycloid, panel = panel.hypocycloid)
```

This example is somewhat unusual in that the panel function is only provided two scalars at a time, which are used to compute and render a complex curve on the fly. Our next example is a whole lot more unusual. Figure 13.2 is produced by

```
> p <- xyplot(c(-1, 1) ~ c(-1, 1), aspect = 1, cycles = 15,
              scales = list(draw = FALSE), xlab = "", ylab = "",
              panel = panel.hypotrochoid)
> p[rep(1, 54)]
```

The panel function, `panel.hypotrochoid()`, does not accept arguments called `x` and `y`. Consequently, the $x$ and $y$ data specified in the formula do not get passed to the panel function at all; their sole purpose is to set up the data rectangle, avoiding the need for a prepanel function. In fact, the only argument explicitly passed on to the panel function is `cycles`, which determines the range of $t$ that defines the curve. Thus, every time the panel function is called, a randomly chosen hypotrochoid is drawn. We draw several of them at once by repeating the first packet several times.

## 13.3 Some more examples

### 13.3.1 An alternative density estimate

As a more serious example, consider the problem of density estimation. The `densityplot()` function computes and displays density estimates given raw data, but it is restricted to the kernel density estimation methods implemented in the `density()` function. Suppose that we wish instead to use the log-spline density estimate (Stone et al., 1997) implemented in the logspline package (Kooperberg, 2007). Because the tools to compute the estimate are already available, writing a panel function to display it is fairly simple. To make sure our panels have the right height, we also have to write a suitable prepanel function.

```
> library("logspline")
> prepanel.ls <- function(x, n = 50, ...) {
      fit <- logspline(x)
      xx <- do.breaks(range(x), n)
      yy <- dlogspline(xx, fit)
      list(ylim = c(0, max(yy)))
  }
> panel.ls <- function(x, n = 50, ...) {
      fit <- logspline(x)
      xx <- do.breaks(range(x), n)
      yy <- dlogspline(xx, fit)
      panel.lines(xx, yy, ...)
  }
```

We can now use these to produce Figure 13.3 with

