12

Interacting with Trellis Displays

High-level functions in lattice produce *"trellis"* objects that can be thought of as abstract representations of visualizations. An actual rendering of a visualization is produced by plotting the corresponding object using the appropriate `print()` or `plot()` method. In this chapter, we discuss things the user can do after this plotting has been completed.

One possible approach is to treat the result as any other graphic created using the grid package, and make further enhancements to the display using the low-level tools available in grid. In particular, the display consists of a tree of viewports, and various grid graphical objects (grobs) drawn within them. The user can move down to any of these viewports and add further objects, or, less commonly, edit the properties of existing objects. The precise details of these operations are beyond the scope of this book, but are discussed by Murrell (2005). In this chapter, we focus entirely on a higher-level interface in the lattice package for similar tasks, which is less flexible,[1] but usually sufficient. The playwith package (Andrews, 2007) provides a user-friendly GUI wrapper for many of these facilities.

12.1 The traditional graphics model

In the traditional R graphics model, displays are often built incrementally. An initial plot is created using a high-level function (such as `boxplot()`), and further commands, such as `lines()` and `axis()`, add more elements to the existing display. This approach works because there is exactly one figure region, and there is no ambiguity regarding which coordinate system is to be used for additional elements. Things are not as simple in a multipanel Trellis display, as one needs the additional step of determining to which panel further increments should apply.

[1] In particular, it provides no facilities for editing existing graphical objects in the manner of `grid.edit()`.

The recommended approach in the Trellis system is to encode the display using the panel function. This ties in neatly with the idea of separating control over different elements of a display; in this paradigm, the panel function represents the procedure that visually encodes the data. In some ways, this takes the incremental approach to the extreme; a panel starts with a blank canvas, with only the coordinate system set up, and the panel function is responsible for everything drawn on it. An apparent deficiency of this model is that the only "data" available to the panel function is the packet produced by the conditioning process. In practice, further data can be passed in through the `...` argument, and panel-specific parts can be extracted if necessary using the `subscripts` mechanism and accessor functions such as `packet.number()` and `which.packet()` (see Chapter 13). A more real deficiency is that this paradigm does not include any reasonable model for interaction.

12.1.1 Interaction

Native R graphics has rather limited support for interaction, but what it does have is often useful. The primary user-level functions related to interaction in traditional R graphics are `locator()` and `identify()`. `locator()` is a low-level tool, returning locations of mouse clicks, and `identify()` is a slightly more specialized function that is used to add text labels to a plot interactively.

The grid analogue of `locator()` is `grid.locator()`, which returns the location of a single mouse click in relation to the currently active viewport. lattice uses `grid.locator()` to provide a largely API-compatible analogue of `identify()` called `panel.identify()`, along with a couple of other similar functions. However, before we can illustrate the use of these functions, we need some more background on the implementation of lattice displays.

12.2 Viewports, `trellis.vpname()`, and `trellis.focus()`

An elementary understanding of grid viewports is necessary to appreciate the API for interacting with lattice plots. Viewports are essentially arbitrary rectangular regions inside which plotting can take place. For our purposes, their most important feature is that they have an associated coordinate system.[2] The process of plotting a *"trellis"* object involves the creation of a number of viewports; for example, every panel, strip, and label has its own viewport. These viewports are retained after plotting is finished, and the associated viewport tree (showing the nesting of viewports within other viewports) can be displayed by calling

```
> library("grid")
> current.vpTree()
```

[2] Points in this coordinate system can be represented in a variety of units, see `?unit` in the grid package for details.

To add to the display in a particular viewport (usually one corresponding to a panel), we need to first make it the active viewport.

Every viewport has a name that can be used to revisit it (using the grid functions `downViewport()` and `seekViewport()`). To make the viewport names predictable, lattice uses the function `trellis.vpname()` to create the relevant names. For example, the names of the x-label viewport and the strip viewport at column 2 and row 1 might be

```
> trellis.vpname("xlab", prefix = "plot1")
[1] "plot1.xlab.vp"
> trellis.vpname("strip", column = 2, row = 1, prefix = "plot2")
[1] "plot2.strip.2.1.vp"
```

where the `prefix` argument is a character string that potentially allows viewports for multiple *"trellis"* displays on a page to be distinguished from each other. However, the user does not typically need to know this level of detail and can instead use the functions `trellis.focus()` and `trellis.unfocus()` to navigate the viewport tree.

