# 11

# Manipulating the *"trellis"* Object

The Trellis paradigm is different from traditional R graphics in an important respect: high-level "plotting" functions in lattice produce objects rather than any actual graphics output. As with other objects in R, these objects can be assigned to variables, stored on disk in serialized form to be recovered in a later session, and otherwise manipulated in various ways. They can also be plotted, which is all we want to do in the vast majority of cases. Throughout this book, we have largely focused on this last task. In this chapter, we take a closer look at the implications of the object-based design and how one might take advantage of it.

## 11.1 Methods for *"trellis"* objects

The S language features its own version of object-oriented programming. To make things somewhat complicated, it has two versions of it: the *S3* or old-style version, and the newer, more formal *S4* version. The fundamental concepts are similar; objects have classes, and some functions are generic, with specific methods that determine the action of the generic when its arguments are objects of certain classes. However, the tools one can use to obtain information about a class or methods of a generic function are different. The lattice package is implemented using the *S3* system,[1] and the tools we describe in this section are specific to it.

The objects returned by high-level functions such as `xyplot()` have class *"trellis"*. We can obtain a list of methods that act specifically on *"trellis"* objects using

```
> methods(class = "trellis")
 [1] dimnames<-.trellis* dimnames.trellis*   dim.trellis*
 [4] plot.trellis*       print.trellis*      summary.trellis*
 [7] tmd.trellis*        [.trellis*          t.trellis*
```

[1] Although it is possible to extend it to *S4*, as we show in Chapter 14.

```
[10] update.trellis*

   Non-visible functions are asterisked
```

The output is useful primarily because it tells us where to look for documentation; for example, the documentation for the `dimnames()` method can be accessed by typing

```
> help("dimnames.trellis")
```

and that for the `[` method by typing

```
> help("[.trellis")
```

Note that this does not give a list of all generic functions that can act on *"trellis"* objects; for instance, `str()` is a generic function with no specific method for *"trellis"* objects, but a default method exists and that is used instead. These comments are not specific to the *"trellis"* class; for example, we could get a similar list of methods for *"shingle"* objects with

```
> methods(class = "shingle")
[1] as.data.frame.shingle* plot.shingle*
[3] print.shingle*         [.shingle*
[5] summary.shingle*

   Non-visible functions are asterisked
```

and a list for all methods for the generic function `barchart()` using

```
> methods(generic.function = "barchart")
[1] barchart.array*   barchart.default* barchart.formula*
[4] barchart.matrix*  barchart.numeric* barchart.table*

   Non-visible functions are asterisked
```

As hinted at by the output, most of these methods are not intended to be called by their full name. The correct usage is described in the respective help page, or sometimes in the help page for the generic. We now look at some of these methods in more detail.

## 11.2 The `plot()`, `print()`, and `summary()` methods

The most commonly used generic function in R is `print()`, as it is implicitly used to display the results of many top-level computations. For *"trellis"* objects, the `print()` method actually plots the object in a graphics device. It is sometimes necessary to use `print()`[2] explicitly, either because automatic printing would have been suppressed in some context, or to use one of the

[2] Or `plot()`, which is equivalent, except that it does not return a copy of the object being plotted.

optional arguments. The most useful arguments of the `plot()` and `print()` methods are described here briefly.

`split`, `position`
: These two arguments are used to specify the rectangular subregion within the whole plotting area that will be used to plot the *"trellis"* object. Normally the full region is used. The `split` argument, specified in the form `c(col, row, ncol, nrow)`, divides up the region into `ncol` columns and `nrow` rows and places the plot in column `col` and row `row` (counting from the upper-left corner). The `position` argument can be of the form `c(xmin, ymin, xmax, ymax)`, where `c(xmin, ymin)` gives the lower-left and `c(xmax, ymax)` the upper-right corner of the subregion, treating the full region as the $[0,1] \times [0,1]$ unit square.

`more`, `newpage`
: By default, a new "page" is started on the graphics device every time a *"trellis"* object is plotted. These two arguments suppress this behavior, allowing multiple plots to be placed together in a page. Specifying `more = TRUE` in a call causes the next *"trellis"* plot to be on the same page. Specifying `newpage = FALSE` causes the current plot to skip the move to a new page.[3]

`panel.height`, `panel.width`
: These two arguments allow control over the relative or absolute widths and heights of panels in terms of the very flexible unit system in grid. A full discussion of this system is beyond the scope of this book, but we show a simple example soon.

