14

New Trellis Displays

Each high-level function in lattice is intended to create a certain type of statistical display by default. Many variations are already built into the default panel functions and can be activated with additional arguments in a high-level function call itself. More extensive modifications can be made by writing custom panel functions, as we have seen throughout this book and particularly in Chapter 13.

Although panel functions can be used to implement entirely novel visualizations, trying to shoehorn such a display into a function intended for another purpose is mostly useful as a one-off, quick-and-dirty solution. For a systematic implementation that could perhaps be used by others, it is often more sensible to create a new function whose name better reflects the nature of the visualization. On the other hand, existing function names are sometimes perfectly appropriate, and it is the data which are in a form that is not directly usable. A typical example of this is a univariate time series; there is really only one choice for the x and y variables in the `xyplot()` call that produced Figure 10.17, and the need for a new function to hide the use of a formula seems wasteful.

Rather than trying to anticipate all potential use cases, lattice provides the groundwork for further extensions by making use of the object-oriented features of R. Each high-level function in lattice is generic, with method dispatch possible on the first argument `x` and possibly (using the formal *S4* system) the second argument `data`. New high-level display functions can be written either as new methods for existing generic functions, or, if it seems appropriate, as an entirely new function which should itself be generic to allow further specialized methods. In this chapter, we give examples of both new methods and new high-level functions implemented using the framework provided by lattice. These can, it is hoped, serve as models for further extensions.[1]

[1] Note that this is by no means the only way to extend lattice; the Hmisc and nlme packages are widely used examples that take different approaches.

14.1 *S3* methods

The high-level functions in lattice are generic functions, which means that new methods can be written to display objects based on their class. Such methods usually end up calling the corresponding *"formula"* method after some preliminary processing. They may have different defaults for some arguments, and even a few new ones. There are a few such methods built into lattice, such as `histogram()` and `qqmath()` methods for numeric vectors, `levelplot()` and `wireframe()` methods for matrices, and (somewhat nontrivial) `barchart()` and `dotplot()` methods for contingency tables as produced by `table()` or `xtabs()`.

Here we give as examples two other methods, defined in the latticeExtra package, for the `xyplot()` generic. The first is for plotting time-series objects, and essentially performs the same task as the `cutAndStack()` function defined in Section 10.5.3. Figure 14.1 is produced by

```
> library("latticeExtra")
> xyplot(sunspot.year, aspect = "xy",
         strip = FALSE, strip.left = TRUE,
         cut = list(number = 4, overlap = 0.05))
```

This time, there is no need to write a wrapper function, and the cuts are specified using a new argument that is only meaningful for this method. Our second example, which is slightly more involved, is also related to time-series data. The `stl()` function decomposes a periodic time-series into seasonal, trend, and irregular components using LOESS (Cleveland et al., 1990). The result is an object of class *"stl"*; the `xyplot()` method for this class is used below to visualize the decomposition of the `biocAccess` data seen previously in Figure 8.2. The data are not in the form of a time-series, so we create one on the fly. To keep the plot from getting too compressed horizontally, data from only the first two months are used.

```
> data(biocAccess, package = "latticeExtra")
> ssd <- stl(ts(biocAccess$counts[1:(24 * 30 * 2)], frequency = 24),
             "periodic")
> xyplot(ssd, xlab = "Time (Days)")
```

The plot shows clear trends of decreased activity during weekends, as well as regular "seasonal" peaks of activity within each day (which happens to be caused by a poorly set up mirror).

Both these examples are primarily useful as demonstrations; *"stl"* objects have a `plot()` method that uses traditional graphics to produce an equivalent visualization, and the zoo package, which deals primarily with time-series data, has more general `xyplot()` methods for time-series objects. Other examples that can serve as prototypes are available in the coda package, and of course in the lattice package itself.