The viewport that is active after a *"trellis"* object has been plotted is the one in which the plotting started (this is usually the root viewport that covers the entire device). The `trellis.focus()` function is used to make a different viewport in the viewport tree active. For example, the panel viewport at column 2 and row 1 might be selected by calling

```
> trellis.focus("panel", column = 2, row = 1)
```

Most arguments of `trellis.vpname()` can be supplied to `trellis.focus()` directly. In addition, it checks for invalid `column` and `row` values and gives an informative error message if necessary. More important, it makes the most common uses slightly simpler. With a single panel display, simply calling

```
> trellis.focus()
```

with no arguments selects the panel. For a multipanel display (on an interactive screen device), the same call allows the user to choose a panel by clicking on it. The viewport chosen by `trellis.focus()` is highlighted by default, making it easy to identify for further interaction. Many of these details can be controlled by additional arguments to `trellis.focus()`. Finally, calling

```
> trellis.unfocus()
```

reverts to the original viewport after undoing any highlighting.

12.3 Interactive additions

Once the desired viewport is active, further additions can be made to the display by making suitable function calls. Such additions usually involve interaction. `grid.locator()` can be used to identify locations of individual mouse clicks, which then need to be handled appropriately. A typical use of

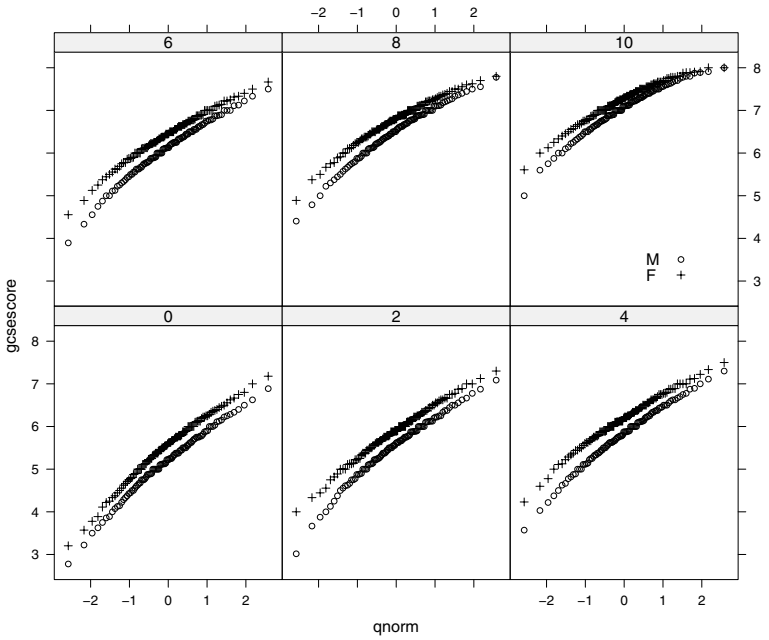

Figure 12.1. A normal quantile plot of `gcsescore` conditioning on `score` and grouping by `gender`. The legend describing the group symbols has been placed inside the plot interactively by clicking on the desired position.

this is to place a legend interactively on a plot. For example, the following code might produce Figure 12.1 after the user clicks on a suitable location.

```
> data(Chem97, package = "mlmRev")
> qqmath(~ gcsescore | factor(score), Chem97, groups = gender,
         f.value = function(n) ppoints(100),
         aspect = "xy",
         page = function(n) {
             cat("Click on plot to place legend", fill = TRUE)
             ll <- grid.locator(unit = "npc")
             if (!is.null(ll))
                 draw.key(simpleKey(levels(factor(Chem97$gender))),
                          vp = viewport(x = ll$x, y = ll$y),
                          draw = TRUE)
         })
```

In this example, the `page` argument has been used to encapsulate the process of asking for a user click and using the result to draw a suitable legend. The `draw.key()` function is normally used to construct a legend, as discussed in Chapter 9, but here it also draws the legend inside a newly created viewport.

The grid function `viewport()` is used to create the temporary viewport on the fly; the new viewport is centered on the location of the mouse click. We did not need to use `trellis.focus()` because we were not adding to any specific panel.