```
> faithful$Eruptions <- equal.count(faithful$eruptions, 4)
> densityplot(~ waiting | Eruptions, data = faithful,
              prepanel = prepanel.ls, panel = panel.ls)
```

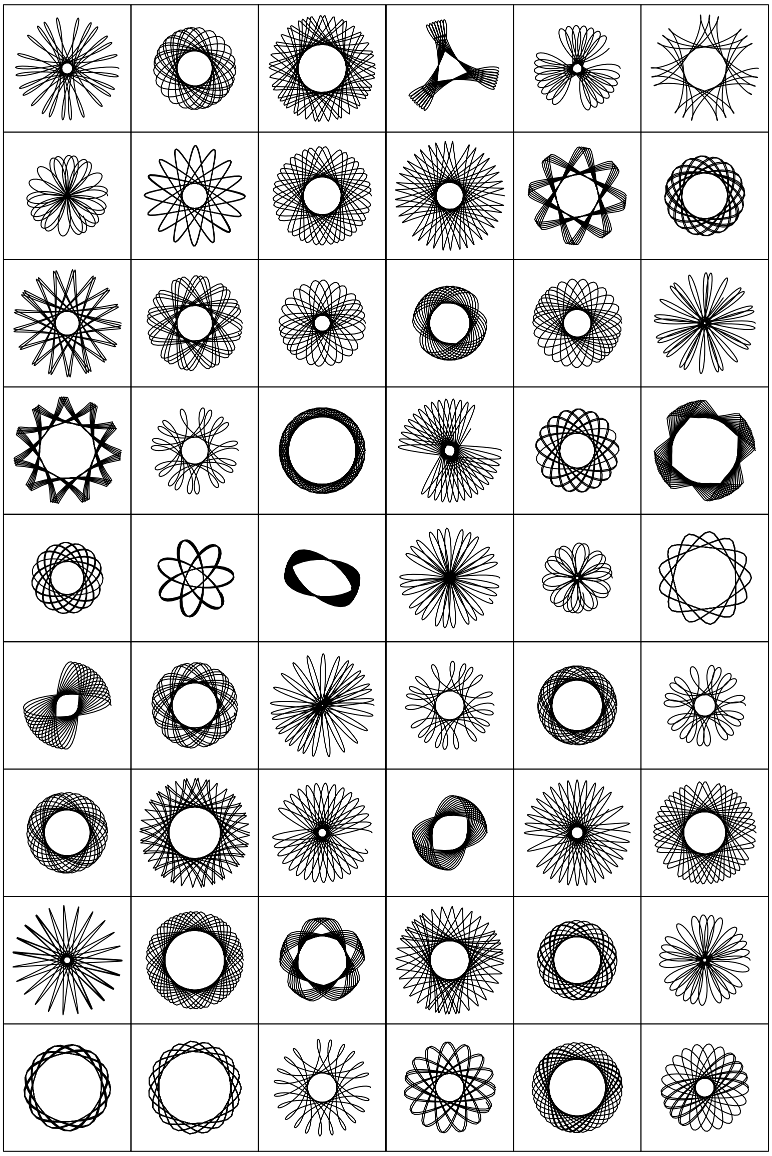

**Figure 13.2.** A series of hypotrochoids with randomly chosen parameters, reminiscent of the popular toy Spirograph®.

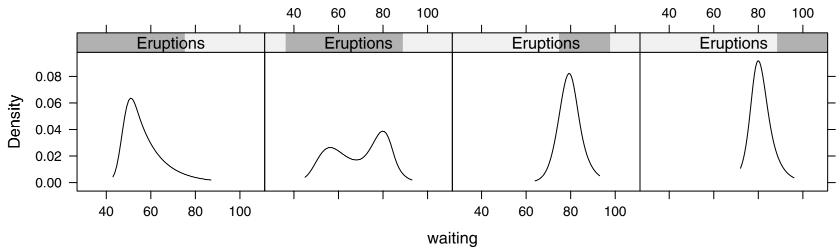

**Figure 13.3.** Conditional log-spline density estimates of waiting times before eruptions of the Old Faithful geyser, implemented with user-defined prepanel and panel functions.

## 13.3.2 A modified box-and-whisker plot

The next example is inspired by Tufte (2001), who describes a few variants of box-and-whisker plots that are motivated by the goal of reducing "non-data ink". In particular, the design we consider graphically summarizes the distribution of a continuous variable using a dot located at the median, and a couple of line segments extending from the first and third quartiles to the corresponding "extremes"; in other words, it is a box-and-whisker plot without the "box" (see `?boxplot.stats` for more concrete definitions). Our intention is not to comment on the merits of the design (especially because it is used here somewhat out of context), but simply to illustrate its implementation. A simple implementation is given by

```
> panel.bwtufte <- function(x, y, coef = 1.5, ...) {
      x <- as.numeric(x); y <- as.numeric(y)
      ux <- sort(unique(x))
      blist <- tapply(y, factor(x, levels = ux), boxplot.stats,
                      coef = coef, do.out = FALSE)
      blist.stats <- t(sapply(blist, "[[", "stats"))
      blist.out <- lapply(blist, "[[", "out")
      panel.points(y = blist.stats[, 3], x = ux, pch = 16, ...)
      panel.segments(x0 = rep(ux, 2),
                     y0 = c(blist.stats[, 1], blist.stats[, 5]),
                     x1 = rep(ux, 2),
                     y1 = c(blist.stats[, 2], blist.stats[, 4]),
                     ...)
  }
```

It is simple in the sense that it does not deal with "outliers" beyond the extremes and only produces vertical plots, but it is good enough to produce Figure 13.4 with