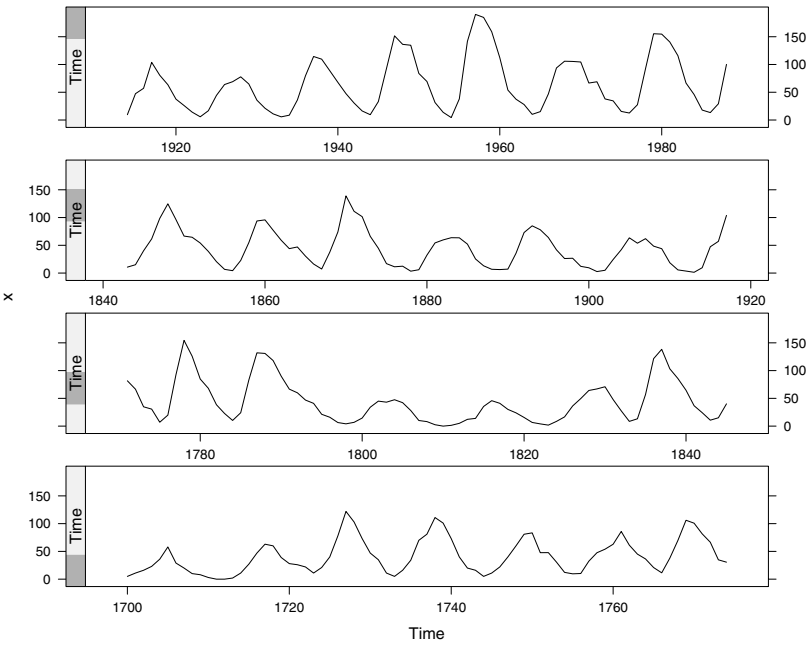

Figure 14.1. A cut-and-stack plot of the yearly number of sunspots between 1700 and 1988, created using an `xyplot()` method for time series data. The aspect ratio, chosen using the 45° banking rule, makes it easy to see that the ascent into peaks are usually steeper than the descents.

14.2 *S4* methods

Although the *S3* scheme works well for plotting highly structured objects, it is insufficient in situations where the flexibility of a formula interface is desirable, but with data objects that do not fit into the restrictive data frame paradigm.

This is important, for example, in the context of modern high-throughput bioinformatics data, where each "response" consists of thousands of measurements on the basic experimental unit, and covariate information on each experimental unit is stored as "phenotype data". The Bioconductor project (Gentleman et al., 2004) handles such data by defining new container classes. We can use such classes as alternative data sources in lattice methods using the multiple dispatch facilities in the *S4* system.[2] In Figure 14.3, we use the

[2] In the *S3* system, the specific method used when a generic function is called depends only on the class of one argument. *S4* generic functions, on the other hand, can select methods based on the classes of multiple arguments. This feature is known as multiple dispatch. The *S4* system has many other features not directly relevant for us; see Chambers (1998) for details.

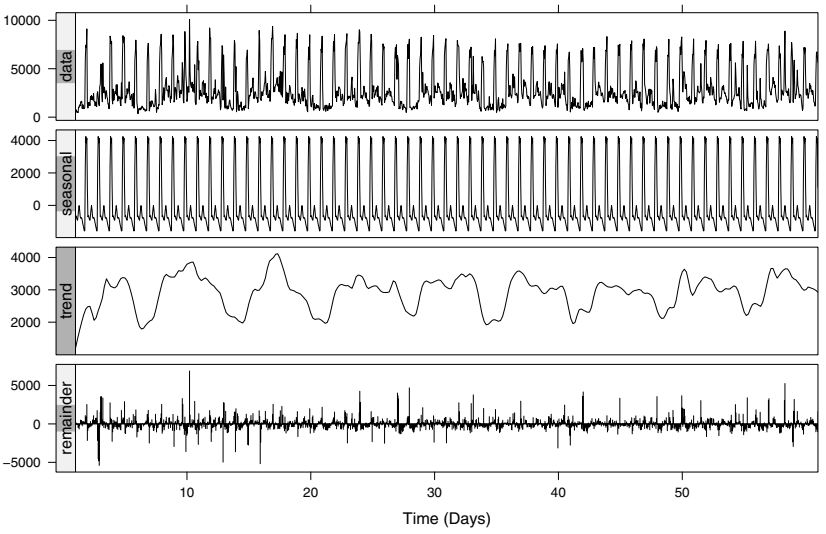

Figure 14.2. An STL decomposition of the hourly number of accesses to `http://www.bioconductor.org` over the period of two months, created using an `xyplot()` method for *"stl"* objects. The "trend" is periodic with a dip during weekends. The "seasonal" component shows the pattern of accesses over a day, with the spikes very likely due to automated activities such as mirroring.

`densityplot()` method from the Bioconductor package flowViz (Duong et al., 2007) that dispatches on a *"formula"* `x` and a *"flowSet"* `data`.

```
> library("flowViz")
> data(GvHD, package = "flowCore")
> densityplot(Visit ~ `FSC-H` | Patient, data = GvHD)
```

The primary challenge in such examples is not multiple dispatch, but rather the handling of potentially large datasets. In this example, `GvHD` is a *"flowSet"* object containing data from 35 samples. Two of the variables in the formula (`Patient` ID and `Visit` number) represent phenotype data associated with the samples. Each sample produces a (on average) $15,000 \times 8$ data matrix; columns in these data matrices (e.g., `FSC-H`) are the variables we are interested in visualizing. The naïve approach would be to convert the full data into an expanded data frame (a "join" operation), but this would produce a data frame with roughly $15,000 \times 35$ rows! The solution used in the flowViz package is to use only the phenotype data to construct a lattice call; the actual data are stored in an environment (as part of the design of the *"flowSet"* class), and the panel and prepanel functions access only one sample at a time as necessary. The flowViz package contains several other examples of *S4* methods for high-level lattice functions.