More complicated interaction modes can be built around `grid.locator()`. lattice provides three (at the time of writing) built-in functions that implement somewhat specialized forms of interaction. These are `panel.identify()`, `panel.identify.qqmath()`, and `panel.link.splom()`. We start with an illustration of `panel.identify()`, which is intended to be used with scatter plots as produced by `xyplot()` to add labels to points chosen interactively. Figure 12.2, showing a scatter plot of the murder rate against life expectancy in U.S. states with a few states labeled, might be the result of

```
> state <- data.frame(state.x77, state.region)
> xyplot(Murder ~ Life.Exp | state.region, data = state,
         layout = c(2, 2), type = c("p", "g"), subscripts = TRUE)
> while (!is.null(fp <- trellis.focus())) {
      if (fp$col > 0 & fp$row > 0)
          panel.identify(labels = rownames(state))
  }
```

There are several points that merit explanation in this sequence of calls. The first is the use of the `subscripts = TRUE` argument in `xyplot()` call. As noted in Section 5.2, panel functions can request an argument called `subscripts` that would contain the indices defining the rows of the data which form the packet in a given panel. Our intention is to label points using the corresponding state names, which are obtained from the row names of the `state` data frame. This represents names for all the data points, whereas we need names that correspond to the states used in individual panels. Obviously, `subscripts` gives us the right set of indices to extract the appropriate subset. Unfortunately, the subscripts are normally not retained if the panel function does not explicitly have an argument called `subscripts`. Specifying `subscripts = TRUE` in the high-level call forces retention of the subscripts, and is advisable for any call that is to be followed by interactive additions.

The next point of note is the use of `trellis.focus()` inside a `while()` loop. As mentioned earlier, calling `trellis.focus()` without arguments allows the user to select a panel interactively. Such a selection can be terminated by a right mouse button click (or by pressing the `ESC` key for the `quartz` device), in which case `trellis.focus()` returns `NULL`. We use this fact to repeatedly select panels until the user explicitly terminates the process in this manner. The user could also click outside the panel area, or on an empty panel; in this case, `trellis.focus()` returns a list with the `row` and `col` components set to 0 (for a normal selection, these would contain the location of the selection). We make sure that a valid selection has been made before we call `panel.identify()` to interactively label points inside the panel.

The final point is the use of `panel.identify()` every time a panel is successfully selected. When called, it allows the user to click on or near points in

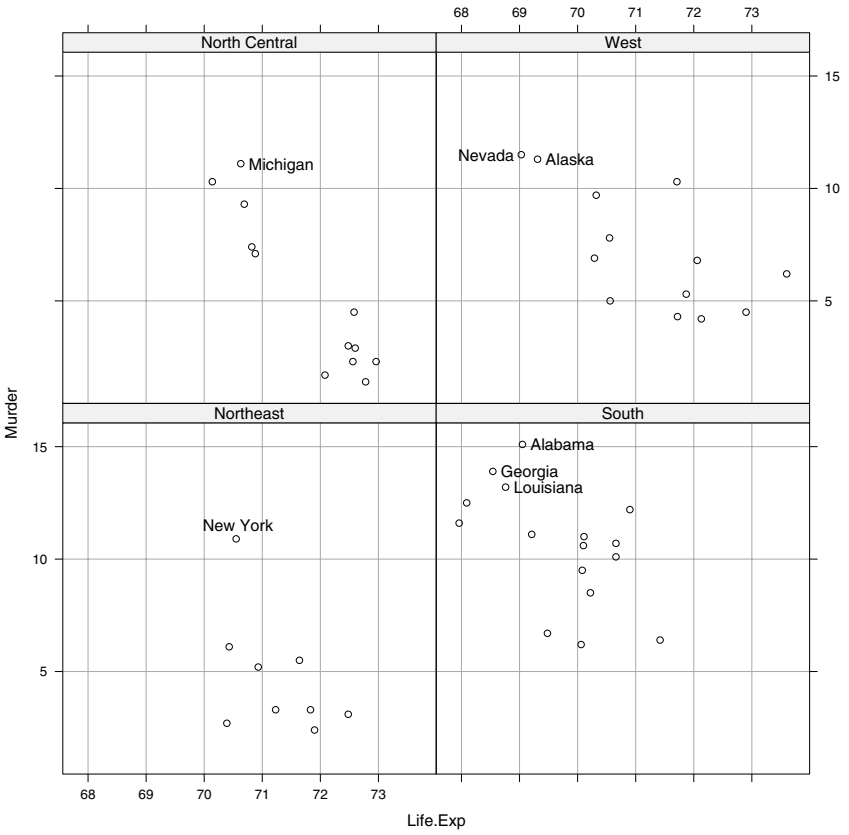

Figure 12.2. A scatter plot of murder rate versus life expectancy in U.S. states by region. In each panel, one or more states have been identified (labeled) by interactively selecting the corresponding points.