```
> data(Chem97, package = "mlmRev")
> bwplot(gcsescore^2.34 ~ gender | factor(score), Chem97,
```

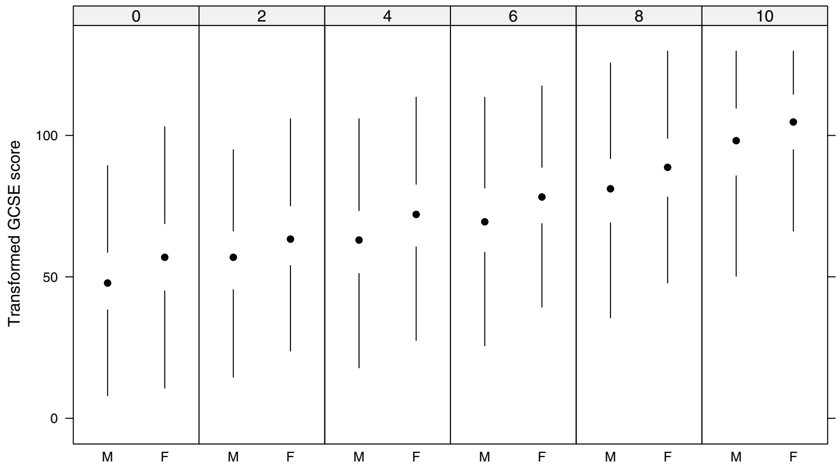

**Figure 13.4.** A variant of the standard box-and-whisker plot, showing the distribution of transformed GCSE scores by gender and the A-level chemistry examination score. The layout is the same as in Figure 3.12, but the encoding is inspired by an example from Tufte (2001).

```
          panel = panel.bwtufte, layout = c(6, 1),
          ylab = "Transformed GCSE score")
```

The result can be compared to Figure 3.12, which shows a regular box-and-whisker plot of the same data in the same layout.

### 13.3.3 Corrgrams as customized level plots

Corrgrams (Friendly, 2002) are visual representations of correlation matrices. They share the basic structure of a levelplot, but usually encode correlations by more than just color or grey level, and reorder the rows and columns by some measure of similarity. We continue with the example in Figure 6.13 to demonstrate a couple of variants.

```
> data(Cars93, package = "MASS")
> cor.Cars93 <-
      cor(Cars93[, !sapply(Cars93, is.factor)], use = "pair")
> ord <- order.dendrogram(as.dendrogram(hclust(dist(cor.Cars93))))
```

Our first panel function uses the ellipse package (Murdoch et al., 2007) to compute confidence ellipses representing correlation values, and additionally fills the ellipses with a color or grey level representing the correlation.

```
> panel.corrgram <-
      function(x, y, z, subscripts, at,
             level = 0.9, label = FALSE, ...)
  {
      require("ellipse", quietly = TRUE)
      x <- as.numeric(x)[subscripts]
      y <- as.numeric(y)[subscripts]
      z <- as.numeric(z)[subscripts]
      zcol <- level.colors(z, at = at, ...)
      for (i in seq(along = z)) {
          ell <- ellipse(z[i], level = level, npoints = 50,
                         scale = c(.2, .2), centre = c(x[i], y[i]))
          panel.polygon(ell, col = zcol[i], border = zcol[i], ...)
      }
      if (label)
          panel.text(x = x, y = y, lab = 100 * round(z, 2), cex = 0.8,
                     col = ifelse(z < 0, "white", "black"))
  }
```

The panel function does not deal with colors explicitly, relegating that computation to the `level.colors()` function. Figure 13.5 is produced by

```
> levelplot(cor.Cars93[ord, ord], at = do.breaks(c(-1.01, 1.01), 20),
            xlab = NULL, ylab = NULL, colorkey = list(space = "top"),
            scales = list(x = list(rot = 90)),
            panel = panel.corrgram, label = TRUE)
```

Because there is no explicit color specification, the defaults provided by the theme active during plotting are used. Our second variant is similar, but this time uses partially filled circles to represent correlations. The circles are drawn using grid functions `grid.polygon()` and `grid.circle()` directly.