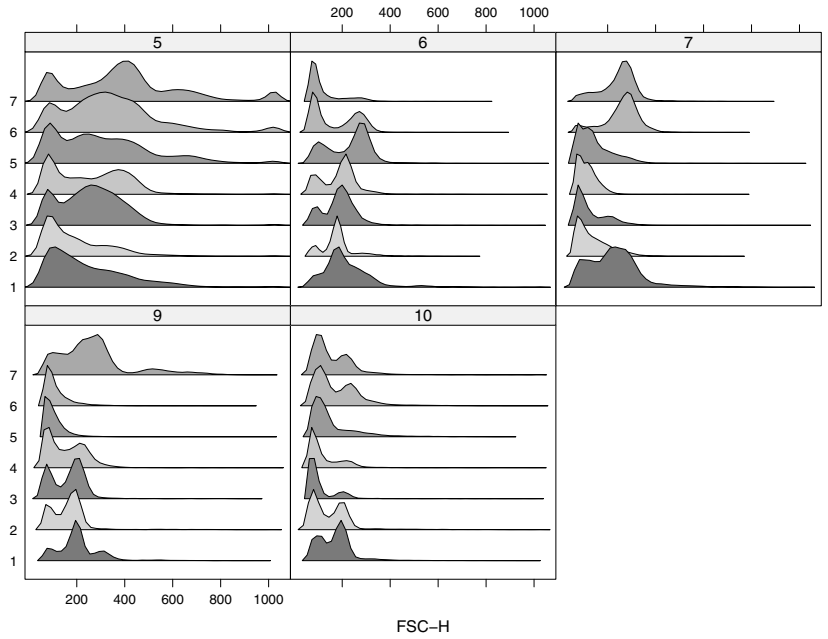

Figure 14.3. A visualization of the `FSC-H` channel in the `GvHD` data, created using a `densityplot()` method with signature `(x = "formula", data="flowSet")`. Each panel represents one patient, and the estimated densities of `FSC-H` for multiple visits are stacked on top of each other within each panel.

14.3 New functions

As we have already seen, existing generic function names may not be meaningful for new visualizations, and a completely new function name is often warranted. It is not necessary to define these functions as generic, but doing so has the benefit of encouraging future extensions. With a coordinated choice of argument names, it also allows multiple methods in multiple packages (perhaps written by different authors) to be used simultaneously without causing naming conflicts. We have already seen the `mapplot()` function in the latticeExtra package used to produce Figure 13.10. Another prototypical example is the `hexbinplot()` function from the hexbin package, which is used as follows to produce Figure 14.4.

```
> library("hexbin")
> data(NHANES)
> hexbinplot(Hemoglobin ~ TIBC | Sex, data = NHANES, aspect = 0.8,
             trans = sqrt, inv = function(x) x^2)
```

The need to add an appropriate legend makes the implementation of `hexbinplot()` particularly instructive; the difficulty arises from the lack of a formal

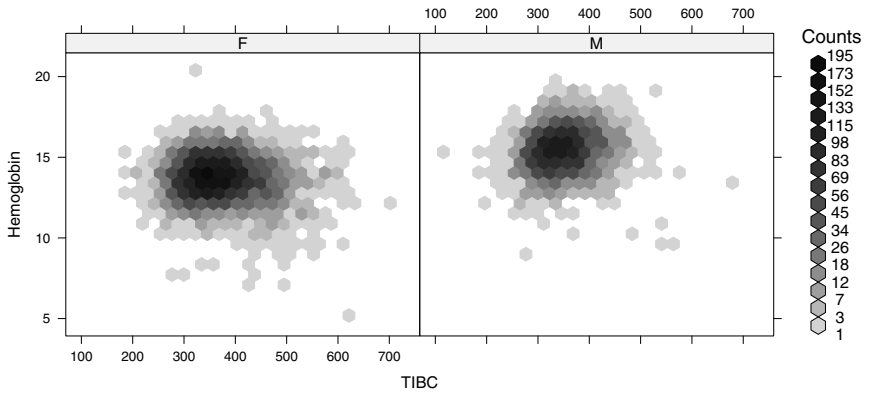

Figure 14.4. A conditional plot implementing the hexagonal binning algorithm of Carr et al. (1987), created using the `hexbinplot()` function. This example is somewhat challenging for the Trellis model, as it requires the panels to communicate information regarding bin counts to the legend.

mechanism to allow the panel function to communicate with the legend. The form of the legend itself poses another challenge, and requires nontrivial programming using grid. The interested reader is referred to the source code of the hexbin package for details.