the selected panel to label them. This process continues until all points are labeled, or the process is explicitly terminated. Our call to `panel.identify()` specifies only one argument, `labels`, containing the labels associated with the full dataset. To make use of these labels, `panel.identify()` also needs to know the coordinates of the data points associated with these labels, and possibly the subscripts that need to be applied before the association is made. These arguments can be supplied to it as the `x`, `y`, and `subscripts` arguments. When `panel.identify()` is called after a call to `trellis.focus()` (or inside the panel function), these arguments may be omitted; they default to the corresponding arguments that would have been supplied to the panel function. Thus, the appropriate choice is made in every panel without explicit involvement of the user. This automatic selection is made using the

`trellis.panelArgs()` function, which in turn uses `trellis.last.object()` to retrieve the last *"trellis"* object plotted. The correct set of arguments is determined using the accessor function `packet.number()`. This and other similar accessor functions are described more formally in Chapter 13.

Our next example illustrates the use of `panel.identify.qqmath()`, which is designed to add labels to a quantile plot produced by `qqmath()`. Figure 12.3 is produced by (after the appropriate pointing and clicking by the user)

```
> qqmath(~ (1000 * Population / Area), state,
         ylab = "Population Density (per square mile)",
         xlab = "Standard Normal Quantiles",
         scales = list(y = list(log = TRUE, at = 10^(0:3))))
> trellis.focus()
> do.call(panel.qqmathline, trellis.panelArgs())
> panel.identify.qqmath(labels = row.names(state))
> trellis.unfocus()
```

Most of the remarks concerning the previous example also apply here. Because the display has only one panel, calling `trellis.focus()` selects it automatically, and no interaction is required. An interesting addition is the call to `panel.qqmathline()`, through `do.call()`, which causes a reference line to be added as if `panel.qqmathline()` had been called as part of the panel function. This time, the correct panel arguments need to be retrieved explicitly using `trellis.panelArgs()`. This approach allows us to make incremental additions to individual panels of a lattice display, much as with the traditional graphics model. This facility is sometimes useful, although its regular use is not recommended as it detracts from the ideal of the *"trellis"* object as an abstraction of the entire graphic.

Our last example of interaction involves the `panel.link.splom()` function, which is designed to work with displays produced by `splom()`. When called, the user can click on a point in any of the subpanels to highlight the corresponding observation in all subpanels. Figure 12.4 is produced by

```
> env <- environmental
> env$ozone <- env$ozone^(1/3)
> splom(env, pscales = 0, col = "grey")
> trellis.focus("panel", 1, 1, highlight = FALSE)
> panel.link.splom(pch = 16, col = "black")
> trellis.unfocus()
```

The `trellis.focus()` call here explicitly chooses a panel, removing any possibility of interaction (although this is redundant in this case as there is only one panel). In addition, setting `highlight = FALSE` ensures that no decoration is added; without it, the display would have been redrawn when the call to `trellis.unfocus()` removed the decoration.