```
> panel.corrgram.2 <-
      function(x, y, z, subscripts, at = pretty(z), scale = 0.8, ...)
  {
      require("grid", quietly = TRUE)
      x <- as.numeric(x)[subscripts]
      y <- as.numeric(y)[subscripts]
      z <- as.numeric(z)[subscripts]
      zcol <- level.colors(z, at = at, ...)
      for (i in seq(along = z))
      {
          lims <- range(0, z[i])
          tval <- 2 * base::pi *
              seq(from = lims[1], to = lims[2], by = 0.01)
          grid.polygon(x = x[i] + .5 * scale * c(0, sin(tval)),
                       y = y[i] + .5 * scale * c(0, cos(tval)),
                       default.units = "native",
                       gp = gpar(fill = zcol[i]))
          grid.circle(x = x[i], y = y[i], r = .5 * scale,
                      default.units = "native")
      }
  }
```

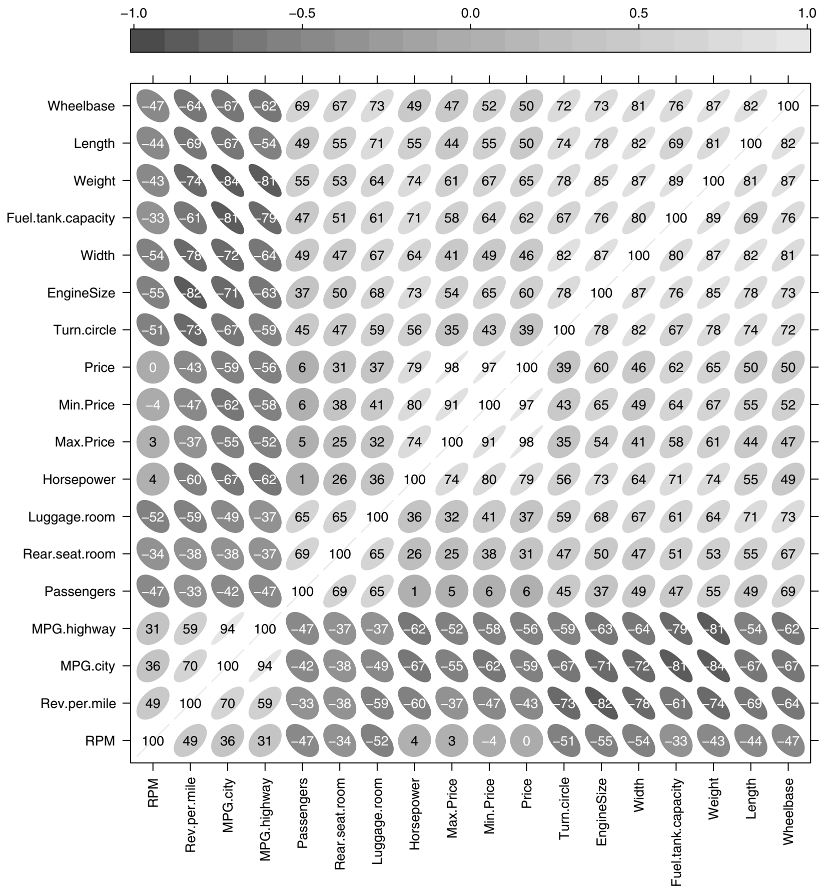

**Figure 13.5.** A corrgram, implemented as a `levelplot()` with a user-defined panel function, showing correlations using ellipses.

As before, color is not handled directly by the panel function. This time, however, we add a `col.regions` argument to the high-level call to `levelplot()`.

```
> levelplot(cor.Cars93[ord, ord], xlab = NULL, ylab = NULL,
            at = do.breaks(c(-1.01, 1.01), 101),
            panel = panel.corrgram.2,
            scales = list(x = list(rot = 90)),
            colorkey = list(space = "top"),
            col.regions = colorRampPalette(c("red", "white", "blue")))
```

col.regions is used by levelplot() to define the color key, but is also passed on to the panel function. From the perspective of the panel function, this is simply a part of the ... argument and is thus passed on unchanged to the level.colors() call, which does use it to compute suitable colors. The resulting display is shown with several other color plates in Figure 13.6.

## 13.4 Three-dimensional projections

Customizing the panel display in cloud() and wireframe(), the two high-level functions that make use of three-dimensional projection, is somewhat more involved. In addition to encoding the packet data, the panel function in this case is also responsible for drawing the bounding box and any axis annotation. One is usually interested only in changing the data-driven part of the display. This part can be controlled separately by specifying the panel.3d.cloud or panel.3d.wireframe arguments, which are technically arguments of the default panel function, and default to panel.3dscatter() and panel.3dwire(), respectively. More details can be found in the help page for these functions. Here, we give a simple example where a regular wireframe plot is supplemented by a contour plot "projected" onto the top surface of the bounding box. This involves computing the locations of the contour lines in the appropriate three-dimensional coordinate system, projecting it using ltransform3dto3d(), and drawing it. The following function executes these steps after calling panel.3dwire() to render the default wireframe display.