14.3.1 A complete example: Multipanel pie charts

Care must be taken when writing new high-level functions to ensure that the expected nonstandard evaluation behavior is retained. Methods that call other high-level functions often need to delay the evaluation of certain arguments, and one way to do so is to make use of `match.call()` and `eval.parent()`. For our final example, we define a new high-level function that explicitly illustrates this approach.

The lattice package does not have a high-level function to draw pie charts because the information encoded by a pie chart can be conveyed more effectively by other graphs. They are a very familiar design nonetheless, and using the gridBase package (Murrell, 2005), which allows us to combine the normally incompatible traditional and grid graphics, we write a panel function that draws pie charts with minimal effort on our part:

```
> panel.piechart <-
      function(x, y, labels = as.character(y),
               edges = 200, radius = 0.8, clockwise = FALSE,
               init.angle = if(clockwise) 90 else 0,
               density = NULL, angle = 45,
```

```
              col = superpose.polygon$col,
              border = superpose.polygon$border,
              lty = superpose.polygon$lty, ...)
  {
      stopifnot(require("gridBase"))
      superpose.polygon <- trellis.par.get("superpose.polygon")
      opar <- par(no.readonly = TRUE)
      on.exit(par(opar))
      if (panel.number() > 1) par(new = TRUE)
      par(fig = gridFIG(), omi = c(0, 0, 0, 0), mai = c(0, 0, 0, 0))
      pie(as.numeric(x), labels = labels, edges = edges,
          radius = radius, clockwise = clockwise,
          init.angle = init.angle, angle = angle,
          density = density, col = col,
          border  = border, lty = lty)
  }
```

Because the form of data required by a pie chart is similar to that in a bar chart, we simply need to define a new function that calls `barchart()` with a new default panel function. Such a function is defined as

```
> piechart <- function(x, data = NULL, panel = "panel.piechart", ...)
  {
      ocall <- sys.call(sys.parent())
      ocall[[1]] <- quote(piechart)
      ccall <- match.call()
      ccall$data <- data
      ccall$panel <- panel
      ccall$default.scales <- list(draw = FALSE)
      ccall[[1]] <- quote(lattice::barchart)
      ans <- eval.parent(ccall)
      ans$call <- ocall
      ans
  }
```

Although this is not quite the standard way of writing functions in the S language, it ensures that arguments passed in as part of the `...` argument of `piechart()` (which may include arguments such as `groups` and `subset` which follow special evaluation rules) are not evaluated prematurely. This function can now be used to produce Figure 14.5.

```
> par(new = TRUE)
> piechart(VADeaths, groups = FALSE, xlab = "")
```

We have ignored our own recommendation in not defining `piechart()` as a generic function, but this is easily fixed. Even as it stands, `piechart()` calls `barchart()` with minimal processing of its arguments, and consequently inherits the method dispatch behavior of `barchart()`.

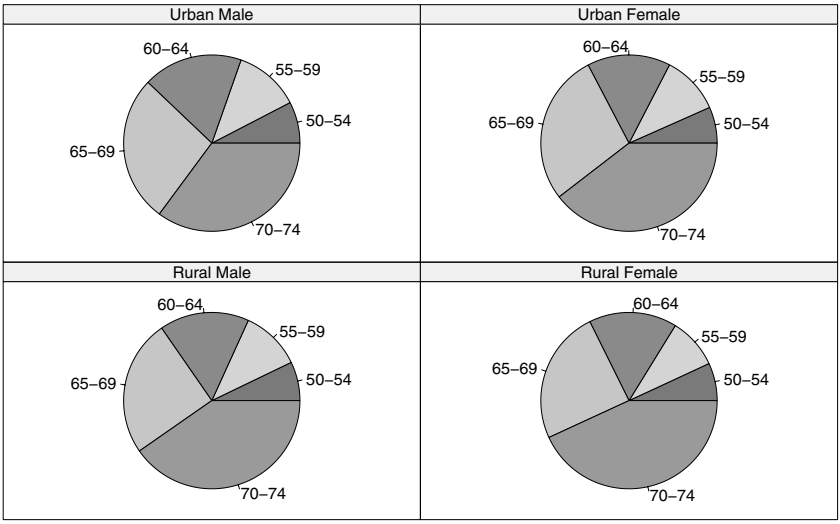

Figure 14.5. Conditional pie charts of the `VADeaths` data, created reusing the traditional graphics function `pie()` and the gridBase package. Compare with Figure 4.3, which presents the same data using a far more effective design.