`packet.panel`
: This argument is a function that determines the association between packet order and panel order. The packet order arises from viewing a *"trellis"* object as an array with margins defined by the conditioning variables, with packets being the cells of the array. Just as regular arrays in R, this can be thought of as a vector with a dimension attribute, and the packet order is the linear order of packets in this vector. On the other hand, the panel order is the order of panels in the physical layout, obtained by varying the columns fastest, then the rows, and finally the pages. Specifying `packet.panel` allows us to change the default association rule, which is implemented by the `packet.panel.default()` function, whose help page gives further details and examples.

Other arguments of the `plot()` method are rarely needed and are not discussed here. Note that just as parameter settings normally specified using `trellis.par.set()` can be attached to individual *"trellis"* objects by adding a `par.settings` argument to high-level calls, arguments to the plot method can also be attached as a list specified as the `plot.args` argument.

[3] The latter is more general, as it allows lattice plots to be mixed with other grid graphics output. Specifically, `newpage` must be set to `FALSE` to draw a *"trellis"* plot in a previously defined viewport.

We have seen the use of the `plot()` method previously in Figures 1.4 (where the `split` and `newpage` arguments were used) and 10.11 (where `position` was used). In the next example, we illustrate the use of `more` to compare two common variants of the dot plot. The first step is to create variables representing suitable *"trellis"* objects.

```
> dp.uspe <-
      dotplot(t(USPersonalExpenditure),
              groups = FALSE,
              index.cond = function(x, y) median(x),
              layout = c(1, 5),
              type = c("p", "h"),
              xlab = "Expenditure (billion dollars)")
> dp.uspe.log <-
      dotplot(t(USPersonalExpenditure),
              groups = FALSE,
              index.cond = function(x, y) median(x),
              layout = c(1, 5),
              scales = list(x = list(log = 2)),
              xlab = "Expenditure (billion dollars)")
```

These are then plotted side by side in a $2 \times 1$ layout to produce Figure 11.1.

```
> plot(dp.uspe,     split = c(1, 1, 2, 1), more = TRUE)
> plot(dp.uspe.log, split = c(2, 1, 2, 1), more = FALSE)
```

Another useful method for *"trellis"* objects is `summary()`. For our next example, we create a dot plot similar to the one in Figure 10.21. The response this time is the `Frost` variable, which gives the mean number of days with minimum temperature below freezing between 1931 and 1960 in the capital or a large city in each U.S. state. We begin by defining a suitable data frame and then creating a *"trellis"* object

```
> state <- data.frame(state.x77, state.region, state.name)
> state$state.name <-
      with(state, reorder(reorder(state.name, Frost),
                          as.numeric(state.region)))
> dpfrost <-
      dotplot(state.name ~ Frost | reorder(state.region, Frost),
              data = state, layout = c(1, 4),
              scales = list(y = list(relation = "free")))
```

which we then summarize using the `summary()` method.

```
> summary(dpfrost)
Call:
dotplot(state.name ~ Frost | reorder(state.region, Frost), data = state,
    layout = c(1, 4), scales = list(y = list(relation = "free")))

Number of observations:
reorder(state.region, Frost)
```