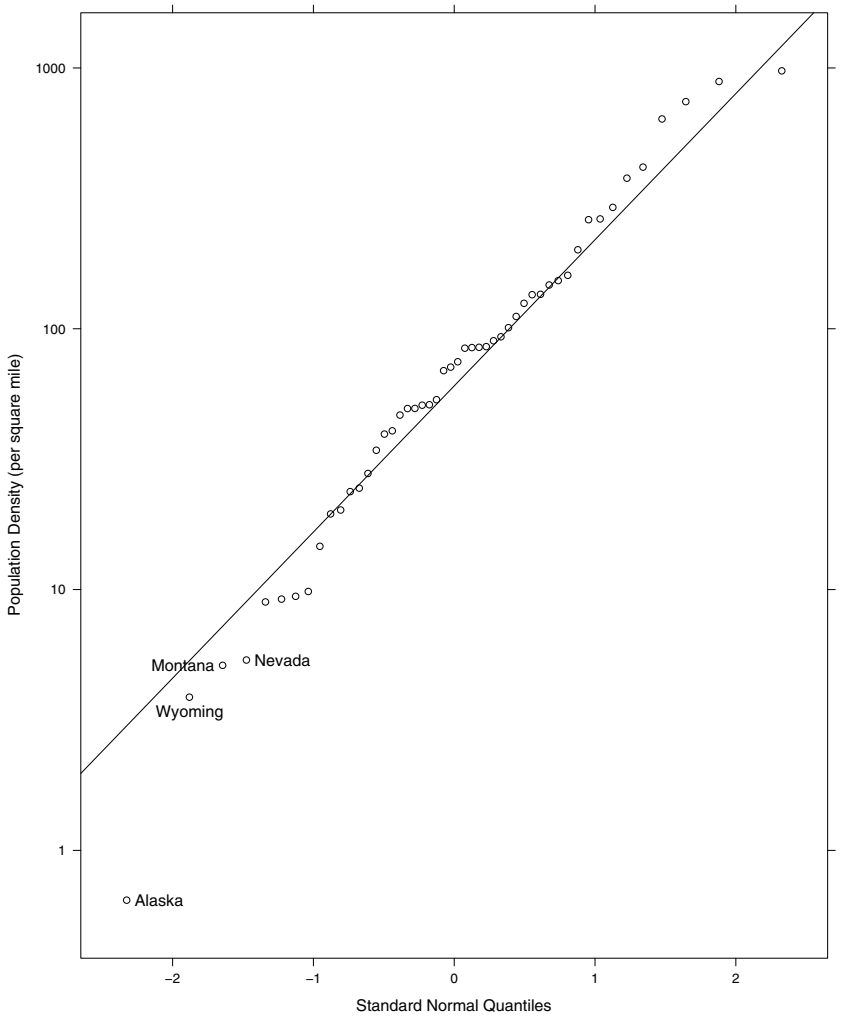

Figure 12.3. Normal quantile plot of population density in U.S. states. Some states have been labeled interactively after adding a reference line through the first and third quartile pairs.

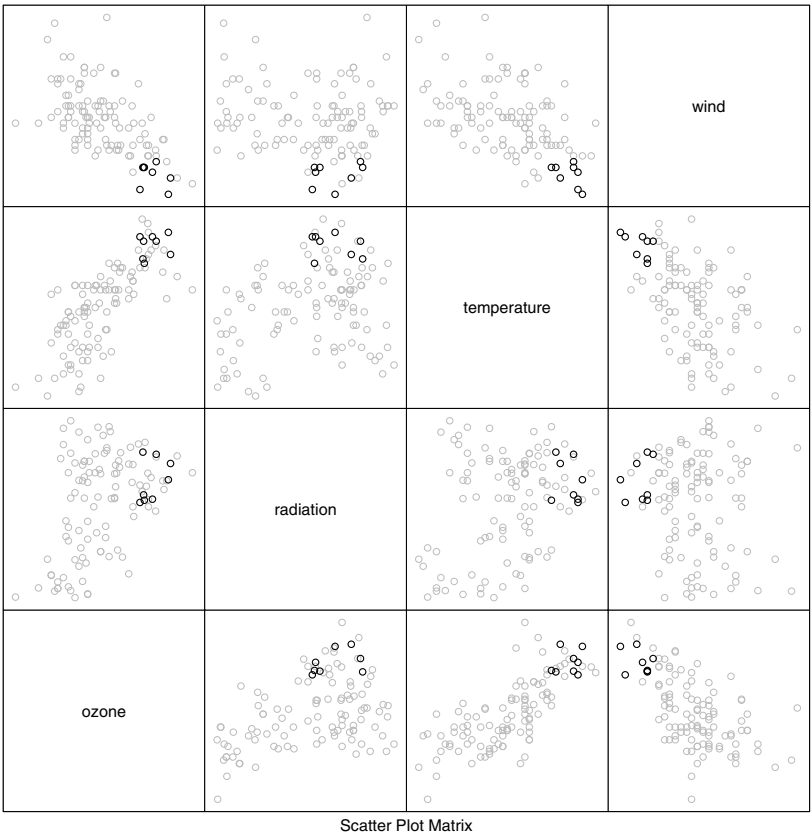

Figure 12.4. Interaction with a scatter-plot matrix. Clicking on a point highlights the corresponding observation in all subpanels.