```
> panel.3d.contour <-
      function(x, y, z, rot.mat, distance,
               nlevels = 20, zlim.scaled, ...)
  {
      add.line <- trellis.par.get("add.line")
      panel.3dwire(x, y, z, rot.mat, distance,
                   zlim.scaled = zlim.scaled, ...)
      clines <-
          contourLines(x, y, matrix(z, nrow = length(x), byrow = TRUE),
                       nlevels = nlevels)
      for (ll in clines) {
          m <- ltransform3dto3d(rbind(ll$x, ll$y, zlim.scaled[2]),
                                rot.mat, distance)
          panel.lines(m[1,], m[2,], col = add.line$col,
                      lty = add.line$lty, lwd = add.line$lwd)
      }
  }
```

It can now be used in a call to wireframe() to produce Figure 13.7.

```
> wireframe(volcano, zlim = c(90, 250), nlevels = 10,
            aspect = c(61/87, .3), panel.aspect = 0.6,
            panel.3d.wireframe = "panel.3d.contour", shade = TRUE,
            screen = list(z = 20, x = -60))
```

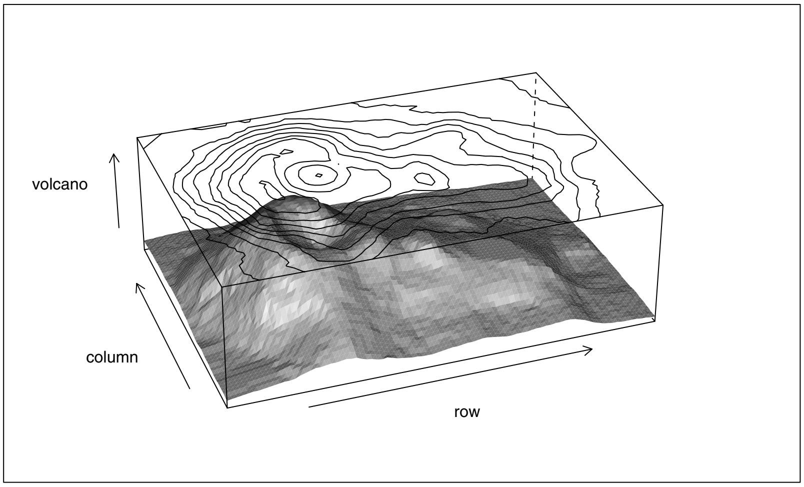

**Figure 13.7.** A three-dimensional view of the Maunga Whau volcano in Auckland created using `wireframe()`, with two-dimensional contours projected onto the top of the bounding box.

Figure 6.5 is another example of customized three-dimensional displays; the code to produce it is given later in this chapter.

It should be emphasized that the conventional R graphics model is far from optimal for three-dimensional displays; apart from the lack of dynamic manipulation, it has no high-level support for object occlusion, which makes it difficult to implement even the simplest of designs, such as a scatter plot combined with a fitted regression surface. Users who regularly work with three-dimensional displays should consider the rggobi and rgl packages, which provide interfaces to powerful alternative visualization systems.

## 13.5 Maps

Choropleth maps use color to encode a continuous or categorical variable on a map. Although somewhat specialized, choropleth maps are popular and effective in conveying spatial information. From an implementation perspective, there is nothing special about these plots; they are simply polygons with fill color derived from an external variable. The more important considerations are the practical ones of obtaining boundaries of the polygons defining the geographical units, and the associated data. In this section, we describe one approach that can be used to create choropleth maps using lattice, and point the reader to an alternative approach implemented in the latticeExtra package.

Tools to work with map data are available in the maps package (Becker et al., 2007) which contains, among other things, predefined boundary databases for several geographical units. In our examples, we use the "county" database, which contains information on counties in the United States. The map() function normally draws a map of a specified database, but can also be used to retrieve information about the polygons that define the map.