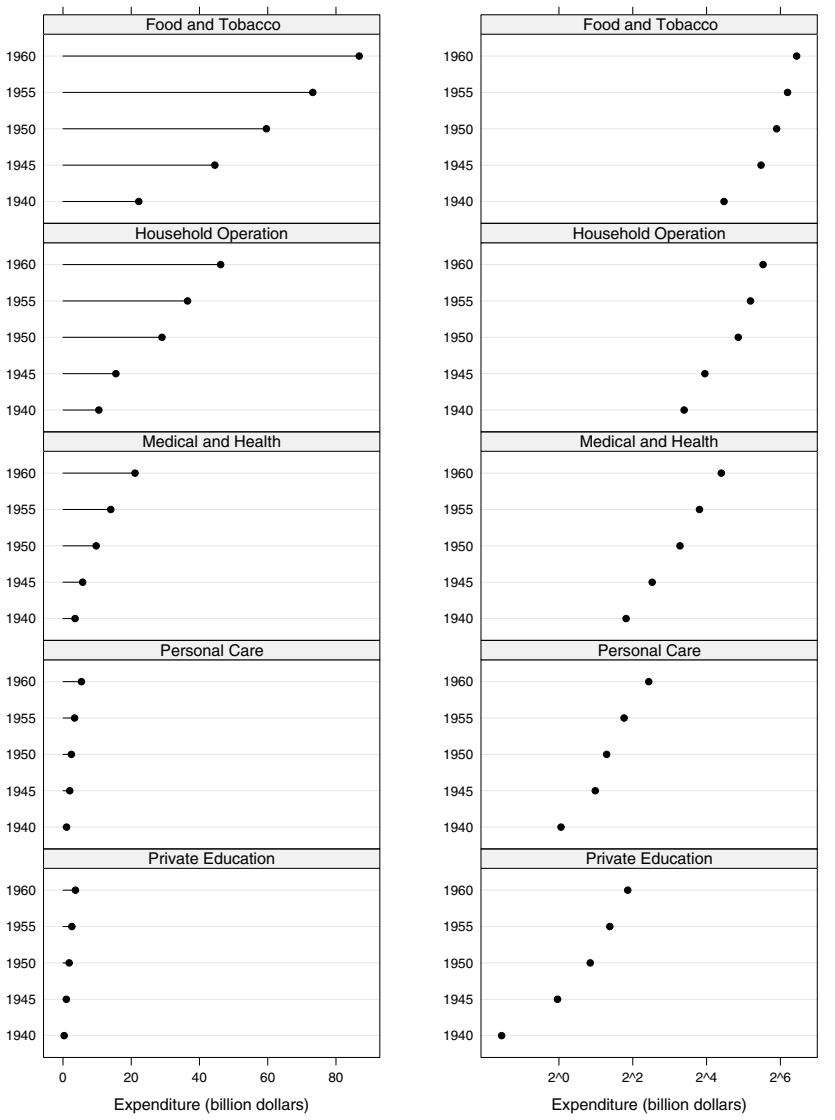


**Figure 11.1.** Two common variants of dot plots, showing trends in personal expenditure (on various categories) in the United States. Lines joining the points to a baseline, as in the plot on the left, are often helpful, but only if a meaningful baseline is available. In this case, patterns in the data are conveyed better by the plot on the right, with the data on a logarithmic scale.

```
        South          West     Northeast North Central
           16            13             9            12
```

The output gives us the call used to produce the object, but more important in this case, it gives us the number of observations (and hence, the approximate range of the $y$-axis) in each panel. We can use these frequencies to change the heights of the panels when plotting the object. Figure 11.2 is produced by[4]

```
> plot(dpfrost,
       panel.height = list(x = c(16, 13, 9, 12), unit = "null"))
```

This is not exactly what we want, as the actual range of the $y$-axis will be slightly different. However, the difference is practically negligible in this case. The `resizePanels()` function, used previously to produce Figure 10.21 and discussed further in the next chapter, does take the difference into account.

## 11.3 The `update()` method and `trellis.last.object()`

Perhaps the most useful method for *"trellis"* objects after `plot()` is `update()`, which can be used to incrementally change many (although not all) arguments defining a *"trellis"* object without actually recomputing the object. We have seen many uses of `update()` throughout this book and only give one more explicit example here.

`update()` is often useful in conjunction with the `trellis.last.object()` function. Every time a *"trellis"* object is plotted, whether explicitly or through implicit printing, a copy of the object is retained in an internal environment (unless this feature is explicitly disabled). The `trellis.last.object()` function can be used to retrieve the last object saved. Thus, the following command will produce Figure 11.3 when issued right after the previous example.

```
> update(trellis.last.object(), layout = c(1, 1))[2]
```

This example also illustrates the indexing of *"trellis"* objects as arrays. The above call recovers the last saved object using `trellis.last.object()` and updates it by changing the `layout` argument. Because the object had four packets, this would normally have resulted in a plot with four pages, but the indexing operator `"["` is used to extract just the second packet.[5]

The indexing of *"trellis"* objects follows rules similar to those for regular arrays. In particular, indices can be repeated, causing packets to be repeated in the resulting plot. A useful demonstration of this feature is given in Figure 11.4, where a three-dimensional scatter plot with a single packet is displayed in multiple panels with gradually changing viewpoints. The figure is produced by

[4] The `"null"` unit is a special grid unit that asks the panels to be as tall as possible while retaining their relative heights. Other units such as `"inches"` or `"cm"` can be used to specify absolute heights.