12.4 Other uses

As we have already seen, it is possible to add pieces to a lattice display non-interactively after it has been drawn. Such use is often convenient, although the same effect can usually be achieved with a suitable panel function. Often, it is useful to simply interrogate a display to obtain information that is not easily available otherwise. For example, consider Figure 11.2, which is a dot plot of the mean number of days with minimum temperature below freezing in the capital or a large city in each U.S. state, conditioning on region. We reproduce the plot in Figure 12.5, but use the same height for every panel initially.

```
> state$name <- with(state,
                     reorder(reorder(factor(rownames(state)), Frost),
                             as.numeric(state.region)))
```

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

Now that the graphic has been plotted, we can obtain the physical layout of panels in the display using the `trellis.currentLayout()` function (see Chapter 13)

```
> trellis.currentLayout()
     [,1]
[1,]    1
[2,]    2
[3,]    3
[4,]    4
```

and use it to compute the exact height of each panel in its native coordinate system:

```
> heights <-
      sapply(seq_len(nrow(trellis.currentLayout())),
             function(i) {
                 trellis.focus("panel", column = 1, row = i,
                               highlight = FALSE)
                 h <- diff(current.panel.limits()$ylim)
                 trellis.unfocus()
                 h
             })
> heights
[1] 16.2 13.2  9.2 12.2
```

It is now trivial to redraw the plot with the physical heights of the panels exactly proportional to their native heights, as was the intent of Figure 11.2. The following produces Figure 12.6.

```
> update(trellis.last.object(),
         par.settings = list(layout.heights = list(panel = heights)))
```

The `resizePanels()` function in the latticeExtra package, used for the same purpose to produce Figure 10.21, is simply this algorithm implemented with some sanity checks.