```
> library("maps")
> county.map <- map("county", plot = FALSE, fill = TRUE)
```

The fill argument causes the return value to be in a form that is suitable for use in polygon() (and hence panel.polygon()); it contains components x and y which are numeric vectors defining the boundaries, with NA values separating polygons. It also contains a vector of names for the polygons, which in this case represent U.S. counties.

```
> str(county.map)
List of 4
 $ x    : num [1:90997] -86.5 -86.5 -86.5 -86.6 ...
 $ y    : num [1:90997] 32.3 32.4 32.4 32.4 ...
 $ range: num [1:4] -124.7  -67.0   25.1   49.4
 $ names: chr [1:3082] "alabama,autauga" "alabama,baldwin" "alabama,..
 - attr(*, "class")= chr "map"
```

External data can be matched with polygons using these names. Getting the names into the same form may require some effort; we assume that this has already been done. Our first example uses the ancestry data in the latticeExtra package.

```
> data(ancestry, package = "latticeExtra")
> ancestry <- subset(ancestry, !duplicated(county))
> rownames(ancestry) <- ancestry$county
```

The data are derived from the U.S. 2000 census, and contain the most frequently reported ancestries in each county. As a first step, we pool the levels that appear infrequently.

```
> freq <- table(ancestry$top)
> keep <- names(freq)[freq > 10]
```

The row names of ancestry match the county names in county.map, and we use this fact to create a vector of ancestry values matching the map database.

```
> ancestry$mode <-
      with(ancestry,
           factor(ifelse(top %in% keep, top, "Other")))
> modal.ancestry <- ancestry[county.map$names, "mode"]
```

Finally, we use a color palette from the RColorBrewer package to produce Figure 13.8 (shown among the color plates). Thanks to the form of the value returned by map(), we can simply use panel.polygon() as our panel function, with a suitable vector of colors passed in as an argument to the high-level call.

```
> library("RColorBrewer")
> colors <- brewer.pal(n = nlevels(ancestry$mode), name = "Pastel1")
> xyplot(y ~ x, county.map, aspect = "iso",
         scales = list(draw = FALSE), xlab = "", ylab = "",
         par.settings = list(axis.line = list(col = "transparent")),
         col = colors[modal.ancestry], border = NA,
         panel = panel.polygon,
         key =
         list(text = list(levels(modal.ancestry), adj = 1),
              rectangles = list(col = colors),
              x = 1, y = 0, corner = c(1, 0)))
```

### 13.5.1 A simple projection scheme

Figure 13.8 plots county boundaries as if they lie on a plane, whereas they actually lie on a sphere. This is typically addressed by using one of several cartographic projection schemes, but another alternative is to convert the polygon boundaries into their three-dimensional representation, and use it in `cloud()`. This is demonstrated in the next example. First, we compute the coordinates of the respective polygons on the globe,

```
> rad <- function(x) { pi * x / 180 }
> county.map$xx <- with(county.map, cos(rad(x)) * cos(rad(y)))
> county.map$yy <- with(county.map, sin(rad(x)) * cos(rad(y)))
> county.map$zz <- with(county.map, sin(rad(y)))
```

and then define a panel function that draws polygons from three-dimensional data.

```
> panel.3dpoly <- function (x, y, z, rot.mat = diag(4), distance, ...)
  {
      m <- ltransform3dto3d(rbind(x, y, z), rot.mat, distance)
      panel.polygon(x = m[1, ], y = m[2, ], ...)
  }
```

Next, we use these to produce Figure 13.9 (see color plates).