[5] The `"["` method actually uses the `update()` method to change the `index.cond` argument, but is more intuitive and performs more error checks.

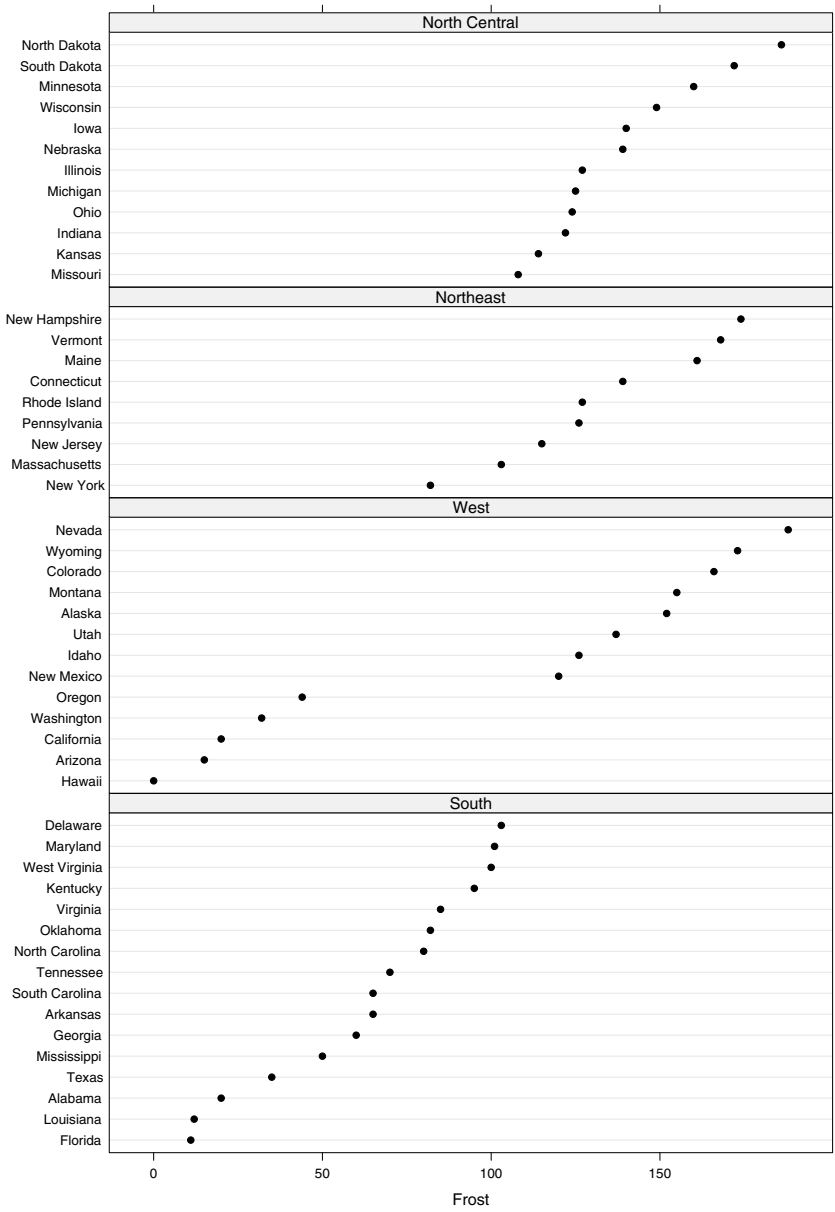


**Figure 11.2.** A dot plot similar to Figure 10.21, using the `Frost` column in the `state.x77` dataset. The heights of panels are controlled using a different method.

```
> npanel <- 12
> rot <- list(z = seq(0, 30, length = npanel),
              x = seq(0, -80, length = npanel))
> quakeLocs <-
      cloud(depth ~ long + lat, quakes, pch = ".", cex = 1.5,
            panel = function(..., screen) {
                pn <- panel.number()
                panel.cloud(..., screen = list(z = rot$z[pn],
                                               x = rot$x[pn]))
            },
            xlab = NULL, ylab = NULL, zlab = NULL,
            scales = list(draw = FALSE), zlim = c(690, 30),
            par.settings = list(axis.line = list(col="transparent")))
> quakeLocs[rep(1, npanel)]
```

The panel function makes use of the `panel.number()` function to detect which panel is currently being drawn. This and other useful accessor functions are described in Chapter 13.