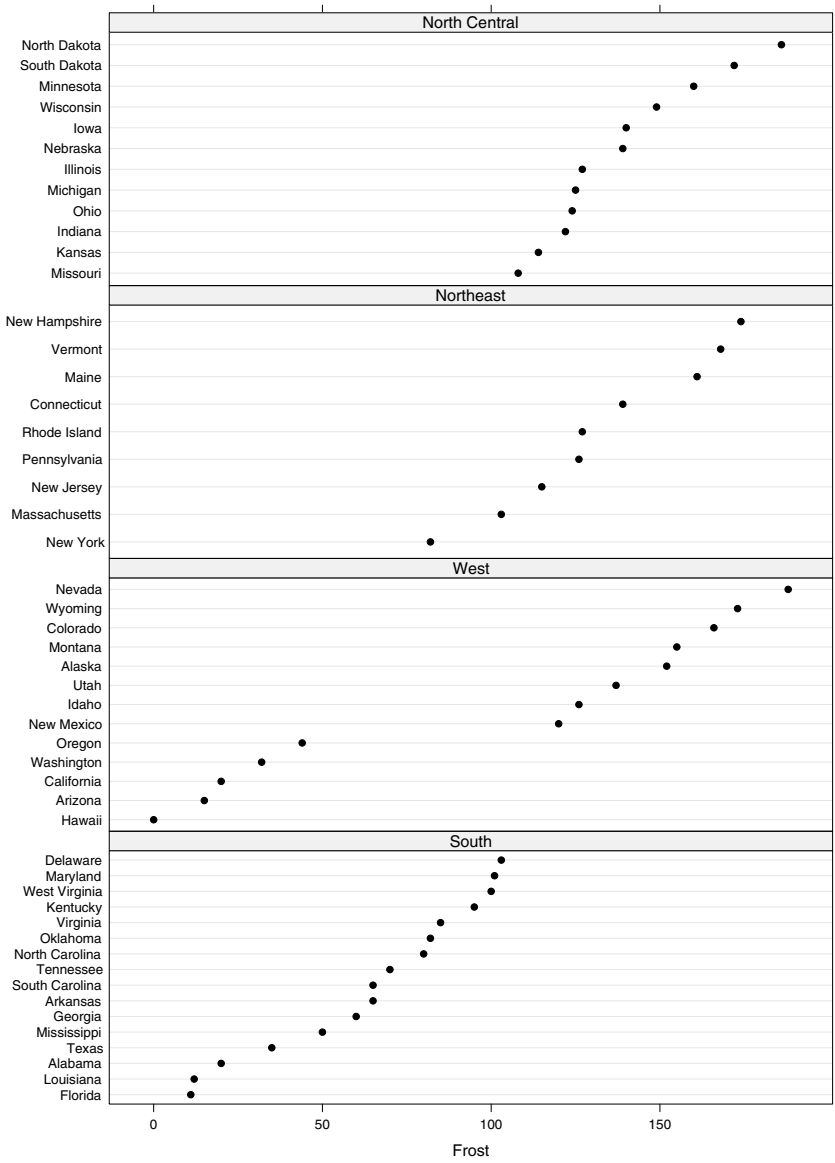

Figure 12.5. Redisplay of Figure 11.2, showing average number of days below freezing in U.S. states, conditioned on geographical region. Each panel has a different number of states, but the same physical height.

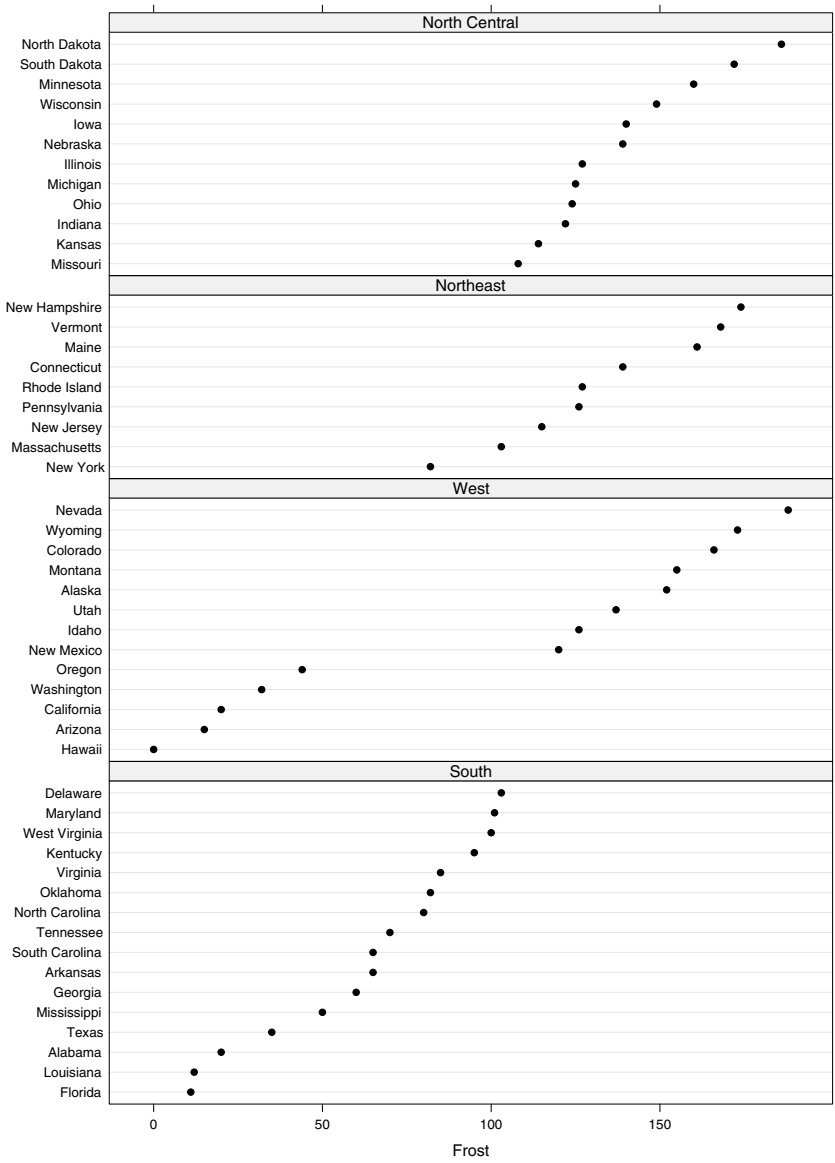

Figure 12.6. Updated form of Figure 12.6, with the physical heights of panels exactly proportional to native heights.