```
> aspect <-
      with(county.map,
           c(diff(range(yy, na.rm = TRUE)),
             diff(range(zz, na.rm = TRUE))) /
           diff(range(xx, na.rm = TRUE)))
> cloud(zz ~ xx * yy, county.map, par.box = list(col = "grey"),
        aspect = aspect, panel.aspect = 0.6, lwd = 0.5,
        panel.3d.cloud = panel.3dpoly, col = colors[modal.ancestry],
        screen = list(z = 10, x = -30),
        key = list(text = list(levels(modal.ancestry), adj = 1),
                   rectangles = list(col = colors),
                   space = "top", columns = 4),
        scales = list(draw = FALSE), zoom = 1.1,
        xlab = "", ylab = "", zlab = "")
```

The `aspect` argument is required to ensure that the relative proportions of the $x$, $y$, and $z$ scales are appropriate.

Another example of the use of maps in a three-dimensional display was given in Figure 6.5. We are finally in a position to understand the call that produced it. The critical step is to define a function that draws the state boundaries on the $x$–$y$ plane.

```
> library("maps")
> state.map <- map("state", plot=FALSE, fill = FALSE)
> panel.3dmap <- function(..., rot.mat, distance, xlim, ylim, zlim,
                          xlim.scaled, ylim.scaled, zlim.scaled)
  {
      scale.vals <- function(x, original, scaled) {
          scaled[1] + (x-original[1]) * diff(scaled) / diff(original)
      }
      scaled.map <- rbind(scale.vals(state.map$x, xlim, xlim.scaled),
                          scale.vals(state.map$y, ylim, ylim.scaled),
                          zlim.scaled[1])
      m <- ltransform3dto3d(scaled.map, rot.mat, distance)
      panel.lines(m[1,], m[2,], col = "grey76")
  }
```

This is then used in combination with the default display to produce the desired effect.

```
> cloud(density ~ long + lat, state.info,
        subset = !(name %in% c("Alaska", "Hawaii")),
        panel.3d.cloud = function(...) {
            panel.3dmap(...)
            panel.3dscatter(...)
        },
        type = "h", scales = list(draw = FALSE), zoom = 1.1,
        xlim = state.map$range[1:2], ylim = state.map$range[3:4],
        xlab = NULL, ylab = NULL, zlab = NULL,
        aspect = c(diff(state.map$range[3:4]) /
                   diff(state.map$range[1:2]), 0.3),
        panel.aspect = 0.75, lwd = 2, screen = list(z = 30, x = -60),
        par.settings =
        list(axis.line = list(col = "transparent"),
             box.3d = list(col = "transparent")))
```

As in the previous example, much of the call is devoted to tweaking the aspect ratio and other such details.

### 13.5.2 Maps with conditioning

The use of `panel.polygon()` as the panel function does not work in multi-panel choropleth maps. The idea of using the map object as the data, with the actual variable of interest sneaked in as a color vector, is also somewhat artificial. A more natural approach is implemented by the `mapplot()` function in the latticeExtra package. We use it to obtain a multipanel choropleth map, this time visualizing a continuous response, the rate of death from cancer among males and females. The data are available in the USCancerRates dataset. The mapproj package (McIlroy et al., 2005) is used to apply a projection directly in the call to map(). Figure 13.10 (see color plates) is produced by

```
> library("latticeExtra")
> library("mapproj")
> data(USCancerRates)
> rng <- with(USCancerRates,
              range(rate.male, rate.female, finite = TRUE))
> nbreaks <- 50
> breaks <- exp(do.breaks(log(rng), nbreaks))
> mapplot(rownames(USCancerRates) ~ rate.male + rate.female,
          data = USCancerRates, breaks = breaks,
          map = map("county", plot = FALSE, fill = TRUE,
                    projection = "tetra"),
          scales = list(draw = FALSE), xlab = "",
          main = "Average yearly deaths due to cancer per 100000")
```

This example illustrates an important point, namely, that custom panel functions, although affording tremendous flexibility, are primarily useful in situations where the role of the variables involved fit into one of a few predefined models. In the next chapter, we discuss how to develop new high-level display functions, such as mapplot(), that let us bypass such constraints.