## 11.4 Tukey mean–difference plot

The Tukey mean–difference plot applies to scatter plots and quantile plots. As the name suggests, the idea is to start with a set of $(x, y)$ pairs, and plot the mean $(x + y)/2$ on the $x$-axis and the difference $x - y$ on the $y$-axis. In terms of plotting, this is equivalent to rotating the $(x, y)$ data clockwise by 45°. The mean–difference plot is most useful when the original data lie approximately along the positive diagonal, as its purpose is to emphasize deviations from that line. M–A plots, popular in the analysis of microarray data, are essentially mean–difference plots.

It is fairly simple to create a mean–difference plot using `xyplot()` after manually transforming the data. As a convenience, the `tmd()` function performs this transformation automatically on *"trellis"* objects produced by `xyplot()`, `qqmath()`, and `qq()`. In the following example, we apply it to the two sample Q–Q plot seen in Figure 3.10. Figure 11.5, produced by

```
> data(Chem97, package = "mlmRev")
> ChemQQ <-
      qq(gender ~ gcsescore | factor(score), data = Chem97,
         f.value = ppoints(100), strip = strip.custom(style = 5))
> tmd(ChemQQ)
```

suggests that the distributions of `gcsescore` for girls and boys differ consistently in variance except for the lowest `score` group.

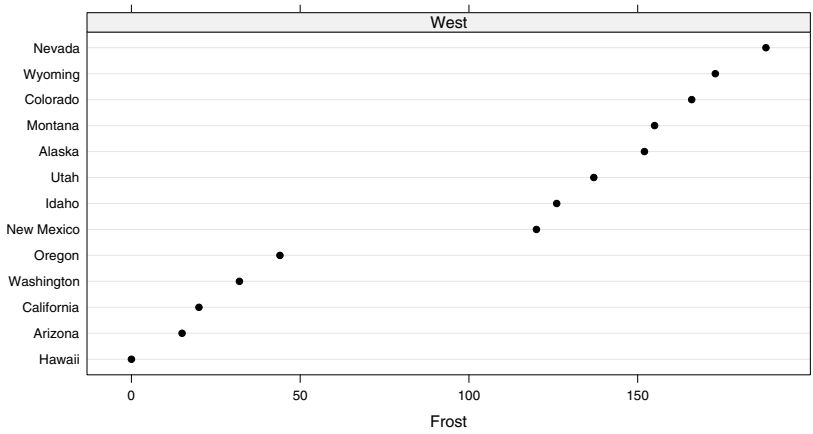


**Figure 11.3.** One panel from Figure 11.2, extracted from the underlying *"trellis"* object.

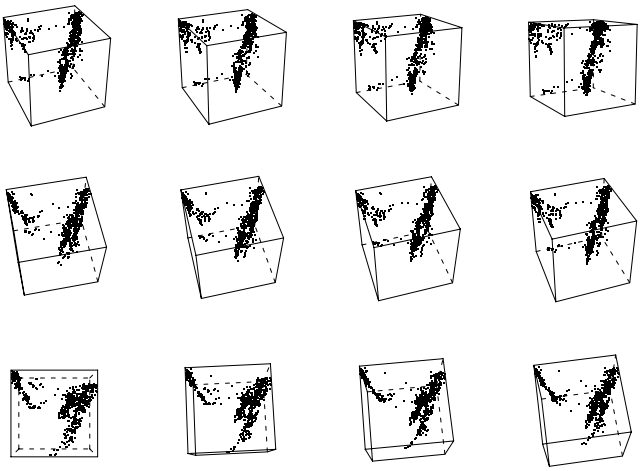

**Figure 11.4.** Varying camera position for a three-dimensional scatter plot of earthquake epicenter positions, from bottom left to top right.

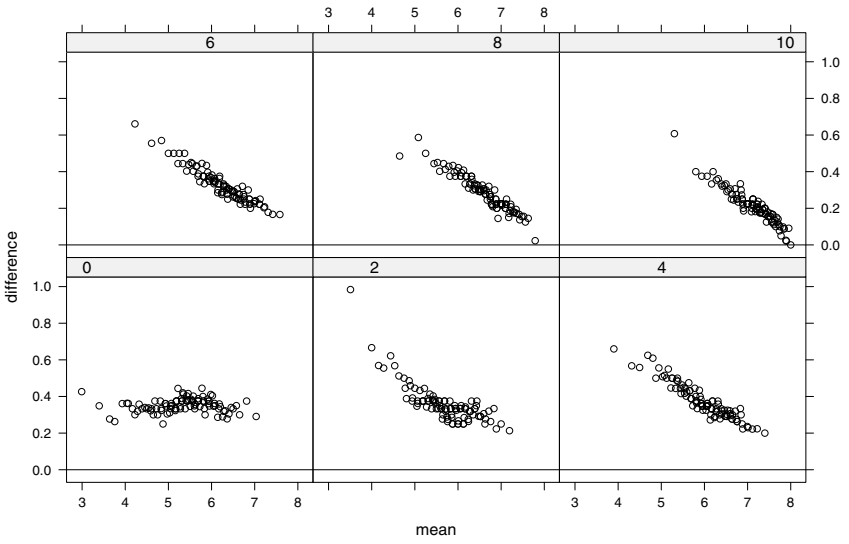


**Figure 11.5.** Tukey mean–difference plot, derived from the two-sample Q–Q plot in Figure 3.10.

## 11.5 Specialized manipulations

The use of strips on top of each panel to indicate levels of conditioning variables, introduced in the original Trellis implementation in S, was a remarkable innovation because it allowed multipanel displays with an arbitrary number of conditioning variables and a layout that is not necessarily tied to the dimensions of the conditioning variable.[6] This generality sometimes makes it difficult to implement designs that are perhaps more useful in special cases. For example, in a multipanel display with exactly two conditioning variables and the default layout (columns and rows representing levels of the first and second conditioning variables), one might want to indicate the levels only on the outer margins, once for each row and column, rather than in all panels. It is possible to realize such a design with lattice, but this requires far more detailed knowledge than warranted. Fortunately, the object model used in lattice makes it fairly simple to write functions that implement such specialized manipulations in a general way. In the next example, we make use of the `useOuterStrips()` function in the latticeExtra package, which implements the design described above.

[6] In contrast, "conditioning plots" as previously implemented in the `coplot()` function indicated the association indirectly, and were limited to two conditioning variables.

Our example makes use of the `biocAccess` dataset, encountered previously in Figure 8.2. Here, we attempt to look at the pattern of access attempts over a day conditioned on month and day of the week.

```
> library("latticeExtra")
> data(biocAccess)
> baxy <- xyplot(log10(counts) ~ hour | month + weekday, biocAccess,
                 type = c("p", "a"), as.table = TRUE,
                 pch = ".", cex = 2, col.line = "black")
```

Just for fun, we note using the `dimnames()` method that the levels of the `month` variable are abbreviated month names, and change them to be the full names.

```
> dimnames(baxy)$month
[1] "Jan" "Feb" "Mar" "Apr" "May"
> dimnames(baxy)$month <- month.name[1:5]
> dimnames(baxy)
$month
[1] "January"  "February" "March"    "April"    "May"

$weekday
[1] "Monday"    "Tuesday"   "Wednesday" "Thursday"  "Friday"
[6] "Saturday"  "Sunday"
```

Of course, we could also have done this by writing a (fairly complicated) custom strip function, or more simply by modifying the levels of `month` beforehand. Finally, we call the `useOuterStrips()` function to produce a modified *"trellis"* object, which produces Figure 11.6.

```
> useOuterStrips(baxy)
```

Although not clear from this example, `useOuterStrips()` throws an error unless the requested manipulation is meaningful, and overrides any previously set layout.

## 11.6 Manipulating the display

Traditional R graphics encourages, and even depends on, an incremental approach to building graphs. For example, to create custom axis labels with traditional graphics, one would first create a plot omitting the axes altogether, and then use the `axis()` function, and perhaps the `box()` function, to annotate the axes manually. Trellis graphics, on the other hand, encourages the whole object paradigm, and the operation of updating serves as the conceptual analogue of incremental changes. The obvious advantage to this approach is that unlike traditional graphics, lattice displays can automatically allocate the space required for long axis labels, legends, and the like, because the labels or legends are known before plotting begins.

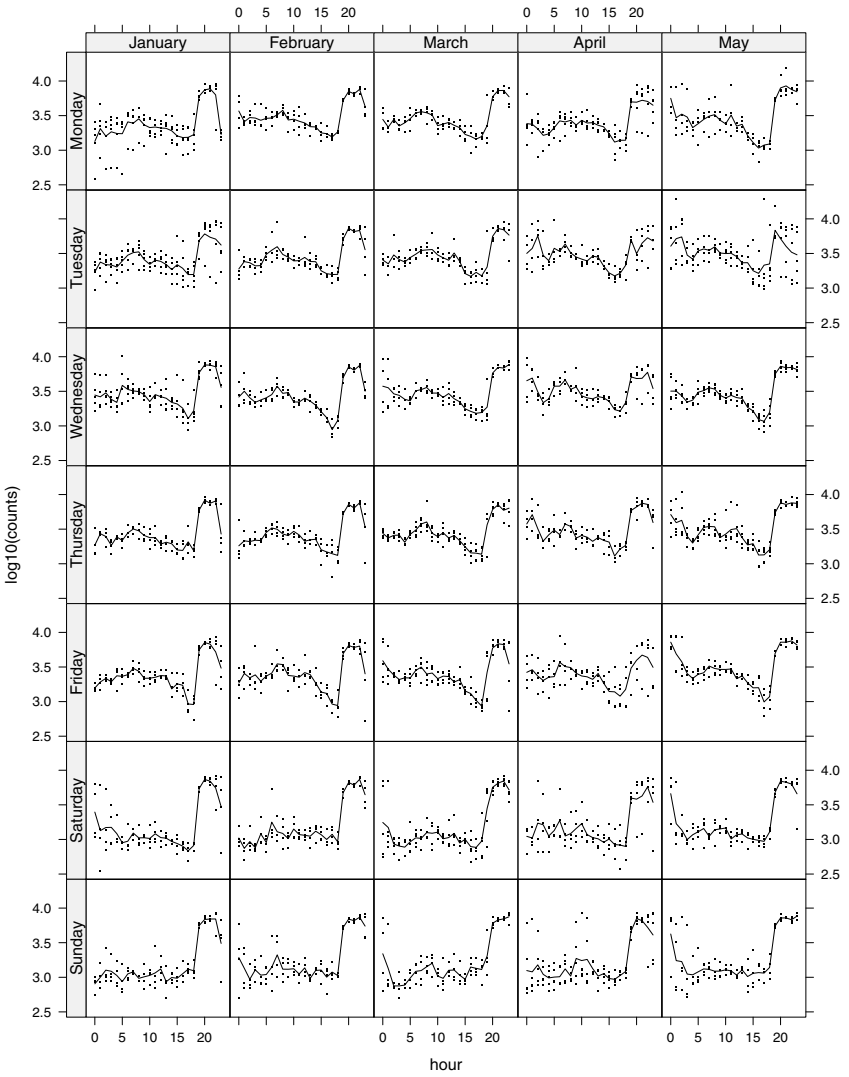


**Figure 11.6.** The number of hourly accesses to `http://www.bioconductor.org`, conditioning on month and day of the week. There is a difference in the patterns in weekdays and weekends, which can be seen more clearly in Figure 14.2. The strips have been manipulated so that they appear only on the top and the left of each column and row, rather than in each panel. This is a useful space-saving device when exactly two conditioning variables are used in the default layout. A color version of this plot is also available.

In some situations however, incremental additions are a necessary part of the workflow; for example, when one wants to identify and label certain "interesting" points in a scatter plot by clicking on them. This does not involve manipulation of the underlying object itself, but rather interaction with its visual rendering. The interface for such interaction is described in the next chapter.